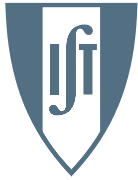
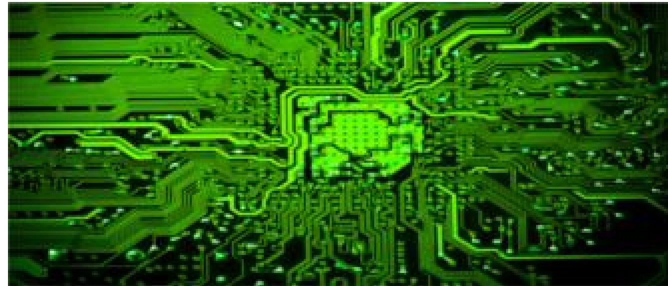


INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa



Suporte hardware para um debugger para o Processador pedagógico P3

Juan José Rebollo Barranco

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Professor José Manuel da Costa Alves Marques
Orientador:	Professor José Carlos Alves Pereira Monteiros
	Professor Paulo Ferreira Godinho Flores
Vogais:	Professor Nuno Cavaco Gomes Horta

Outubro 2012

“Podemos cometer muitos erros nas nossas vidas, exceto um: o que nos destrói” — Paulo Coelho

“O homem que cometeu um erro e não corrigi-lo, está cometendo um outro erro mais” — Confúcio

“Se a depuração é o processo de remoção de bugs, então a programação deve ser o processo de introduzir-lhos.” — Edsger Dijkstra

Agradecimentos

Gostaria de agradecer aos professores José Monteiro e Paulo Flores pela sua ajuda, apoio e motivação. Num ano que foi difícil pela carga académica e num país estrangeiro eles sempre me deram soluções aos meus problemas e é por eles numa grande parte que consegui terminar a minha Master Thesis.

Também agradecer aos meus companheiros de aulas e professores em Huelva, Sevilha e Lisboa. Foram um fator decisivo na consecução deste mestrado.

Aos meus amigos em Jerez, Huelva, Sevilha e Lisboa, porque foram numa grande parte a minha motivação em todos estos anos. Sempre confiaram em mim e não podia falhar-lhes.

À minha família, porque também foram sempre um apoio desde a distância e nunca deixaram de dar-me uma palavra de animo quando precisei.

A minha namorada, a Laura, porque sem ela todos estos anos não tenho certeza o que é que houver acontecido comigo. Ela foi sempre a luz do farol que guiou-me.

Uma palavra especial aos meus pais. Obrigado à minha mãe, Ana, pela sua ternura. Obrigado ao meu pai, José, pelo seu sacrifício. Eles fizeram-me o homem que hoje sou.

Resumo

Alguns dos professores do IST criaram um processador pedagógico chamado P3. Com ele, os alunos da disciplina *Arquitectura de Computadores* podem desenvolver melhor os seus conhecimentos em relação a esta área.

Para o aproveitamento disto por parte dos alunos foi criado um software que simula o suporte hardware no qual é simulado o P3 e as suas interfaces. Neste simulador podemos fazer debug dos programas assembly que queremos programar no processador mas isto tem uma grande desvantagem: a velocidade com programas grandes ou complexos.

Posteriormente, um aluno fez na sua tese o desenvolvimento do processador em VHDL assim como uma interface para que o P3 possa interagir com outros dispositivos de entrada/saída do suporte hardware onde vai ser programado numa FPGA e assim poder ver o processador a correr de maneira real.

O objectivo da presente tese é desenvolver um debugger hardware para o processador P3 de maneira que se possa fazer o debug de um programa que esteja a correr no P3 de maneira real através do prompt do programa de carregamento e não num simulador como até o momento.

O resultado foi testado em programas reais — principalmente no projecto da disciplina *Arquitectura de Computadores* do ano 2011/2012 — e, em linhas gerais, foi satisfatório ainda que a frequência tenha sido reduzida devido à nova lógica inserida.

Palavras Chave

FPGA, P3, Debugger, VHDL, Xilinx, Trap Flag.

Abstract

Some of the IST′s teachers created a pedagogical processor called P3. With it, students which were registered in the *Computers Arquitecture* subject can develop knowledge related with this scope better.

Later, a student made in his thesis the development of the whole processor in VHDL as well as an interface for P3 could interact with other hardware input/output devices which will be programmed in an FPGA so we can see the processor operate real.

To take advantage of this for the students was created software that simulates the hardware support which is programmed the P3 and its interfaces. In this simulator we can debug assembly programs we want to program into the processor but it has one major disadvantage: the speed with large or complex programs.

The aim of this thesis is to develop a hardware debugger for the P3 processor so you can do debug a program that is running on P3 by the loading program′s prompt and not in a simulator like until now.

The result was tested on real programs — especially in the *Computer Architecture* discipline project of the year 2011/2012 — and, in general, satisfactory although the frequency was seen compromised due to new electronic devices inserted.

Keywords

FPGA, P3, Debugger, VHDL, Xilinx, Trap Flag.

Conteúdo

1 Introdução **1**

1.1 Motivação 2

1.2 Objectivos 3

1.3 Principais Contribuições 3

1.4 Organização da Dissertação 4

2 Estado da Arte **5**

2.1 GDB 6

2.2 OllyDbg 6

2.3 SoftICE 8

2.4 MacsBug 9

2.5 Resumo das características dos Debuggers analizados 10

2.6 Características gerais do *Debugger* 10

3 Hardware e Ferramentas de Desenvolvimento **13**

3.1 O Processador P3 14

3.1.1 História e motivações 14

3.1.2 Arquitectura 14

3.1.2.A Registos. 14

3.1.2.B Bits de Estado. 15

3.1.2.C Memória. 15

3.1.2.D Entradas/Saídas. 15

3.1.2.E Interrupções. 16

3.1.2.F Microcódigo. 16

3.2 Hardware 17

3.2.1 Digilent D2-SB System Board 17

3.2.2 Digilent DIO5 Peripheral Board 17

3.2.3 Digilent PIO1 Parallel I/O Board 17

3.2.4 Digilent MEM1 Memory Module 1 18

3.3 Arquitectura original do sistema 18

3.4 Ferramentas de Desenvolvimento 19
3.4.1 Xilinx ISE 10.1 20
3.4.2 ModelSim PE Student Edition 10.1c 21

4 Arquitectura 25
4.1 Funcionalidades a desenvolver 26
4.1.1 O comando Run 26
4.1.2 O comando Cont 26
4.1.3 O comando Step 26
4.1.4 O comando List 27
4.1.5 O comando Code 27
4.1.6 O comando Br 27
4.1.7 O comando Del 28
4.1.8 O comando Print 28
4.1.9 O comando Write 28
4.1.10 O comando Exit 29
4.2 O *Debugger* 29
4.2.1 Ativar e desativar o modo *Debugger* 29
4.2.2 Armazenamento dos Breakpoints 31
4.2.3 Decodificação dos Comandos 31
4.3 Modificações ao D2-SB 35
4.4 Modificações ao Processador P3 37

5 Implementação e Resultados 41
5.1 Metodologia de Desenvolvimento 42
5.2 Resultados 42
5.3 Observações 46

6 Conclusões 47

Lista de Figuras

1.1 *Screenshot* do *p3sim* 2

2.1 GDB *Screenshot*. 7
2.2 Ajuda do GDB *Screenshot*. 8
2.3 OllyDbg *Screenshot*. 9
2.4 SoftICE *Screenshot*. 10
2.5 MacsBug *Screenshot*. 11

3.1 Placa *D2-SB*. Peça central, contém a FPGA a programar. 17
3.2 Placa *DIO5*. Contém os dispositivos de entrada e saída. 18
3.3 Placa *PIO1*. Fornece um meio de comunicação através de uma porta paralela. . . 18
3.4 Placa *MEM1*. Encarregada de fornecer de uma memória flash ao conjunto. 19
3.5 Arquitectura original do sistema. 20
3.6 Microsoft Visual C++ 2008 Express Edition *Screenshot*. 21
3.7 Xilinx ISE 10.1 *Screenshot*. 22
3.8 ModelSim PE Student Edition 10.1c *Screenshot*. 23

