EMULAÇÃO DUM PROCESSADOR ASSOCIATIVO VECTORIAL COM A PLATAFORMA EDgAR

António J. Esteves
Dep. Informática, Universidade do Minho, 4710 Braga, Portugal
Email: esteves@di.uminho.pt

18 de Maio 1996

Relatório Técnico UMDITR9602

Resumo

Descreve-se um emulador do GLiTCH, um processador associativo vectorial, com uma plataforma vocacionada para projectar sistemas com componentes de h/w e de s/w. Faz-se uma descrição breve da estrutura do GLiTCH e duma arquitectura em que ele pode ser aplicado. Mostra-se como na implementação do emulador se utilizou a metodologia de co-projecto de h/w e s/w, embora algumas tarefas sejam executadas manualmente. Apresentam-se as principais fases do projecto do emulador: a partição em componentes a implementar em s/w e em h/w, e a síntese destas componentes e da sua interface. Conclui-se que, dada a melhoria de desempenho obtida com o emulador relativamente a um simulador, a metodologia utilizada é válida. Descreve-se também a linguagem *assembly* desenvolvida para evitar ter que programar o GLiTCH com microcódigo.

Palavras Chave: *emulador, processador associativo vectorial, co-projecto de h/w e s/w*

Índice

1 Introdução

Neste documento apresenta-se a emulação do GLiTCH, um processador associativo vectorial vocacionado para visão por computador. A especificação deste processador foi introduzida noutros trabalhos, por isso aqui deu-se mais atenção à forma de emulação do que à arquitectura do processador e do sistema em que se insere [DST+89].
O processador GLiTCH, sendo um processador associativo vectorial que contém vários elementos de processamento de 1 bit (PEs), difere dos vectores do tipo SIMD no tipo de memória ligada a cada PE: no GLiTCH a memória associada a cada PE são 64 bits de memória identificada pelo seu conteúdo e não pela sua posição física.
Uma possível arquitectura baseada no processador GLiTCH é apresentada de forma simplificada na figura 1. Nesta arquitectura, os GLiTCH formam um caminho de dados entre um dispositivo fonte (câmara de vídeo, unidade de armazenamento,...) e um dispositivo de saída (monitor, unidade de armazenamento,...). Como seria de esperar, o "encaminhador de dados"permite obter um interligação entre os vários processadores mais flexível do que uma ligação fixa (tipo cubo, ou outra) e que permite maior taxa de transferência de dados, ao permitir ligar directamente dois processadores que não são vizinhos na cadeia.
O bloco "controlador"tem como missão o controlo de todo o sistema e inclui como componentes principais: (i) um "sequenciador de instruções"que determina qual o microcódigo a enviar para cada GLiTCH e que gere situações de salto determinadas pelas *flags* dos processadores; (ii) uma unidade que guarda o microcódigo para cada processador; (iii) um *transputer* que disponibiliza processamento escalar adicional; (iv) um "hospedeiro"que difunde as instruções para os PEs de cada processador.
A estrutura do GLiTCH inclui 5 blocos principais: um vector de 64 elementos de processamento de 1 bit (**PEs**), uma zona de memória associada a cada PE com 64 posições (**CAM**), um encaminhador de padrões entre o exterior do *chip* e a CAM (**PBL**), um descodificador de microinstruções (**ROM**) e um registo de deslocamento com 64 por 8 posições (**VSR**). A figura 2 mostra esta estrutura.

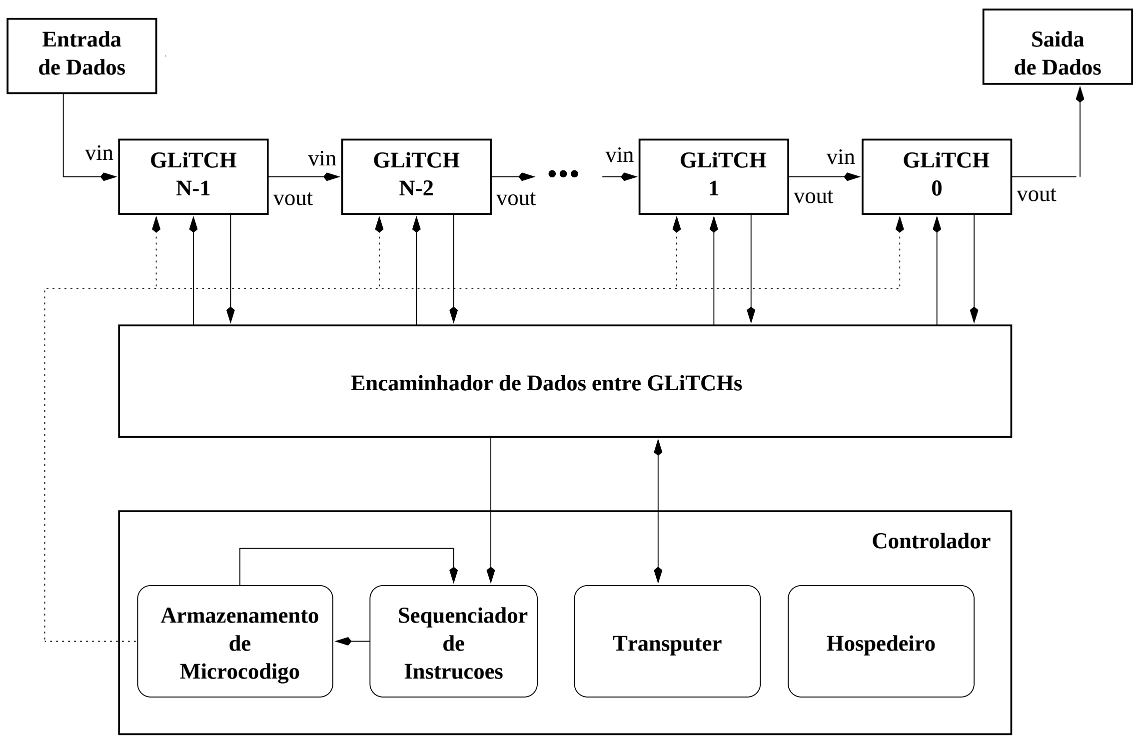

Figura 1: Arquitectura para visão por computador baseada no processador GLiTCH.