4.1 Hardware do ligado e desligado do *Debugger*. 30
4.2 Hardware do armazenamento dos *Breakpoints* do *Debugger*. 32
4.3 Hardware do armazenamento e decodificação dos comandos do *Debugger*. . . . 33
4.4 Arquitectura modificada do sistema. 36
4.5 Arquitectura modificada do P3. 37
4.6 Arquitectura modificada do *Data Unit*. 38

5.1 Projecto AC 2011-2012, *Pacman*. 43
5.2 P3Loader - Debugger: Operações disponíveis. 43
5.3 P3Loader - Debugger: Funcionamento dos comandos *Print* e *Write*. 44
5.4 P3Loader - Debugger: Funcionamento dos comandos *Br* e *Step*. 45

Lista de Tabelas

2.1 Resumo/Comparativa das características dos debuggers analizados. 11

5.1 Comparativa do número de linhas inicial e final. 44

5.2 Comparativa do número de *Slices* ocupados inicial e final. 44

5.3 Comparativa da frequência máxima inicial e final. 45

Lista de Acrónimos

API *Application Programming Interface*

CAR *Control Address Register*

CISC *Complex Instruction Set Computer*

CPLD *Complex Programmable Logic Device*

EPP *Enhanced Parallel Port*

FPGA *Field Programmable Gate Array*

GDB *GNU DeBugger*

GNU *GNU is Not Unix*

GPL *General Public License*

IST *Instituto Superior Técnico*

LCD *Liquid Crystal Display*

LED *Light-Emitting Diode*

P3 *PPP - Pequeno Processador Pedagógico*

PROM *Programable Read only Memory*

PS/2 *Personal System/2*

ROM *Read only Memory*

RISC *Reduced Instruction Set Computer*

SPP *Standar Parallel Port*

SRAM *Static Random Access Memory*

USB *Universal Serial Bus*

VGA *Video Graphics Array*

VHDL *VHSIC Hardware Description Language*

VHSIC *Very High Speed Integrated Circuits*

1

Introdução

Conteúdo

1.1 Motivação 2
1.2 Objectivos 3
1.3 Principais Contribuições 3
1.4 Organização da Dissertação 4

O P3 é um microprocessador CISC pedagógico de 16 bits feito no IST que tem como objectivo o ensino da estrutura interna de um processador na disciplina *Arquitectura de Computadores* sem as desvantagens que podem ter o ensino desta matéria com processadores comerciais.

No 2011 um aluno do IST na sua tese [1] fez o desenvolvimento do processador em VHDL assim como umas interfaces para poder programar o processador numa FPGA e poder interagir com ele através dos dispositivos de entrada e saída que tem a placa onde se programa.

1.1 Motivação

Com o objectivo de que o P3 possa ser estudado por todos os alunos desta disciplina, programou-se um simulador — *p3sim*, Figura 1.1 — no que pode-se interagir com uma interface que consegue simular a placa na que é programado o P3.

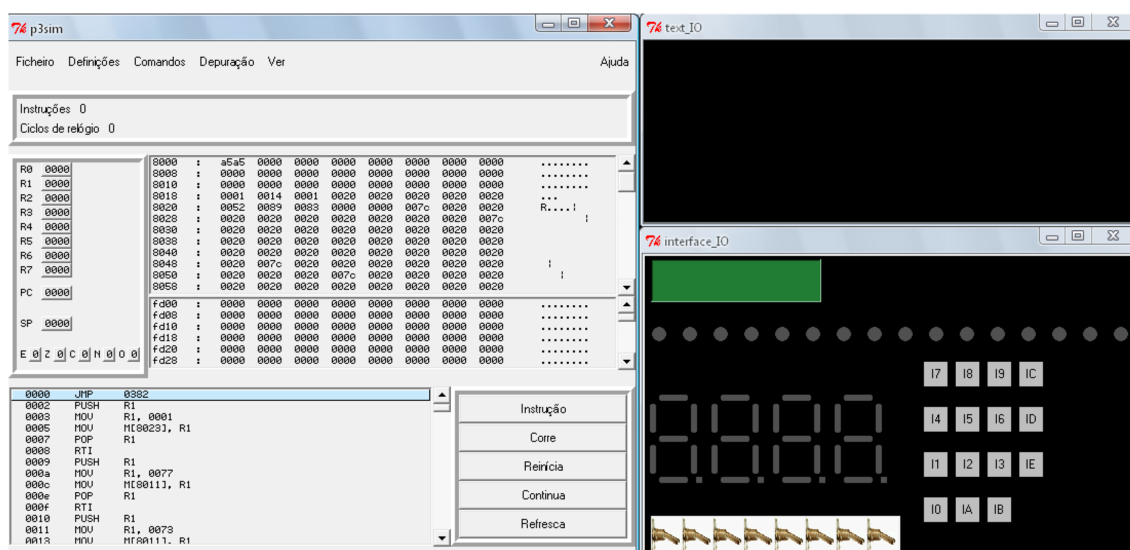


Figura 1.1: *Screenshot* do *p3sim*

Neste simulador, assim como no P3, o programador vai escrever o programa em código assembly com o alvo de que o programa *p3as* — o assembler — traduzir os mnemônicos a código máquina para que o processador possa executar as instruções — no simulador ou na placa.

Durante as sessões laboratoriais este simulador permite a execução e o debug de código assembly mas esta solução só é mais ou menos viável quando o programa assembly não é demasiado grande. Se assim for, o simulador trabalha devagar pelo que esta solução torna-se pouco produtiva e, em algumas ocasiões, inviável.

Outra solução pode ser carregar o código máquina do programa assembly no P3 — uma vez o P3 esteja programado na placa — diretamente mas de esta maneira não é possível fazer o debug ao programa carregado pelo que não vamos poder inserir *breakpoints*, fazer a execução *step-by-step*, etc, pelo que isto pode fazer o nosso trabalho muito mais difícil corrigir problemas.

Como última opção o programador pode fazer uma depuração na placa através dos dispositivos de entrada/saída do próprio hardware mas temos uma muito grande desvantagem para o programador: deve aprender a linguagem de desenho hardware VHDL para poder usar estes dispositivos de alguma maneira diferente à que já tem fixada — além de fazer um estudo pormenorizado do processador P3 para saber quais são as linhas que devem ser ligadas com os dispositivos entrada/saída da placa. Como podemos deduzir, esta maneira de programar tem muita dificuldade para o programador pelo que devemos arranjar alguma outra maneira de depurar o código assembly de uma forma mais eficiente e produtiva.

1.2 Objectivos

Os debuggers têm um papel muito importante na programação devido principalmente a que é por eles que os programadores podem parar a execução dos programas, pôr *breakpoints*, ver o conteúdo das variáveis, poder executar só uma instrução para ver o que é que muda, etc. É por tudo isto que o programador pode limpar de erros o programa para o seu correto funcionamento.

É por tudo o apresentado anteriormente que o objectivo deste trabalho é realizar a implementação em FPGA de um debugger com o que vamos poder controlar a execução das instruções no processador e que vai fazer que possamos realizar o debug de programas assembly grandes sem os problemas de velocidade que temos com o simulador.

Para isto, além do desenho do debugger em VHDL, vai ser preciso alterar um programa que foi feito com o objectivo de carregar o código assembly no P3 ou dados nas memórias que o processador tem. Este programa chama-se *P3Loader* e vai ser alterado para ter a opção de entrar no modo debug no qual, através do prompt, vamos poder inserir breakpoints, parar o P3, ver e mudar os seus registos, ver e mudar o conteúdo da sua memória, etc.

1.3 Principais Contribuições

Agora que o trabalho fica terminado, todos os alunos vão poder fazer debug a programas assembly complexos sem ter as desvantagens que o simulador tinha e poder desenvolver uma melhor e maior capacidade no âmbito da arquitectura de computadores.