Dispondo-se actualmente da arquitectura descrita na figura 1 com a excepção dos *chips* GLiTCH,

pensou-se na sua emulação através da plataforma EDgAR também disponível. Esta plataforma inclui componentes ditas de s/w (processador *transputer*) e componentes ditas de h/w (FPLDs) [Est94]. Deste modo, pode utilizar-se a técnica de co-projecto de h/w-s/w para projectar o emulador, obter uma versão com maior desempenho que um simulador e que permite uma interface de h/w entre o GLiTCH e a arquitectura de visão referida.
O projecto de emulação aqui apresentado surge na sequência doutro projecto em que se simulou o GLiTCH [Alv92]. A solução utilizada na emulação consistiu essencialmente na implementação das microinstruções que caracterizam o GLiTCH através de código que corre no *transputer* ou de lógica das FPLDs do EDgAR. Dado que programar sob a forma de microinstruções se revelou uma tarefa ingrata, escreveu-se uma linguagem *assembly* de mais alto nível que o microcódigo, logo mais compreensível e que não exige um conhecimento tão detalhado da arquitectura do GLiTCH.

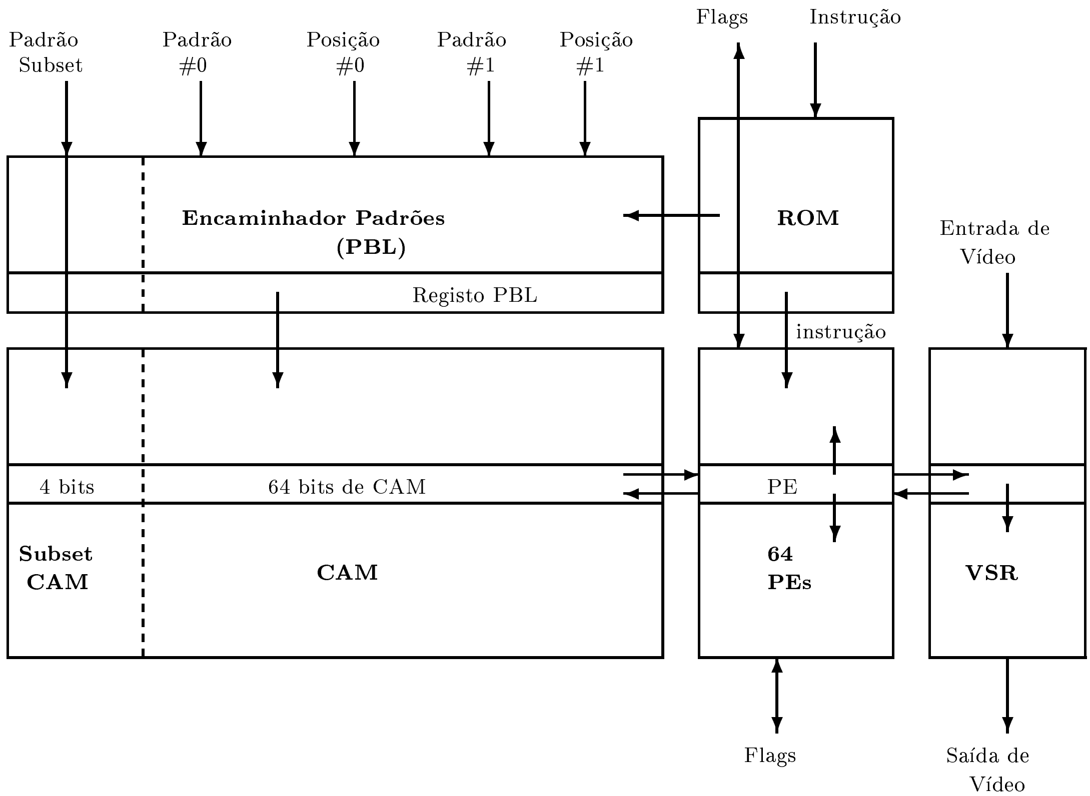

Figura 2: Estrutura do processador GLiTCH com 5 blocos.

A exposição está estruturada em 4 partes. Na parte 2 apresenta-se a implementação do emulador, com destaque para a partição em componentes de h/w de s/w, a descrição e implementação das partições resultantes e da interface entre partições. Na parte 3 expõe-se a linguagem *assembly* definida para programar o GLiTCH. A parte 4 contém os resultados quantitativos mais significativos do trabalho e na parte 5 enunciam-se melhoramentos ao emulador e as principais conclusões.

2 Implementação do GLiTCH na Plataforma EDgAR

A plataforma EDgAR ao incluir um *transputer*, o qual pode ser ligado a uma arquitectura heterogénea que possua outros *transputers*, e vários FPLDs (MACHs e FPGAs), é adequada para projectar de forma rápida sistemas que podem ser bem complexos, obtendo-se implementações

com partes em s/w e partes em h/w. Este processo designa-se de co-projecto de h/w-s/w.
Quando se dispuser de todas as ferramentas de suporte ao processo de co-projecto, a implementação de sistemas começa com uma descrição de alto nível de todo o sistema, valida-se esta especificação, efectua-se a partição em componentes a implementar em h/w e componentes a implementar em s/w, e por fim faz-se a síntese destas componentes e da sua interface [EFSP95].

2.1 Partição do GLiTCH em Componentes de H/W e de S/W

Não dispondo ainda da ferramenta de partição automática, começou-se o processo de implementação do GLiTCH pela sua partição manual e depois fez-se a síntese das partes a implementar em s/w, das partes a implementar em h/w e da interface.
Elegendo o desempenho do processador como principal critério a utilizar na definição das partições, mas tendo em atenção as particularidades de cada bloco que constitui o processador, chegou-se à seguinte partição: o VSR gera uma partição de h/w e os outros blocos resultam numa partição de s/w. Explica-se agora o porquê destas partições.
Optando por uma granularidade grossa surgem 2 candidatos fortes para implementar em h/w: a CAM e o VSR. Dado que a quantidade de elementos de armazenamento da CAM é elevado (64*68), seriam necessárias 8 FPGAs do tipo incluído na plataforma, a qual só contém 4. Excluindo a CAM, optou-se por implementar apenas o VSR em h/w porque é o bloco que tem o menor custo de comunicação com o resto do GLiTCH, porque numa das duas operações que executa não comunica com o resto do GLiTCH e como é um registo bidimensional que opera nas duas direcções (deslocamento e rotação), se fosse implementado em s/w uma das operações seria muito penalizada em termos de tempo de execução. Como a CAM é implementada em s/w, uma razão para que os PEs e o PBL sejam também implementados em s/w é a forte interligação entre estes blocos e a CAM.