Além disso, os professores da disciplina vão ter maior facilidade quando tiveram que fornecer de exemplos de programas aos seus alunos para estes desenvolver as capacidades que *Arquitectura de Computadores* requer.

Em geral, o ensino do IST neste âmbito vai ficar muito melhor e mais eficiente que anteriormente, sendo os seus próprios alunos os que colaboraram com as suas teses na melhoria do seu ensino.

1.4 Organização da Dissertação

Nesta dissertação vamos ver como é que são desenvolvidas as seguintes seções:

Capítulo 2 É realizada uma revisão do estado da arte para pôr em situação dos debuggers que há no mercado atualmente.

Capítulo 3 Vemos uma série de elementos que precisamos para o desenvolvimento da tese — o processador P3, o hardware tecnológico usado, a versão inicial do desenho hardware do P3 e as ferramentas software.

Capítulo 4 Neste capítulo vamos ver a arquitectura — a nova e as modificações das antigas — desenvolvida para o nosso alvo.

Capítulo 5 Aqui podemos ver a metodologia usada, os resultados e algumas observações.

Capítulo 6 Por último, neste capítulo, vamos ver as conclusões da tese.

Para uma melhor legibilidade é preciso dizer que nas figuras é usada a cor verde para os sinais de entrada, a vermelha para os sinais de saídas, azul para sinais bidirecionais e laranja para sinais ou elementos novos — inseridos para a realização do nosso objectivo.

2

Estado da Arte

Conteúdo

2.1 GDB **6**
2.2 OllyDbg **6**
2.3 SoftICE **8**
2.4 MacsBug **9**
2.5 Resumo das características dos Debuggers analizados **10**
2.6 Características gerais do *Debugger* **10**

Para a implementação do debugger, também vamos aprender como é que outros debbugers funcionam. Vamos tentar fazer um debbuger que possa oferecer algumas das vantagens dos principais debuggers do mercado.

Aqui seguido vamos ver alguns mas antes devemos dizer que ao final de cada debugger vamos poder ler umas linhas vendo as semelhanças e diferenças dos debuggers comerciais com o projecto que estamos a desenvolver.

2.1 GDB

O GNU Project debbuger permite-nos ver o que é que acontece no interior do programa que estamos a executar ou ver que é que está a fazer no momento que faz a paragem.

O GDB [2] pode realizar principalmente quatro operações:

- Começar executar o programa com alguma mudança específica nas variáveis que altere o comportamento do programa.
- Fazer que o programa pare quando especifiquemos que tem que parar através da inserção de breakpoints.
- Rever o que é que aconteceu quando o programa foi parado.
- Mudar registos e posições de memória no programa. Desta maneira podemos experimentar corrigindo os efeitos de um erro ou aprender mais sobre o comportamento do programa.

O programa que está a ser depurado pode ser escrito em *Ada*, *C*, *C++*, *Objective-C*, *Pascal*, etc. Estos programas podem ser executados de maneira nativa ou remota. A nova versão do GDB é 7.3.1 e podemos baixar de maneira gratuita. Na Figura 2.1 podemos ver um *screenshot* do debugger.

O GDB é software livre, protegido pela GNU GPL. Esta licença dá-nos a liberdade de copiar ou adaptar o programa licenciado. Cada pessoa que conseguir uma cópia também fica com a liberdade para modificar essa cópia e para distribuir-lha.

A nossa solução vai ser muito parecida do GDB devido a que vai ter as quatro funcionalidades descritas mas só vamos poder executar o código assembly do P3. Além disso, também vai ter uma ajuda como vemos que o GDB tem na Figura 2.2 e vai ser totalmente livre para os alunos que estejam interessados.

2.2 OllyDbg

O OllyDbg [3] é um debugger de código assembler de 32 bits para sistemas operativos *Microsoft Windows*. Faz especial ênfase na análise do código binário pelo que é muito útil quando não

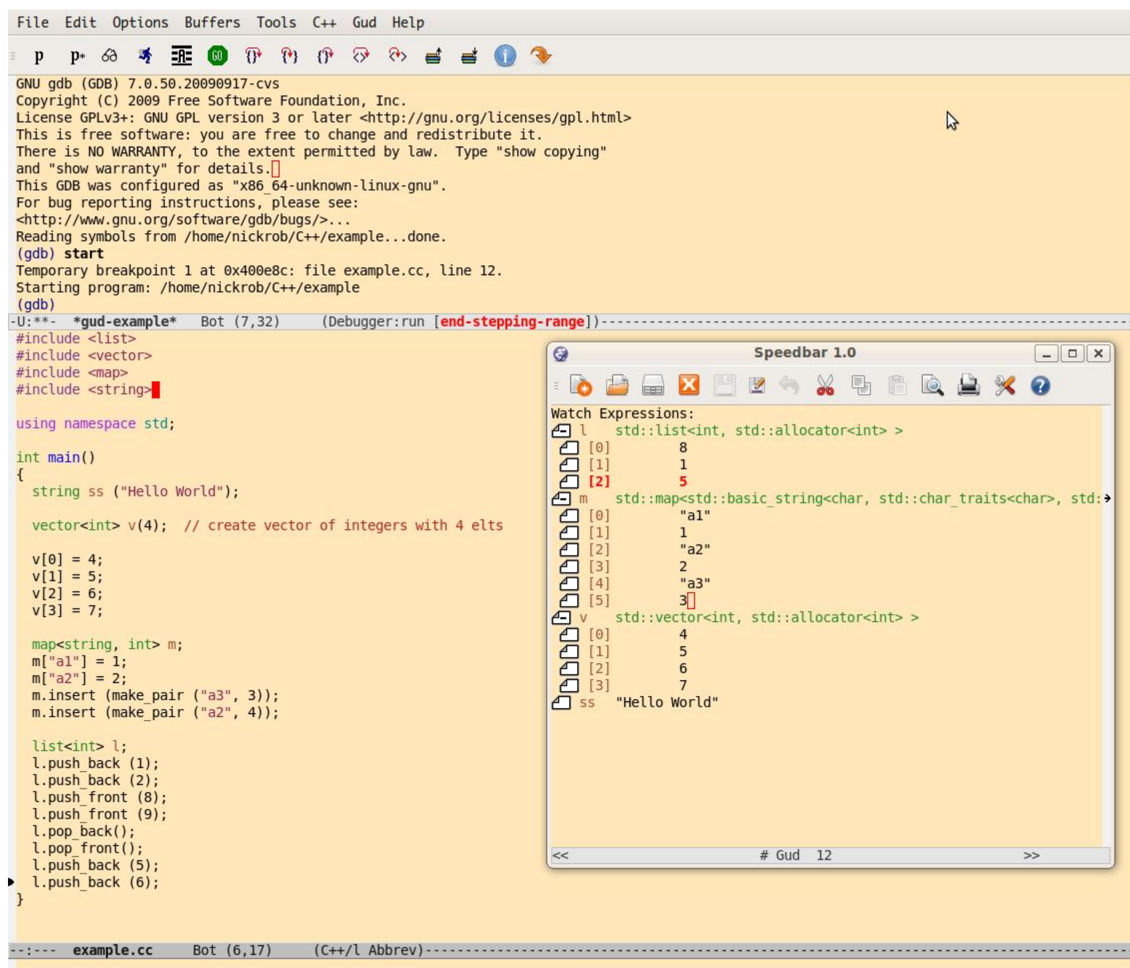


Figura 2.1: GDB *Screenshot*.

está disponível o código fonte do programa. Pode ver e modificar registos, reconhece procedimentos, chamadas ás API, estruturas de bucles e switches, tabelas, constantes e strings, assim como localiza rotinas de arquivos objecto e de bibliotecas. Não precisa de instalação.

Actualmente está em desenvolvimento a versão 2.0. É um software sem custos mas a licença shareware requer que os utilizadores registem-se com o autor.