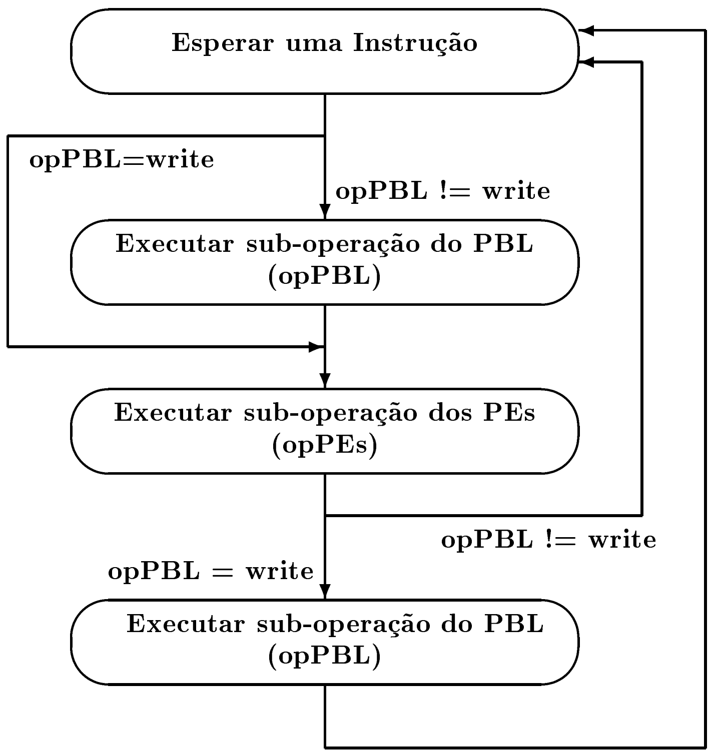

Figura 3: Modelo Utilizado na Emulação do GLiTCH.

2.2 Componentes do GLiTCH a Implementar em S/W

2.2.1 Descrição Funcional das Componentes a Implementar em S/W

As componentes do GLiTCH que são implementadas em s/w, através do *transputer*, são o vector de PEs, a CAM mais a CAM-de-subset com 64*64+64*4 bits, o encaminhador de padrões e o descodificador de microinstruções.
Para justificar como se implementam estas componentes, faz-se um resumo das características mais relevantes.
O descodificador de microinstruções recebe a parte do microcódigo que define as operações do PBL e dos PEs e descodifica-a por forma a enviar ao PBL e aos PEs os sinais adequados à execução da sua quota parte da microinstrução. A sub-operação do PBL é iniciada um ciclo antes da sub-operação dos PEs, e se for uma operação de escrita só termina um ciclo após a sub-operação dos PEs.
O encaminhador de padrões (PBL) permite enviar para a CAMmais a CAM-de-subset 3 (2+1) padrões ternários e 2 posições na CAM, os quais são incluídos no microcódigo e chegam do exterior do *chip*. Normalmente, estes padrões são comparados com o conteúdo da CAM mais a CAM-de-subset nas posições referidas e geram 3 sinais que chegam de cada linha da CAM ao PE correspondente (MD0, MD1 e MS). O encaminhador permite também trocar 32 bits de dados, nos dois sentidos, entre a CAM e o exterior. O registo PBLR alberga os padrões, ou outros dados, que chegam do exterior do *chip* e os dados enviados do *chip* para o exterior. Os dados que passam pelo PBLR podem ser as TAGs dos PEs, ou 32 bits duma posição qualquer duma linha da CAM.
A memória associativa pelo conteúdo, já referida como CAM e CAM-de-subset, tem um tamanho de 64*64 e 64*4 linhas por colunas, e executa essencialmente dois tipos de operações: (i) cada linha envia para o PE associado os sinais MD0, MD1 e MS (este é proveniente da CAM-de-subset), de acordo com a correspondência (*match*) entre os padrões enviados pelo PBL e o conteúdo da CAM e da CAM-de-subset; (ii) permite trocar as TAGs entre os PEs e o PBLR, efectuando uma rotação variável na posição dessas TAGs (mecanismo de *barrel*), usando para isso diagonais a 45^o que atravessam a CAM a intervalos de 4 células.
O vector de elementos de processamento é constituído por 64 PEs que formam uma cadeia que troca informação entre si nos 2 sentidos. Cada PE pode ser visto como uma ALU de 1-bit. Cada ALU contém 3 registo e duas unidades funcionais. Os registos são a TAG, o CARRY e o MS (subset), e as unidades são: (i) uma unidade que realiza operações lógicas, aritméticas, de *match*, entre outras; (ii) uma unidade responsável pela execução de 2 operações comuns em visão por computador, que podem ser definidas como a procura da 1ª TAG com valor "1"nos 64 PEs e a divisão ao meio e de forma alternada das TAGs com valor "1". A ALU permite ainda ler, ou escrever, informação do VSR para as TAGs.

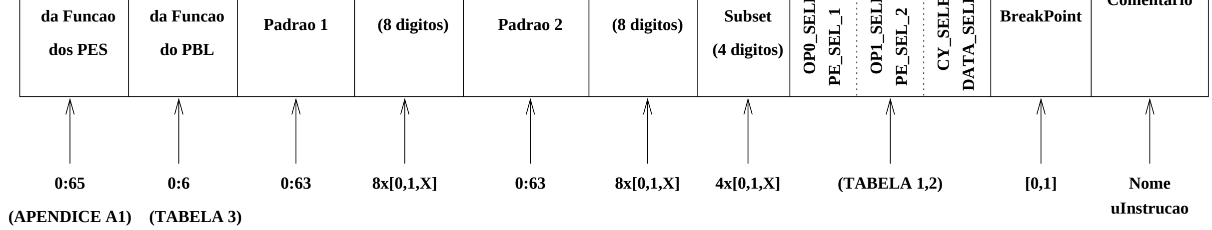

Figura 4: Formato das microinstruções do emulador do GLiTCH.

OP0_SELECT		OP1_SELECT		CY_SELECT	
código controlo	saída do bloco	código controlo	saída do bloco	código controlo	saída do bloco
1	MD0	1	MD1	1	CY_IN
2	DCARE	2	DCARE	2	DCARE
3	NOT_MD0	3	NOT_MD1	3	UM
		4	TAG_IN	4	ZERO
		5	NOT_TAG_IN		

Tabela 1: Codificação dos sinais de controlo dos blocos OP0_SELECT, OP1_SELECT e CY_SELECT dos PEs (colocada na variável *linhas* dos programas).

PE_SEL_1		PE_SEL_2		DATA_SELECT	
código controlo	entradas seleccionadas	código controlo	entradas seleccionadas	código controlo	entradas seleccionadas
0	ALL	0	ALL	0	GLOBAL
1	SUBSET	1	TAG	1	TAG
2	NOTSUB	2	CARRY	2	CARRY
		3	NOTTAG		
		4	NOTCY		

Tabela 2: Codificação dos sinais de controlo dos blocos PE_SEL_1, PE_SEL_2 e DATA_SELECT dos PEs (colocada na variável *linhas* dos programas).

2.2.2 Implementação em S/W de Componentes

Verifica-se que as partes do GLiTCH que mais interactuam são todas implementadas em s/w, por isso pode pensar-se nas componentes implementas em s/w quase como sendo o próprio GLiTCH. Esta constatação facilita a apresentação da forma de implementadas em s/w dos PEs, da CAM, do PBL e do descodificador de instruções.

Funcionando cada *chip* GLiTCH como um "caminho que faz operações sobre os dados", e dado que não existem neste processador operações que alterem o curso normal de execução das microinstruções, o modelo que representa o funcionamento deste processador é bastante simples (figura 3).

Apresentam-se a seguir os aspectos mais importantes da concretização deste modelo.

Em primeiro lugar, a explicação para a separação da execução das microinstruções em 3 etapas foi referida no ponto anterior, quando se explicou o funcionamento do descodificador de microinstruções.

Como se pode deduzir pela figura 3, a entrada no emulador é o microcódigo das operações a executar. Actualmente o microcódigo é lido dum ficheiro ("*.prg") que contém o valor do nº de microinstruções a executar e uma linha com microcódigo por cada uma das microinstruções.

O formato do microcódigo e a explicação de como se definem os campos que o constituem são incluídos na figura 4, que por sua vez recorre às tabelas 1, 2 e 3.

O bloco PBL é implementado codificando 7 funções, cada uma correspondendo a uma operação do PBL (Tabela 3).

Por seu lado, o bloco que representa o vector de PEs é implementado pela codificação de 66 funções correspondentes às 66 operações que os PEs executam e que são definidas na sua *ROM Table*. Cada função é manipulada através duma posição num *array* de apontadores para funções, e onde a posição equivale ao *opcode* dessa função.

Nº Usado no μcódigo	Função do PBL
0	pbl_nop()
1	pbl_match()
2	pbl_write()
3	pbl_read2()
4	pbl_read3()
5	pbl_mergeR()
6	pbl_mergeL()

Tabela 3: Codificação das funções do PBL usada nos programas.

A CAM é implementada por um *array* bidimensional e a funcionalidade é representada pelas funções que implementam o PBL e os PEs.
Depois de introduzidos os aspectos mais genéricos do modelo de implementação em s/w de componentes do GLiTCH, descrevem-se agora alguns pormenores.

(I) O 1º e 2º campos do microcódigo indicam qual a função a executar pelos PEs e pelo PBL, respectivamente.

(II) Se a função a executar pelo PBL for **match**, utilizam-se o padrões 1, 2 e Subset e as posições 1 e 2, incluídos no microcódigo, para gerar os sinais MD0, MD1 e MS. Se a função a executar for **write**, utilizam-se os padrões/posições referidos para controlar o que se escreve, e onde se escreve na CAM. As outras funções do PBL, **read** e **merge**, escrevem (lêem) dados no (do) PBLR a partir do (para o) exterior do GLiTCH. Nas funções **write** e **read** utiliza-se o campo **shared signals** do microcódigo, que representa os sinais de controlo PE_SEL_1 / PE_SEL_2 / DATA_SELECT, para definir o que se escreve ou lê da CAM (Tabela 2).

(III) Como as funções a executar pelos PEs são 66 e por isso não é viável falar sobre cada uma delas, refere-se apenas 4 aspectos: (i) as funções incluem um "case"para cada um dos valores possíveis para o campo *shared signals* que, excepto nas funções de escrita/leitura da CAM, representa os sinais de controlo OP0_SELECT / OP1_SELECT / CY_SELECT. Para determinar todos os valores possíveis para este campo, é necessário analisar todas as variantes de cada microinstrução; (ii) as funções colocam nos registos de cada PE os valores que se obtêm a partir da *ROM Table* (Apêndice A1) e da definição da ALU [RA91]; (iii) se for caso disso, as funções guardam nas *latches* ALT / FIRST / ENABLE os valores definidos pela *ROM Table*; (iv) as funções respeitam ainda os sinais/condições da mesma Tabela, não incluídos nos pontos anteriores. Por exemplo, a condição TAG_ROUTE define o que chega à TAG de cada PE: se a própria TAG, se a TAG do PE vizinho mais significativo, se a TAG do PE vizinho menos significativo ou um bit da coluna menos significativa do VSR.

(IV) O Apêndice A2 contém o código da implementação em s/w de componentes do GLiTCH.

Para consolidar o que foi referido sobre a implementação de componentes em s/w, ilustra-se com o caso das microinstruções que usam a função **arith_this** executada pelos PEs. Analisando a definição na *ROM Table* e consultando a definição da ALU, pode mencionar-se que:

(1) A partir do valor a guardar no registo de Subset (MS) e da condição **bufctl** conclui-se que o registo MS guarda o resultado vindo da CAM em consequência do *match* do padrão de subset, ou seja, fica a saber-se que a operação do PBL é **match**.

(2) O registo CY é igual à saída CY_OUT do bloco ALU_LOGIC da ALU, ou seja, $CY = ((OP1\hat{\ }OP0)\&CY\_IN)|(OP1\&OP0)$.

(3) O registo TAG é igual à saída SUM do bloco ALU_LOGIC, ou seja, $TAG = CY\_IN\hat{\ }(OP1\hat{\ }OP0)$.

(4) A condição TAG_ROUTE selecciona para a entrada TAG_IN dos blocos ALU_LOGIC, OP1_SELECT e TAG_REG a própria TAG dos PEs.

(5) As *latches* mantêm os seus valores.