Além disto, dizer que é frequentemente usado para fazer engenharia reversa.

O debugger que estamos a desenvolver não vai fazer ênfase no código binário — ainda que vai ser de muito baixo nível — e não só vai trabalhar sob *Microsoft Windows* mas, como no OllyDbg, vamos poder alterar registos e vamos fazer foco no código assembly e não vai precisar de instalação.

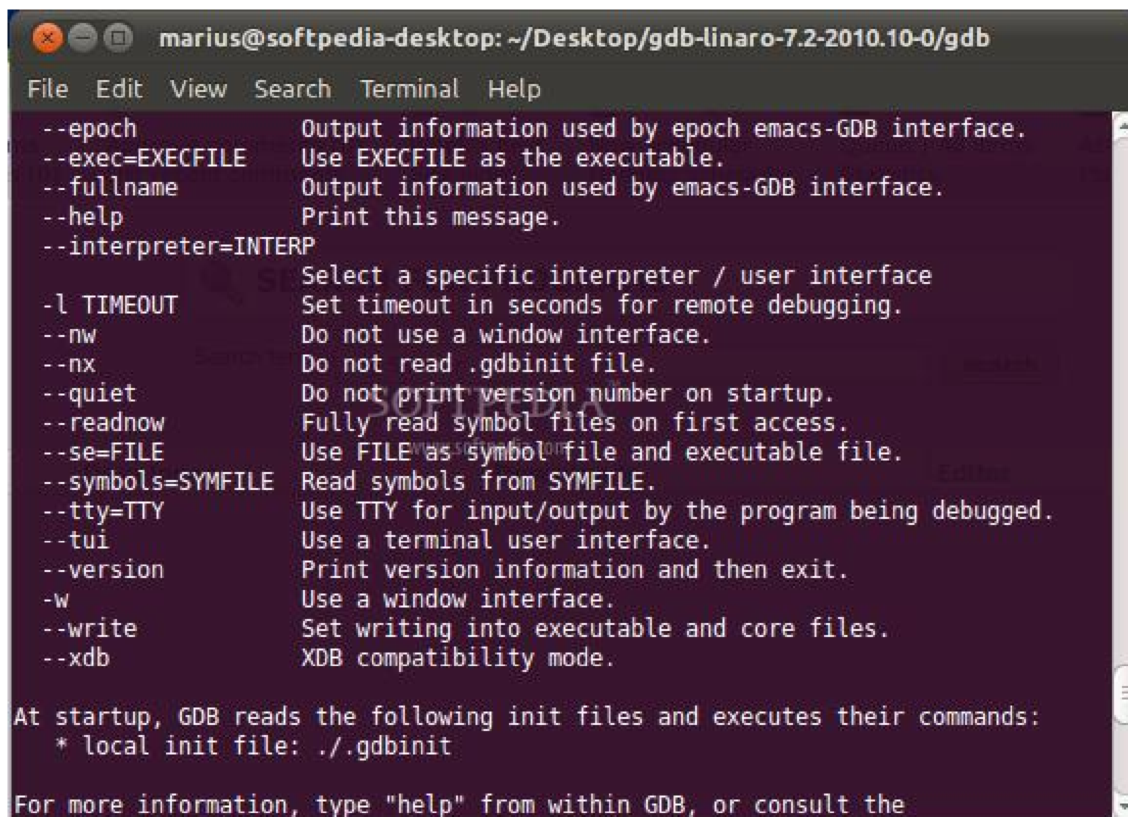


Figura 2.2: Ajuda do GDB *Screenshot*.

2.3 SoftICE

O *SoftICE* [4] é uma ferramenta *software debugging* que fornece da capacidade de fazer debug no nível hardware — modo *kernel*, tem aceso a todo o hardware — a debuggers *PCDOS* e *MSDOS*. É um programa proprietário e de pagamento para *Microsoft Windows* e é desenhado para executar-se sob *Windows* de maneira que o sistema operativo não saiba que está em execução.

O SoftICE foi desenhado com estes três objectivos:

- Utilizar a capacidade da máquina virtual 80386 para conseguir fazer debug.
- Trabalhar com debuggers existentes. Procura-se fornecer a outros debuggers de uma ferramenta com a que possam desenvolver capacidades no nível hardware.
- Ser um programa fácil para o utilizador e que trabalha numa janela.

Além disso, é um debugger excepcionalmente útil na programação de drivers e é compatível com as últimas versões do sistema operativo de *Microsoft*. Na Figura 2.4 podemos ver uma *screenshot* do debugger.

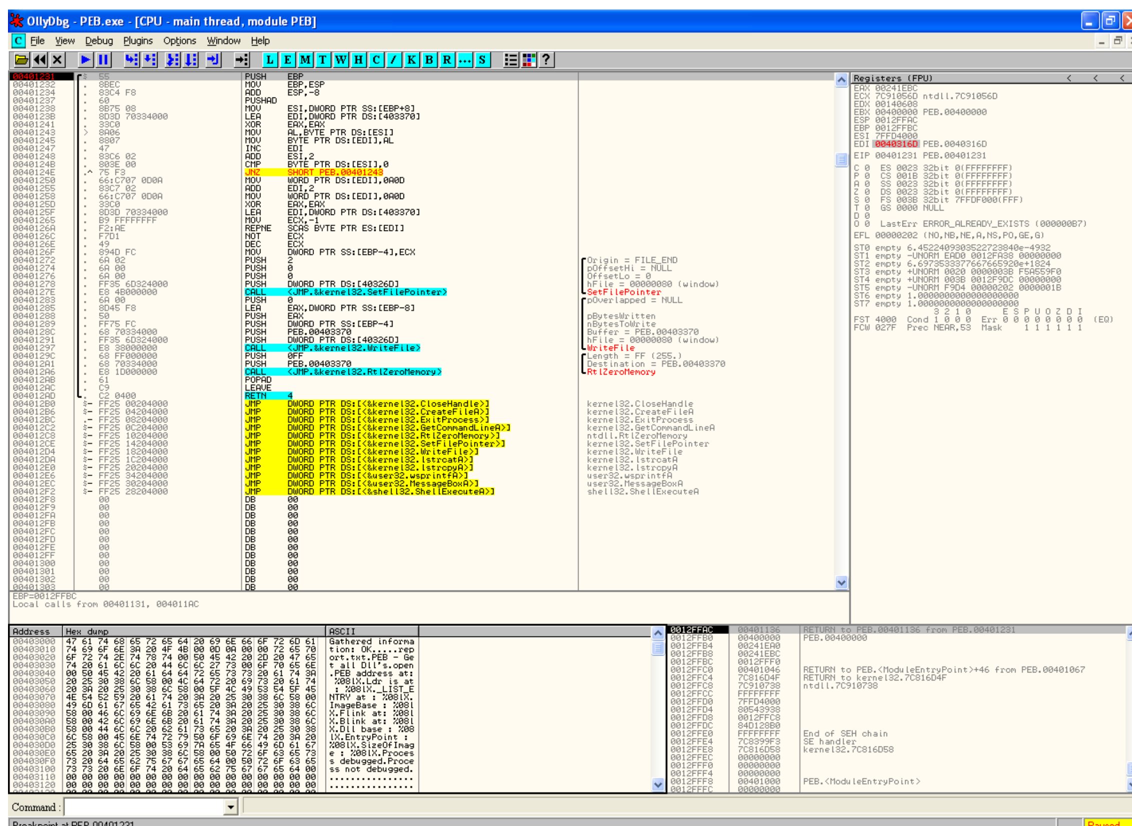


Figura 2.3: OllyDbg *Screenshot*.

A diferença do SoftICE, este projecto vai ser totalmente gratuito para todos os alunos das disciplinas *Arquitectura de Computadores* do IST. Em relação com o modo *kernel*, o projecto vai ter acesso a uma grande parte do hardware mas não tanto como para dizer que vai trabalhar neste modo. Por outra parte, o nosso projecto vai ser um debugger completo, não um programa que vai fornecer de outras funcionalidades a outros debuggers. Além disto, o nosso projecto, como SoftICE, também vai ter uma entrada de comandos.