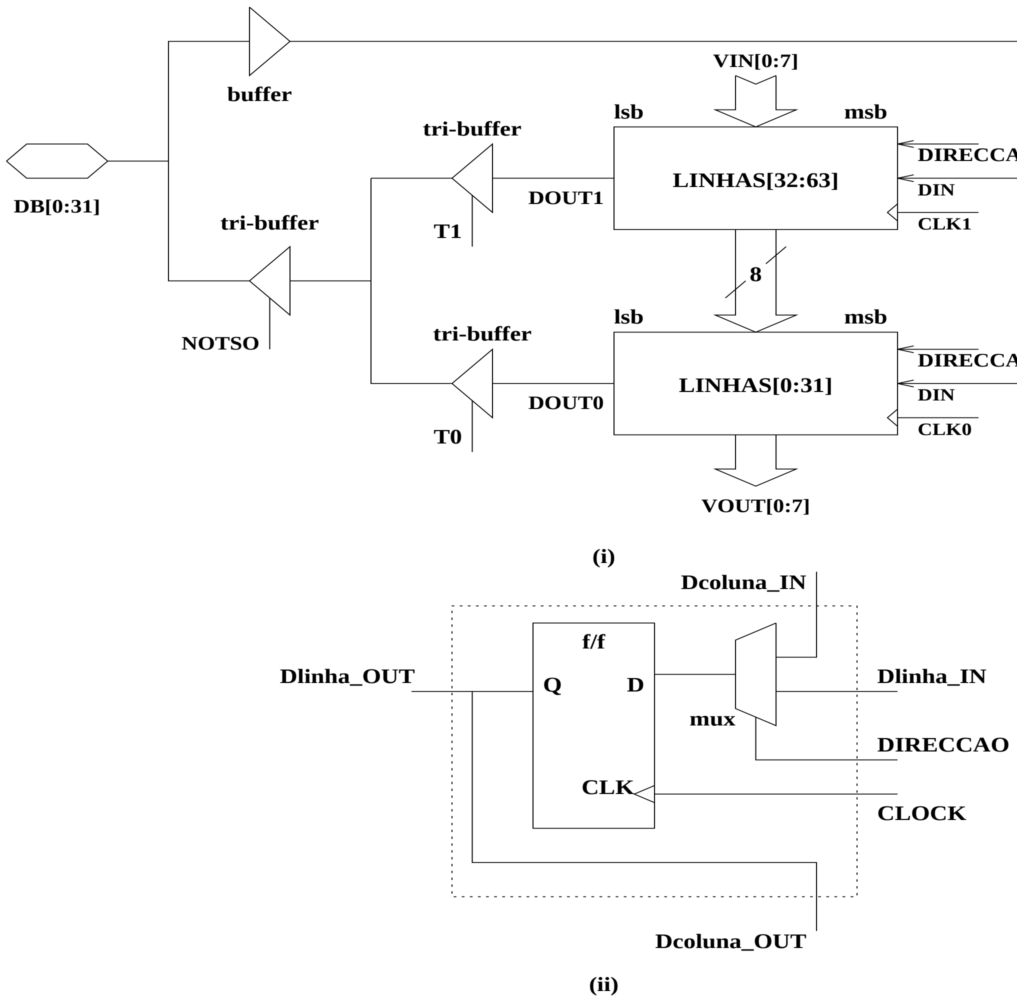

Figura 5: (i)Esquema de implementação do VSR em h/w; (ii)Célula de armazenamento dum bit. Legenda: $DIRECCAO = SHIFT$; $CLK0 = VSRCLK \cdot SHIFT + \overline{NOTADDR0} \cdot \overline{NOTWR} \cdot \overline{NOTLCACS} \cdot \overline{SHIFT}$; $CLK1 = VSRCLK \cdot SHIFT + NOTADDR0 \cdot \overline{NOTWR} \cdot \overline{NOTLCACS} \cdot \overline{SHIFT}$; $NOTSO = \overline{AB3 \cdot \overline{NOTRD} \cdot \overline{NOTLCACS} \cdot \overline{SHIFT}}$; $T0 = NOTADDR0$; $T1 = \overline{NOTADDR0}$.

Dos 5 pontos anteriores deduz-se que: (i) a tarefa mencionada em (1) é realizada pelo PBL; (ii) o ponto (5) não corresponde a nenhuma acção e o ponto (4) serve apenas para saber que valores são possíveis na entrada TAG_IN dos blocos citados; (iii) a função dos PEs é responsável pelas tarefas referidas nos pontos (2) e (3), para cada um dos valores possíveis no campo *shared signals*. O valor deste campo define os operandos OP0, OP1 e CY_IN utilizados para calcular TAG e CY. Se não se fizesse ideia dos valores necessários em *shared signals*, poderia implementar-se uma função que permitisse todos os valores possíveis, e que são: 4 (OP0=0, 1, MD0, NOT_MD0) *

6 (OP1=0, 1, MD1, NOT_MD1, TAG_IN, NOT_TAG_IN) * 3 (CY_IN=0, 1, CY_IN). No caso presente usaram-se 17 valores, pois os outros não são necessários.
A implementação desta função pode ser consultada no código do ficheiro **arithrom.c** do Apêndice A2.

2.3 Componentes do GLiTCH a Implementar em H/W

2.3.1 Descrição Funcional das Componentes a Implementar em H/W

De acordo com o processo de partição estabelecido, a partição de h/w corresponde ao registo de deslocamento para vídeo (VSR). O VSR é um registo organizado em matriz com dimensões 64*8 linhas por colunas, em que o nº de colunas corresponde ao nº de PEs do GLiTCH e o nº de linhas é imposto pelo tamanho dos dados (vídeo, ou outro tipo) que recebe ou envia para o exterior do *chip*.
A funcionalidade do VSR pode ser representada pela duas operações que efectua sobre os dados nele guardados: deslocamento de linhas (**shift**) e rotação de colunas (**swap**).
Na operação **shift** o VSR funciona independentemente dos outro blocos do processador e com a frequência dum relógio externo. Este facto traduz-se em que o VSR pode funcionar em paralelo com os restantes blocos, ou seja, enquanto os PEs operam sobre um conjunto de dados na CAM, o VSR pode enviar para o exterior um conjunto de dados já processado pelos PEs, ou então ler do exterior um novo conjunto de dados para os PEs processarem. A operação **shift** lê do exterior do *chip* (pelos pinos VIN[]) um byte de dados e regista-o na linha mais significativa, ao mesmo tempo desloca todas as linhas de uma posição e envia a linha menos significativa para o exterior (pelos pinos VOUT[]). Esta operação permite implementar a microinstrução **shiftVsr**.
Na operação **swap** lêem-se as TAGs dos 64 PEs para a coluna mais significativa do VSR, ao mesmo tempo que se rodam todas as colunas de uma posição e se envia a coluna menos significativa para os PEs. Esta coluna é colocada nas TAGs ou então é desprezada. Esta operação permite implementar as microinstruções **rotate_imag** e **extract_imag**, e é utilizada em todas as outras em que o bloco TAG_ROUTE da ALU selecciona para saída a entrada IMAGE (ROTATE_IN na definição da ALU).

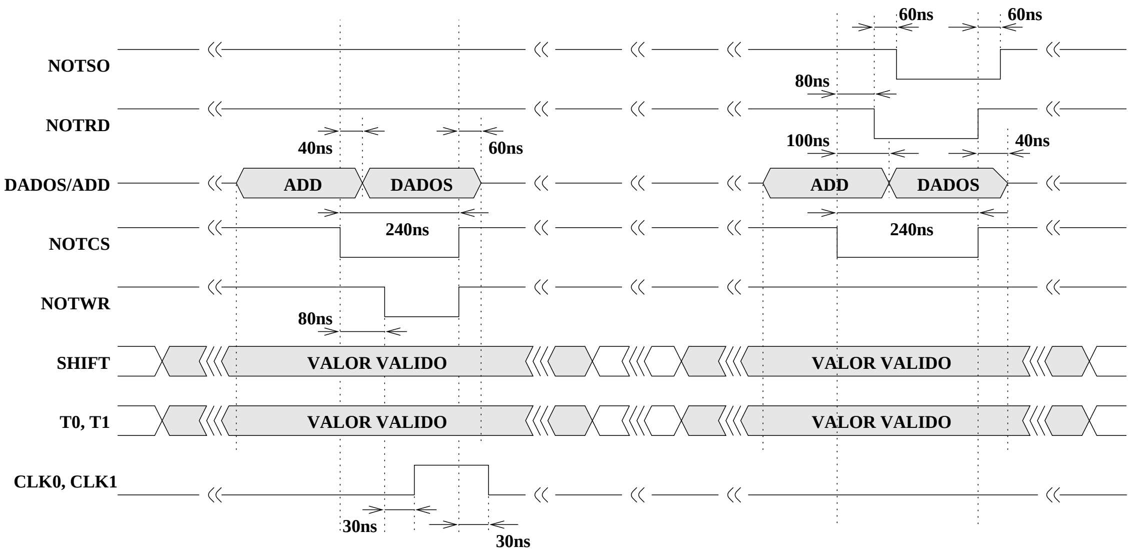

Figura 6: Temporização de sinais usados na implementação do VSR com uma FPGA do EDgAR.

2.3.2 Implementação em H/W de Componentes

O VSR foi sintetizado numa FPGA XC3090 de 175 pinos disponível no EDgAR. Devido a uma imposição desta plataforma, a ligação por colunas entre o VSR e o exterior é efectuada em duas

etapas, manipulando-se em cada etapa meia coluna. Este facto resulta de a ligação entre FPGAs e *transputer* ser de 32 bits e não 64 bits como é a largura das colunas.
A figura 5(i) ilustra, de forma sintética, como se implementou o VSR na FPGA, enquanto que na parte (ii) desta figura se mostra a célula que armazena um bit em cada linha ou coluna. Esta célula permite circulação de informação horizontalmente (por linhas) e verticalmente (por colunas), de acordo com o sinal DIRECCAO.
Os esquemáticos completos relativos à implementação do VSR encontram-se no Apêndice A4. O esquemático topo da hierarquia é **P2_V2.1** e os restantes são **VSR.1**, **LINHA.1** e **FFD_CE.1**.
Quando a complexidade dos sistemas implementados em FPGA atinge valores para os quais o encaminhamento dos sinais se torna crítico, tem todo o interesse conhecer a temporização do sinais envolvidos, por forma a verificar o correcto funcionamento desses sistemas. É este o caso da implementação do VSR, uma vez que a relação entre lógica alocada e lógica existente é elevada. Esta é a principal razão porque se incluiu a figura 6 com a temporização dos sinais envolvidos na implementação do VSR.
Foi com base na análise temporal que se alterou a lógica que gera os sinais de controlo, resultando num menor desempenho do sistema mas funcionando adequadamente. Por exemplo, os sinais de controlo SHIFT, T0 e T1 passaram a ser provenientes de F/Fs actualizados por uma operação de escrita anterior à ordem de operação do VSR. Com esta alternativa eliminaram-se as complicações dos atrasos nestes sinais, restando como fonte de possíveis complicações nas temporizações da figura 6, os sinais NOTSO (que depende de NOTRD) e CLK0/CLK1 (que dependem de NOTWR). Pela observação destes sinais e dos esquemáticos, vê-se que o único problema resultante dos seus atrasos é a degradação do desempenho.
Outra alteração que resultou da análise temporal, foi a eliminação de sinais de habilitação dos F/Fs e a introdução de lógica equivalente nos sinais de relógio; esta alteração, embora contrária à regra de não utilização de lógica nos sinais de relógio, revelou-se adequada uma vez que não existem picos casuais e evita que as transições do sinal de relógio cheguem aos F/Fs antes dos sinais de habilitação. Esta relação temporal inadequada entre sinal de relógio e sinal de habilitação é difícil de evitar porque as FPGAs possuem dois caminhos para sinais de relógio que introduzem atrasos inferiores aos outros caminhos da FPGA, devendo estes caminhos ser usados para evitar os problemas de *fan-out*.