2.4 MacsBug

O MacsBug [5] é um depurador de baixo nível — linguagem assembly/nível máquina — para o sistema operativo de *Macintosh*. A versão original foi desenvolvida pela *Motorola* como um debugger geral para os seus sistemas *68000*.

O MacsBug oferece muitos comandos para desmontagem, procura e visualização de dados. Não é instalado por defeito nos computadores com *Mac OS* mas todos os *Macintosh* desde o *Macintosh Plus* têm um debugger na sua ROM conhecido como *MicroBug*.

A versão final do MacsBug foi a 6.3.3, desenvolvida em Setembro do ano 2000. Esta versão trabalha com todas as máquinas lançadas no período julho-setembro de 2000, incluindo o *Power*

Figura 2.4: SoftICE *Screenshot*.

Mac G4, o *Power Mac G4 Cube*, o *iMac Family* e o *iBook Family*. Na Figura 2.5 podemos ver um *screenshot* deste debugger.

Como MacsBug, o nosso projecto vai se de baixo nível e vai ter comandos para a visualização e modificação de dados — tantos dos registos internos do P3 como da memória.

2.5 Resumo das características dos Debuggers analizados

Na Tabela 2.1 podemos ver um resumo/comparativa das características dos debuggers analizados nesta seção.

2.6 Características gerais do *Debugger*

Como temos visto nas seções anteriores, o *Debugger* que vamos desenvolver vai ter características de alguns dos debuggers mais importantes do mercado. Para desenvolver estas características vai ser preciso implementar algumas funcionalidades concretas — podem ver-se com mais detalhe na Seção 4.1.

```
SP
03FF0ECE
CE E03AFFD5
D2 00000000
D6 00040013
DA 00000006
DE 00005810
E2 04000000
E6 5E09789A
EA 01D60000
EE 00000400
F2 04003FFC
F6 D9620000
FA 00000000
FE 00040013
02 00000006
06 00005810
0A 04000000
0E 3FFC2830
12 01030000
16 5E09789A
1A 04000190
1E 04000400
22 03FF30C2

CurApName
˘˘˘•˘fÄ˘˘˘…

Int 0  RM
SR  Smxnzvc

D0 00000008
D1 00000001
D2 00000045
D3 00000001
D4 00040013
D5 00000000
D6 000001EA
D7 00000018

A0 3FDE9278
A1 3FF798F0
A2 3FFC2830
A3 01030000
A4 3FDE9068
A5 3FFC83CA
A6 03FF0EF0
A7 03FF0ECE
```

```
  MacsBug 6.6.3, Copyright Apple Computer, Inc. 1981-2000
    Type  HELP  or  ?  to learn about MacsBug commands.

    If you got here by crashing, you may want to try  StdLog  to record
    some details about the crash and your system configuration.

User break at 3FFCDE86 HGETVOL+FE05A

  26-Oct-2007 00:00:05 (since boot = 5 seconds)
  (It looks like "Debugger" was loading.)
  Machine = #1206 (??), System $0922
  ROM version $077D, $45F6, $0001 (ROMBase $FFC00000)
  VM is off
  NIL^ = $FFC10000
  Stack space used = +98
User break at 3FFCDEAC HGETVOL+FE080
 Welcome to MacsBug (Thank you for holding down the Control key)
  (It looks like "Debugger" was loading.)
--------------------------------------------------------------------------------
  HGETVOL
            3FFCDEAC  *MOVE.B     #$01,$0008(A6)                    | 1D7C 0001 0008
            3FFCDEB2   BRA.S      HGETVOL+FE0F6         ; 3FFCDF22  | 606E
            3FFCDEB4   MOVE.W     #$B1E0,-$2510(A5)                 | 3B7C B1E0 DAF0
```

Figura 2.5: MacsBug *Screenshot*.

Característica	GDB	OllyGdb	SoftICE	MacsBug
Leitura/Escrita em memória	✓	✓	✓	✓
Entrada de comandos	✓	✓	✓	✓
Nível Hardware	×	✓	✓	✓
Foco no código Assembly	×	✓	✓	✓
Licença	Livre	Shareware	Proprietário	Proprietário
Corre sob MS	×	✓	✓	✓
Corre sob Linux	✓	×	×	×
Nao precisa instalação	✓	×	×	×

Tabela 2.1: Resumo/Comparativa das características dos debuggers analizados.

O *Debugger* vai precisar ter uma entrada de comandos devido a que vamos usar a própria interface do programa *P3Loader* — que vai ser acrescentado para este alvo — e, desde aí, o utilizador poder interagir com o hardware através de comandos.

Os comandos a desenvolver vão ser cuidadosamente escolhidos de acordo com as funcionalidades a desenvolver. Estas vão ser:

Ligado e desligado do *Debugger* Para atender esta funcionalidade vão ser criados os comandos *startD* — dentro do contexto do *P3Loader* — que vai iniciar o modo debug e ligar o

Debugger e *exit* — dentro do contexto do modo debug — que vai desligar o *Debugger* e voltar ao menu inicial do *P3Loader*. Esta funcionalidade vai ser imposta pela necessidade de ligado e desligado que temos.

Paragem em pontos concretos da execução Para isto devemos fornecer os comandos necessários para o armazenamento de breakpoints. Estes vão ser os comandos *br* que vai armazenar um novo breakpoint e o *del* que vai apagar um dos breakpoints inseridos anteriormente. Esta funcionalidade foi extraída do GDB.

Apresentação dos breakpoints inseridos Esta funcionalidade vai ser fornecida através do comando *list* que vai apresentar pelo ecrã os breakpoints inseridos até o momento.

Modificação e impressão do conteúdo dos registos do P3 e da memória Com o objectivo de desenvolver esta funcionalidade vamos fornecer ao utilizador os comandos *write* que vai modificar o conteúdo do registo ou da memória e o *print* que vai fornecer uma maneira de apresentar os dados da memória ou dos registos pelo ecrã. Esta funcionalidade foi extraída do *MacsBug*, que também a implementa.

Começar e depurar a execução Para isto vamos ter até três comandos para fornecer ao utilizador todas as opções possíveis. Os comandos vão ser: o *run* para começar a execução desde o início do programa com todos os registos a 0; o *cont* que vai continuar a execução do programa desde onde foi parado; e o *step*, que só vai executar uma instrução e vai fazer que o processador volte a ficar parado.

Apresentação do código Para isto vamos ter o comando *code*, que vai apresentar pelo ecrã o conteúdo da memória das posições que o utilizador pedir.

Mostrar uma descrição dos comandos Com o comando *help* vamos executar uma ajuda onde ver como é que os comandos devem ser escritos, as opções que têm e os parâmetros que devem levar. Como vemos na Figura 2.2, esta funcionalidade vai ser semelhante à do GDB.

3

Hardware e Ferramentas de Desenvolvimento

Conteúdo

3.1 O Processador P3 . . . . **14**
3.1.1 História e motivações 14
3.1.2 Arquitectura 14
3.2 Hardware **17**
3.2.1 Digilent D2-SB System Board 17
3.2.2 Digilent DIO5 Peripheral Board 17
3.2.3 Digilent PIO1 Parallel I/O Board 17
3.2.4 Digilent MEM1 Memory Module 1 18
3.3 Arquitectura original do sistema **18**
3.4 Ferramentas de Desenvolvimento **19**
3.4.1 Xilinx ISE 10.1 20
3.4.2 ModelSim PE Student Edition 10.1c 21

3.1 O Processador P3

Como o P3 [6] é um microprocessador CISC, tem um conjunto de instruções amplo e com uma grande semântica em cada instrução. É devido a isto que a codificação e decodificação de cada uma delas vai ter uma maior complexidade que com outros microprocessadores como os RISC.