2.4 Interface entre Partições de H/W e de S/W

A interface entre a partição de s/w e de h/w deve implementar todas as ligações entre as 2 partições por forma a que o conjunto emule o GLiTCH. As ligações que é necessário estabelecer são: (i) a ligação das colunas mais e menos significativa do VSR com os PEs, (ii) a ligação das linhas mais e menos significativa do VSR com o exterior do *chip* e (iii) a ligação ao VSR, a partir dos PEs e do exterior, dos sinais de controlo.
Para implementar as ligações enumeradas podem utilizar-se dois tipos de recursos da plataforma EDgAR: os barramentos de dados e de endereços e os conectores que permitem interligar os componentes de lógica programável (MACHs e FPGAs).
A ligação definida no ponto (i) foi implementada através do barramento de 32 bits que liga a FPGA do VSR ao *transputer*. Não é necessário adicionar lógica para conseguir esta parte da interface porque o VSR já está organizado em duas metades de 32 bits (figura 7).
Na versão actual do emulador não se implementam ligações para o exterior do *chip*, resultando assim que a ligação (ii) entre as partições de s/w e de h/w é feita do VSR para o *transputer*, que por sua vez simula as ligações para o exterior. Para implementar esta ligação utiliza-se uma 2ª FPGA conjuntamente com conectores que a ligam à FPGA do VSR, como se ilustra na figura 7. Estas ligações podiam ser incluídas na FPGA do VSR, mas como isso implicava utilizar mais lógica duma FPGA já com uma taxa de alocação elevada, não se seguiu essa opção pela dificuldade que apresentava a sua concretização.
A ligação definida em (iii) foi parcialmente implementada com o mesmo método da ligação (ii)

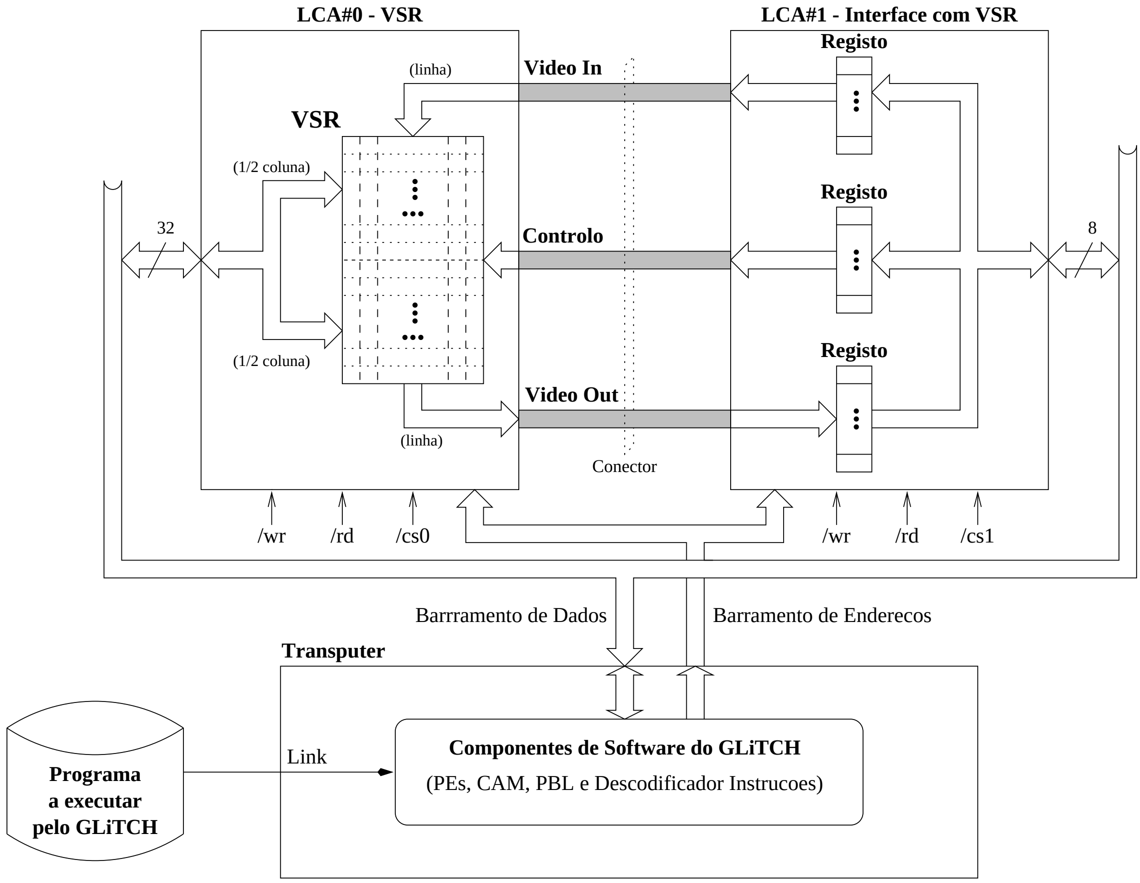

Figura 7: Implementação do GLiTCH com componentes de *hardware* e de *software* da plataforma EDgAR.

e a outra parcela utiliza os sinais de controlo da própria FPGA do VSR (NOTWR, NOTRD, NOTLCACS).
A implementação da parte da interface alocada à 2a FPGA encontra-se no Apêndice A4 (ficheiro **INTERFAC.1**) e a lista de ligações físicas entre duas FPGAs está no Apêndice A5.
A figura 7 dá uma perspectiva global da implementação do GLiTCH através da plataforma EDgAR, recorrendo a componentes de h/w e de s/w.

3 Linguagem *Assembly* para o GLiTCH

Como foi dito no tópico sobre a implementação de componentes em s/w, a entrada que acciona o emulador é o microcódigo das operações a executar. Dado que o microcódigo constitui uma péssima interface para o utilizador, escreveu-se uma linguagem **assembly** para o GLiTCH. Esta linguagem foi adaptada duma versão usada num simulador do GLiTCH [AR89], podendo consultar-se o manual da linguagem definida no Apêndice A3.
A sintaxe da linguagem é muito simples, pois é constituída apenas por funções com parâmetros. A sintaxe destas funções é igual à das funções na linguagem C. A simplicidade resulta do facto de, como já foi mencionado, o GLiTCH funcionar como um caminho de dados.
Cada uma das funções que compõem a linguagem está completa, no sentido que se permite que os parâmetros assumam todas as combinações possíveis para esse caso.
A linguagem definida não é a única possível para o mesmo microcódigo, dada a complexidade do caminho de dados que constitui o GLiTCH.
O manual do Apêndice A3 inclui a explicação de cada função, os valores possíveis para os parâmetros, os valores que resultam nos 3 registos dos PEs e o microcódigo correspondente a

Tipo de Lógica	**Percentagem (%)**
Células Base (CLBs)	87
Células de I/0 (IOBs)	40
Flip-flops	80
Buffers Tri-state	9
Linha Longas com Tri-state	80

Tabela 4: Lógica alocada na FPGA do VSR.