A programação do P3 é feita através de instruções assembly. Ainda que seja um processador CISC, a codificação das instruções assim como o hardware do próprio P3 são simples — comparado com outros CISC — com o alvo do aprendizagem dos alunos.

Cada uma das instruções têm umas microinstruções que devem ser executadas para a realização passo a passo da ação que a instrução deve completar.

Tanto o hardware do P3 assim como a codificação e as microinstruções das instruções são conhecidas e dadas nas aulas de *Arquitectura de Computadores*.

3.1.1 História e motivações

O P3 foi inicialmente desenvolvido pelos docentes da disciplina *Arquitectura de Computadores* do IST. Quando se propuseram desenhar um novo processador tiveram dois objectivos principais:

- Uniformizar a matéria das aulas teóricas e das aulas práticas. Até aquele momento, em teoria, estudava-se um processador sem existência real, e no laboratório desenvolviam-se trabalhos com *assembly x86*.
- Permitir a introdução de novas instruções assembly e alteração das existentes através da micro-programação.

3.1.2 Arquitectura

3.1.2.A Registos.

Vamos descrever os registos internos que são visíveis ao programador de P3. É necessário dizer que todos os registos são inicializados a 0 depois de um reset do processador:

R0-R7 Registos para uso genérico. Podemos armazenar qualquer dado neles mas o registo R0 sempre vai ficar 0.

PC *Program Counter* ou Contador de Programa. É um ponteiro para o endereço da próxima instrução.

SP *Stack Pointer*. Apontador para o topo da pilha. É utilizado de forma directa — só na inicialização — e indirecta. Usado, principalmente, para armazenar o estado do P3 quando se executa uma interrupção.

RE Registo de Estado. Nos 5 últimos bits deste registo estão armazenados os bits de estado do processador. Este registo não se pode manipular directamente.

3.1.2.B Bits de Estado.

No processador existem 5 bits de estado do ponto de vista do programador. Os restantes bits deste registo têm o valor 0. Estes bits são:

O *Overflow*. Mudará a 1 se o valor da última operação aritmética excede a capacidade do operando destino.

N *Negative*. Indica que o resultado da última operação aritmética foi negativo.

C *Carry*. Indica que a última operação gerou um bit de transporte para além da última posição do operando destino.

Z *Zero*. Mudará a 1 se o valor resultante da última operação foi 0.

E *Enable interrupts*. Se está a 1 habilitará as interrupções e desabilitar-as se for 0. Único bit de estado que só é modificado por instruções.

3.1.2.C Memória.

O espaço endereçável de memória é de 64k palavras com um comprimento de cada palavra de 16 bits. Há numerosos modos de endereçamento e com qualquer deles podemos aceder a qualquer dos endereços.

3.1.2.D Entradas/Saídas.

Os endereços de memória a partir do endereço FF00h estão reservados para o espaço de entradas e saídas. Desta maneira, vemos que o espaço de entradas e saídas é *memory mapped*.

Os dispositivos entrada/saída que temos à nossa disposição são:

Janela de texto Fornece uma interface com o teclado e o ecrã do computador. Temos 4 portos nesta janela: leitura (FFFFh) — porto que permite receber caracteres teclados na janela de texto —, escrita (FFFEh) — porto que permite escrever um dado caracter na janela de texto —, estado (FFFDh) — porto que permite testar se houve alguma tecla premida na janela de texto — e controlo (FFFCh) — porto que permite posicionar o cursor na janela de texto.

Botões de pressão 15 interruptores de pressão. A ativação de cada um de estes botões gera uma interrupção com o correspondente vector de interrupção.

Interruptores 8 interruptores cujo estado pode ser lido através do endereço FFF9h.

LEDs Cada um dos bits da palavra escrita no porto FFF8h vai definir o estado de cada um dos 16 leds.

***Display* de 7 segmentos** São 4 *displays* — conjunto de 7 LEDs — que, através de escritas nos portos FFF0h, FFF1h, FFF2h e FFF3h podem ser alterados.

LCD *Display* de texto com 2 linhas e 16 colunas. Tem istos 2 portos: FFF5h (para escrever um dado carácter) e FFF4h (para posicionar o cursor).

Máscara de interrupções Filtro posicionado no endereço FFFAh que permite selecionar quais dos 16 primeiros vectores de interrupção estão habilitados.

Temporizador Fornece a geração de uma interrupção depois de um intervalo de tempo real definido pelo utilizador. Tem 2 portos: FFF7h — controlo; liga ou desliga o temporizador — e FFF6h — para indicar o número de intervalos de 100 ms que queremos decorrer antes do lançamento da interrupção.

3.1.2.E Interrupções.

O simulador disponibiliza de 16 botões para interrupções dos quais só 15 têm atribuída uma interrupção. Qualquer destas interrupções ativa um sinal *INT* ligada a um dos pinos externos do processador que são testados depois de cada instrução para verificar se há alguma interrupção pendente.

O botão que ainda não tem interrupção atribuída é o que vamos usar no nosso projecto para que o utilizador possa interromper a execução do P3 entretanto esteja no modo debug.

3.1.2.F Microcódigo.

Como já foi dito anteriormente, para executar as instruções devem ser executadas as microinstruções que têm associadas. Para isto o P3 tem três memórias ROM: a ROM A, a ROM B e a ROM de controlo. O conteúdo da ROM A assim como da ROM B vão ser ponteiros para endereços da memória ROM de controlo que é onde vai ficar armazenado todo o microcódigo.

Na ROM A vai ter armazenados os endereços da ROM de controlo correspondentes aos modos de endereçamento. Assim, dependendo da instrução e os dados que tiver que ler ou escrever, o CAR vai ser carregado com o endereço concreto que vai executar a microrotina que precisamos da ROM de controlo.

Por outro lado, na ROM B vão ficar armazenados os endereços da ROM de controlo correspondentes às microrotinas de cada instrução. Também vai ser carregado no registo CAR para a sua execução.

Por último, e como também foi dito anteriormente, na ROM de controlo vai ser armazenado todo o microcódigo que vai ser apontado pelos conteúdos das outras duas memórias ROM.

Todas estas memórias podem ser modificadas pelo que vai ser possível inserir novas instruções ou modificar as existentes. Através do programa *P3Loader* vai ser possível modificar estas memórias para ajustar o funcionamento às nossas necessidades.

3.2 Hardware

O hardware que vamos usar para a implementação do P3 consiste em quatro módulos.

3.2.1 Digilent D2-SB System Board

Figura 3.1: Placa *D2-SB*. Peça central, contém a FPGA a programar.

A *Digilent D2-SB System Board* [7] é o elemento central da placa — Figura 3.1. Contém uma FPGA *Xilinx Spartan2E XC2S200E PQ208* [8] ligada aos restantes módulos através de conectores de 40 pinos. A FPGA está ainda ligada a um oscilador eletrônico de 50 MHz, um botão de pressão e um LED. A programação da FPGA pode ser guardada numa PROM *XC18V02* evitando assim a necessidade de reprogramação a cada utilização.

3.2.2 Digilent DIO5 Peripheral Board

A *Digilent DIO5 Peripheral Board* [9] contém os dispositivos de entrada/saída necessários: 16 botões de pressão, 16 LEDs, 8 interruptores, um display de sete segmentos com 4 dígitos, um LCD com 16 x 2 caracteres, uma porta PS/2 e uma porta VGA. Contém ainda um CPLD *Xilinx CoolRunner XCR3128XL TQ144* que permite fazer encaminhamento de sinais ou algum processamento mais elementar. Na Figura 3.2 podemos ver como é a placa.

3.2.3 Digilent PIO1 Parallel I/O Board

A Figura 3.3 mostra-nos o dispositivo hardware *Digilent PIO1 Parallel I/O Board* [10], que permite comunicar a placa com outros dispositivos usando uma porta paralela via SPP ou EPP [11].

Figura 3.2: Placa *DIO5*. Contém os dispositivos de entrada e saída.