Tipo de Atraso	**Valor (ns)**	**Freq. Máxima (MHz)**
Pior Atraso Acumulado	73	13
Pior Atraso entre F/Fs	25	40
Pior Atraso de F/Fs para PADs	31	32
Pior Atraso de PADs para F/Fs	36	27

Tabela 5: Atrasos mais significativos na FPGA do VSR.

essas funções.

4 Resultados

Verificou-se o correcto funcionamento da partição de h/w e das partes da partição de s/w que não utilizam a interface com outros processadores GLiTCH. As partes da partição de s/w que operam com vários *chips* não puderam ser totalmente testadas, uma vez que o trabalho realizado só emula o *chip* GLiTCH e não o seu exterior.
As tabelas 4 e 5 contêm resultados quantitativos sobre a partição de h/w.
Relativamente ao tempo de execução de microinstruções, deu-se ênfase apenas às microinstruções que utilizam a partição de h/w. Como o tempo mínimo que se mede é $64\mu s$, efectuaram-se medidas para a execução de 100 microinstruções. Na tabela 6 compara-se o tempo de execução das microinstruções em duas situações: uma em que o VSR é implementado como foi descrito até aqui (em h/w) e outra em que o VSR foi simulado em s/w (ficheiro **vsr-sw.c** do Apêndice A2). Nesta tabela não se contabilizou o tempo de busca das microinstruções.

Instrução	**Implementação em S/W**	**Implementação em H/W**
rotate_imag	240 ns	98 ns
shiftVsr	420 ns	80 ns

Tabela 6: Tempo de execução de microinstruções do GLiTCH. (Nota: como **extract_imag** é igual a **rotate_imag** na parte respeitante ao h/w, não foi incluída)

5 Conclusões e Trabalho Futuro

Entre os melhoramentos mais significativos, ou partes em falta no emulador contam-se:

(i) Falta escrever um "compilador" da linguagem *assembly* para microcódigo, por forma a usufruir totalmente das vantagens desta linguagem. É conveniente testar o emulador com programas mais elaborados do que os usados e adequar as funções que operam com vários *chips*. A lista destas funções é a seguinte: merge_tags(), bounce_tags(), long_shift(), alt(), cy_alt(), tag_alt() e first(). Para que a função shiftVsr() funcione de acordo com a especificação do GLiTCH, é preciso alterar a interface entre o GLiTCH e o exterior, passando duma interface de s/w para outra de h/w (ver o próximo ponto).

(ii) Uma alteração a introduzir no emulador, para que este possa ser testado com a arquitectura de visão referida na figura 1, consistirá em colocar numa FPGA registos para todos os sinais que correspondem a pinos do GLiTCH e ligações destes registos para os conectores da plataforma EDgAR.

(iii) Um estratégia alternativa para testar o emulador com a arquitectura completa, a qual inclua vários GLiTCHs, será a emulação dessa arquitectura. Embora o h/w imponha limitações ao nº de VSRs a emular, o s/w permite grande liberdade neste aspecto, dada a possibilidade de ligar o *transputer* da plataforma a um sistema com dezenas de *transputers*. O modelo do sistema *multi-chip* não é complicado porque se pode utilizar o protocolo assíncrono dos canais virtuais e físicos do *transputer*. Os canais podem assim implementar a ligação entre dois processos que correspondem à emulação de dois *chips*.

(iv) Falta explorar o paralelismo ao nível da partição de s/w, o que traz problemas de sincronização à execução das microinstruções.

(v) Para baixar o tempo de busca das microinstruções é conveniente introduzir algumas alterações no código do emulador: eliminar, ou reduzir, a utilização de funções do C e principalmente, colocar o microcódigo em memória em vez de num ficheiro.

Uma das conclusões a extrair do trabalho realizado tem a ver com o ganho obtido no tempo de execução das microinstruções quando se utiliza h/w, o qual ficou abaixo do esperado. Uma das explicações para este facto está na arquitectura do EDgAR, que ao não permitir controlo das FPGAs sobre a memória, obriga a que seja sempre sob ordem do *transputer* que as acções de leitura ou escrita se processam. Outra razão tem a ver com a estratégia empregue na interface h/w-s/w, pois optou-se por uma solução fiável e simples, mas que degrada o desempenho por obrigar o *transputer* a uma operação de I/O adicional. Na microinstrução **rotate_imag** o ganho ainda foi inferior do que em **shiftVsr**, pois esta microinstrução é executada em duas partes iguais, ou seja, o ganho caiu para metade.
Demonstrou-se a validade da metodologia de co-projecto ao obter-se ganhos significativos no desempenho do emulador, enquanto que o custo adicional da implementação, resultante do maior tempo de projecto, é inferior.
Verifica-se ainda que a programação do GLiTCH não é simples, uma vez que a arquitectura é bastante diferente da dos processadores mais familiares. Uma alternativa para contornar este inconveniente será escrever **macros** que realizem as tarefas mais comuns na área de aplicação. Por exemplo, em visão por computador teria interesse escrever macros para calcular medianas, convoluções ou produtos.

Referências

[Alv92] Manuel J. D. Alves. Emulação dum Processador Associativo Vectorial - GLiTCH, December 1992. Relatório de Estágio da Lic. em Eng. de Sistemas e Informática.

[AR89] A.W.G.Duller and R.H.Storer. GLiTCH Simulation, Version 2.1. Technical report, Dept. of Electrical and Electronic Eng., Bristol University, UK, October 1989.

[DST+89] A. W. G. Duller, R. H. Storer, A. R. Thomson, E. L. Dagless, M. R. Pout, and A. P. Marriot. Design of an Associative Processor Array, 1989.

[EFSP95] António Esteves, João M. Fernandes, Henrique Santos, and Alberto Proença . *A plataforma EDgAR na Prototipagem Rápida de Sistemas Digitais.* In *Proceedings of 2º Encontro Nacional do Colégio de Engenharia Electrotécnica da Ordem dos Engenheiros*, pages 83–89. Ordem dos Engenheiros, December 1995. IST, Lisboa, Portugal.

[Est94] António J. Esteves. Prototipagem Rápida de Sistemas Digitais - Parte I. Technical report, Departamento de Informática Universidade do Minho, July 1994.

[RA91] R.H.Storer and A.R.Thomson. ALU Function Definition, Version 2.1. Technical report, Dept. of Electrical and Electronic Eng., Bristol University, UK, March 1991.