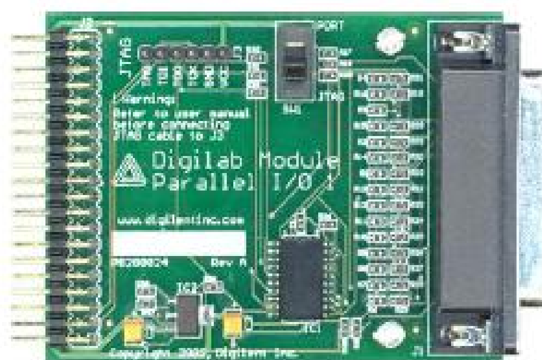


Figura 3.3: Placa *PIO1*. Fornece um meio de comunicação através de uma porta paralela.

3.2.4 Digilent MEM1 Memory Module 1

Esta última placa, a *Digilent MEM1 Memory Module 1* [12], é um banco de memórias que, neste caso, contém uma SRAM *ISSI IS61 LV51 28AL-10TI* — vai ter uma capacidade de 512KB; 512K endereços com 1 byte por endereço [13] — e uma memória flash *28F004B3* — também vai ter uma capacidade de 512KB; 512K endereços com 1 byte por endereço [14]. Podemos ver a placa na Figura 3.4.

3.3 Arquitectura original do sistema

No esquema da Figura 3.5 vemos a representação da antiga arquitectura que se programava na FPGA — desenvolvida numa tese anterior [1] —, sem as modificações realizadas. Nesta figura podemos ver que está composta por cinco módulos:

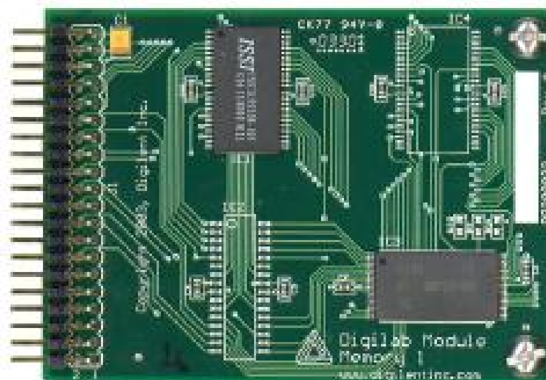


Figura 3.4: Placa *MEM1*. Encarregada de fornecer de uma memória flash ao conjunto.

P3 Contém o processador, controlador de entradas/saídas e gestor de interrupções.

Clock Control Responsável por gerar os diversos sinais de relógio necessários e os sinais de inicialização.

MEM1 Interface Possibilita a leitura/escrita na memória externa.

PIO1 Interface Possibilita a leitura/escrita de dados através da porta paralela.

DIO5 Interface Responsável pela comunicação com a placa DIO5 e, implicitamente, com os dispositivos de entrada/saída.

Vai ser neste nível onde vamos inserir o dispositivo *Debugger*. Este dispositivo vai interagir com outros de esta arquitectura já implementados como vamos poder ver na seção 4.3.

Nesta arquitectura, além de inserir o dispositivo *Debugger*, vamos ter que fazer algumas modificações tanto nas ligações entre os dispositivos como dentro do próprio P3 e alguns dos seus componentes.

3.4 Ferramentas de Desenvolvimento

Nesta seção vamos ver as ferramentas software de desenvolvimento que foram necessárias para a realização da tese.

Para começar dizer que para carregar os programas assembly no P3 programado na FPGA, vamos usar o programa *P3Loader*, feito em *C++*. Este programa deve ser modificado para que, além do carregamento das memórias do P3, também possa entrar no modo debug e interagir com o processador quando fica parado. Desta maneira o utilizador vai poder realizar o debug ao seu programa assembly.

Como compilador podia ser escolhido qualquer que dera para *C++*. Finalmente, foi escolhido o *Microsoft Visual C++ 2008 Express Edition* devido, além da alta funcionalidade que tem, que

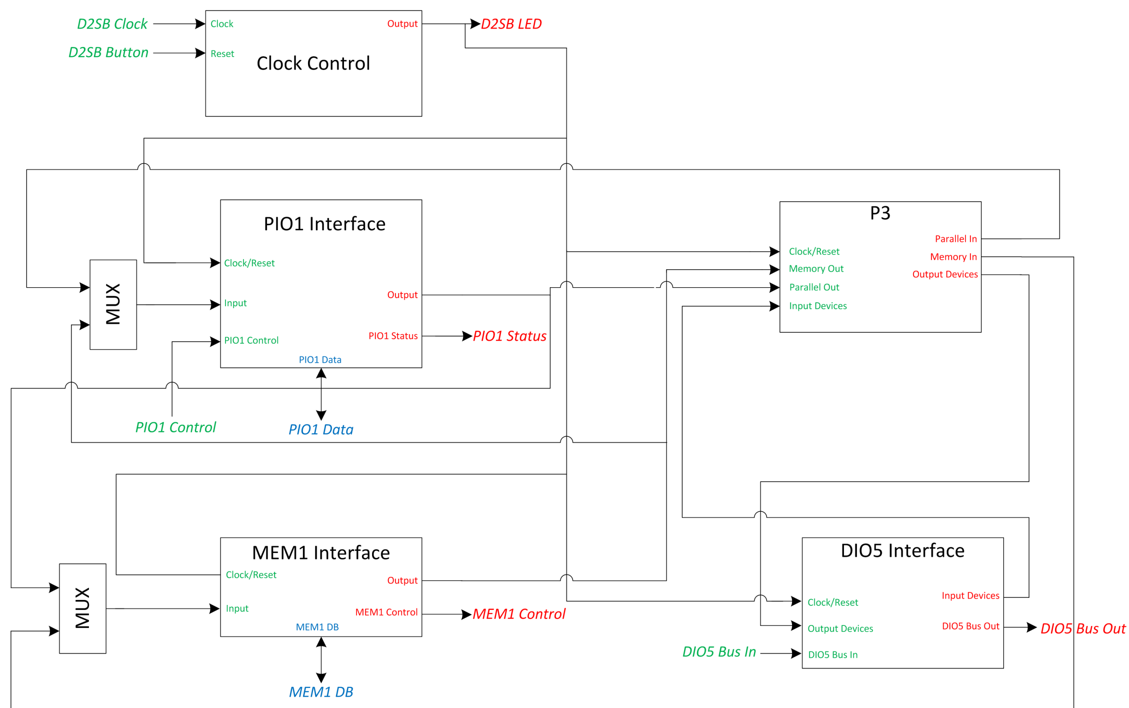


Figura 3.5: Arquitectura original do sistema.

neste compilador realizou-se as primeiras versões do programa *P3Loader*, pelo que já sabemos que corria bem. Na Figura 3.6 podemos ver uma *screenshot* do programa.

3.4.1 Xilinx ISE 10.1

O software principal utilizado vai ser o *Xilinx ISE Design Suite10.1* [15]. Este software é constituído por diversas ferramentas de desenvolvimento de sistemas digitais. Podemos ver uma *screenshot* dele na Figura 3.7.

A principal razão da escolha de este software é que vamos a trabalhar sobre hardware Xilinx. Neste pacote temos o editor VHDL, sintetizador, e compilador que gera uma implementação e é capaz de carregar todo no dispositivo.

Com esta ferramenta vamos poder desenvolver e depurar o código VHDL que escrevamos para o desenho do *Debugger* e as modificações ao P3 para a sua adaptação. Depois, com a ferramenta *iMPACT* — vem incluída no programa — podemos carregar o desenho desenvolvido na FPGA para começar a sua depuração.

Podemos perguntar-nos o porquê de não usar a última versão deste programa mas a resposta é simples: as últimas versões do *Xilinx ISE* não permitem configurar o dispositivo FPGA que temos na nossa placa.

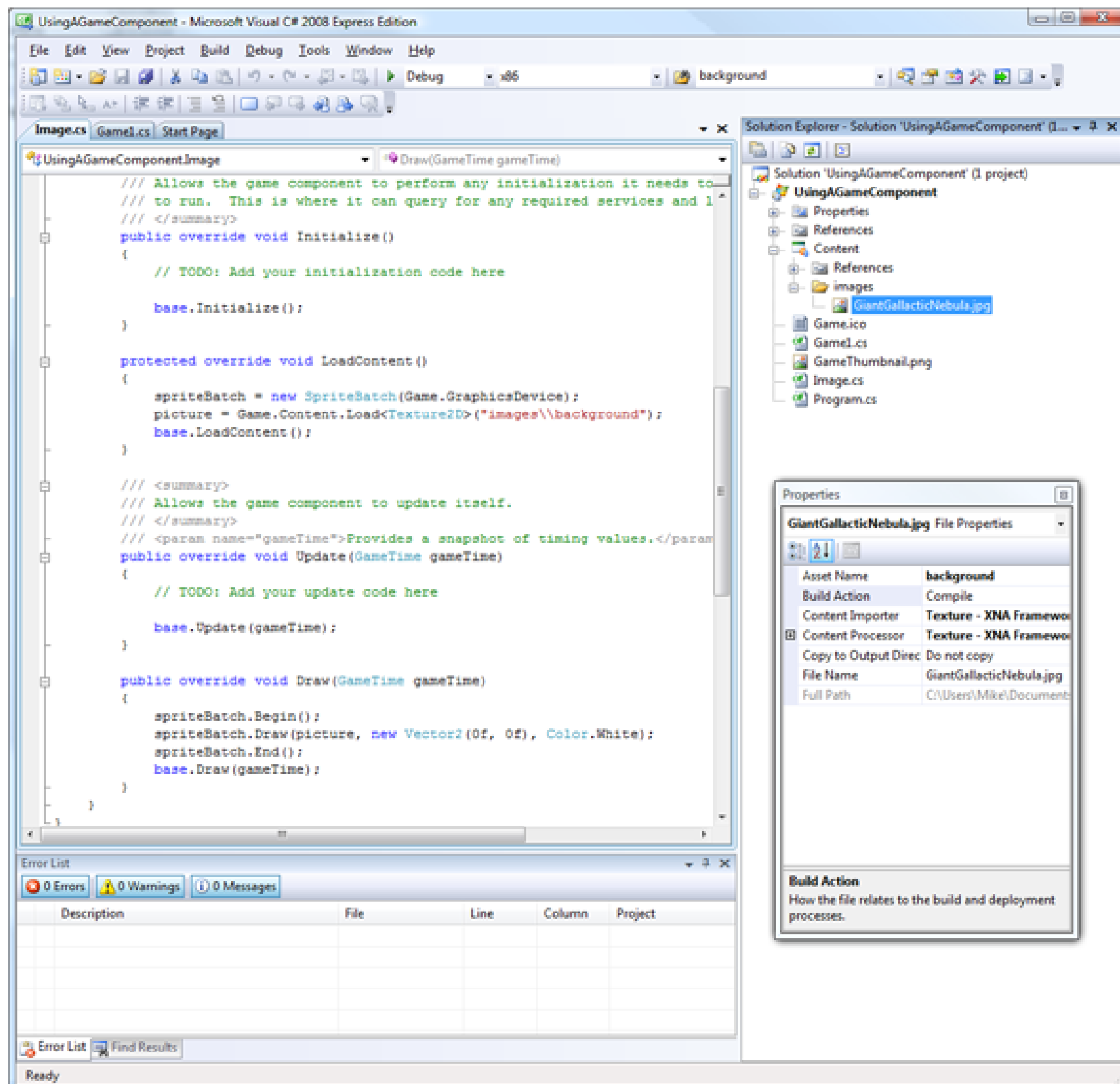


Figura 3.6: Microsoft Visual C++ 2008 Express Edition *Screenshot*.

3.4.2 ModelSim PE Student Edition 10.1c

Para a simulação das partes separadas da tese foi preciso o uso do *ModelSim PE Student Edition 10.1c*. Há outras versões mais completas mas a versão para estudantes era a única que não levava um custo económico adicional. Na Figura 3.8 podemos ver uma *screenshot* da versão utilizada.

Além de alguma limitação no tamanho do projecto, este software é muito completo e prático. Com ele pudemos ver os cronogramas dos sinais, o *dataflow*, etc, mas não podemos simular todo o projecto completo por causa do tamanho do mesmo.

Com esta ferramenta podemos olhar todos os sinais que o dispositivo tem e forçar-lhos para poder ver o comportamento do desenho. Os sinais podem ser forçadas a 0, a 1 ou como relógio — o período é configurável — para depois pôr a correr o dispositivo — se o utilizador quer, o

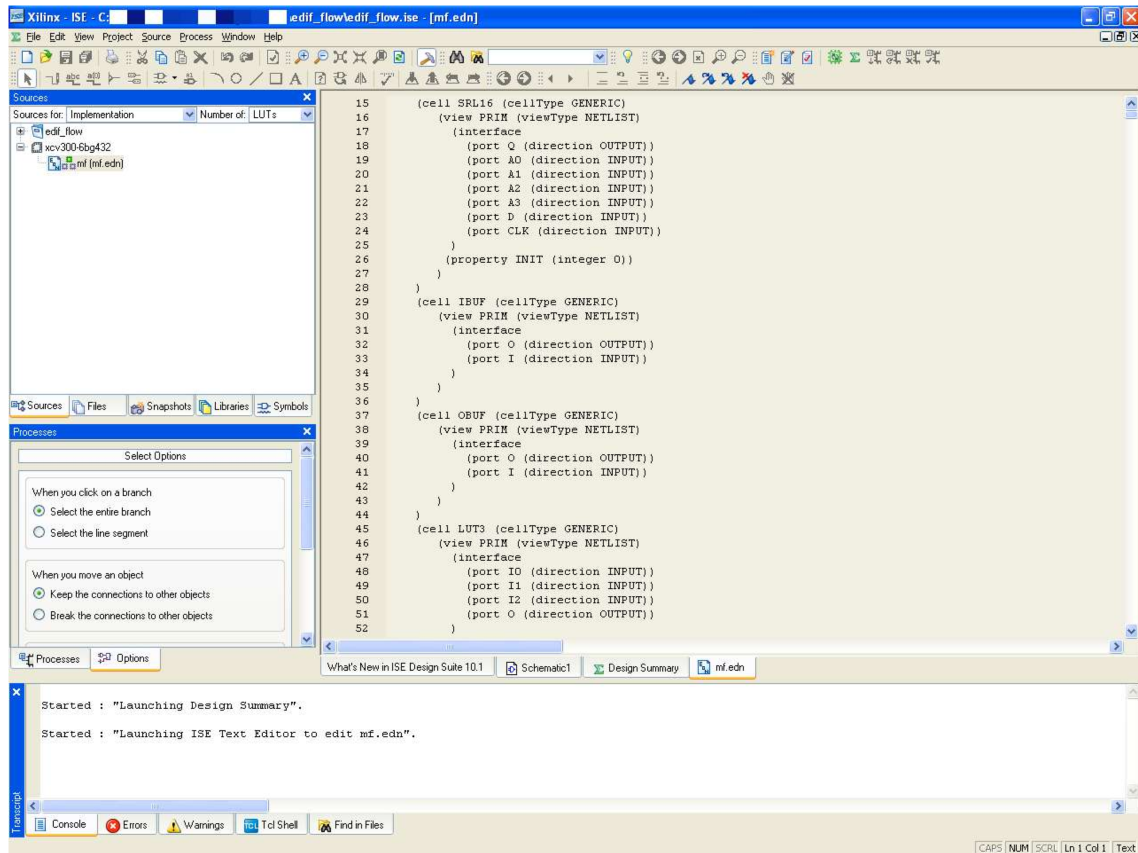


Figura 3.7: Xilinx ISE 10.1 *Screenshot*.

dispositivo pode correr só um tempo concreto — e poder remover falhas vendo como é que os sinais interagem.

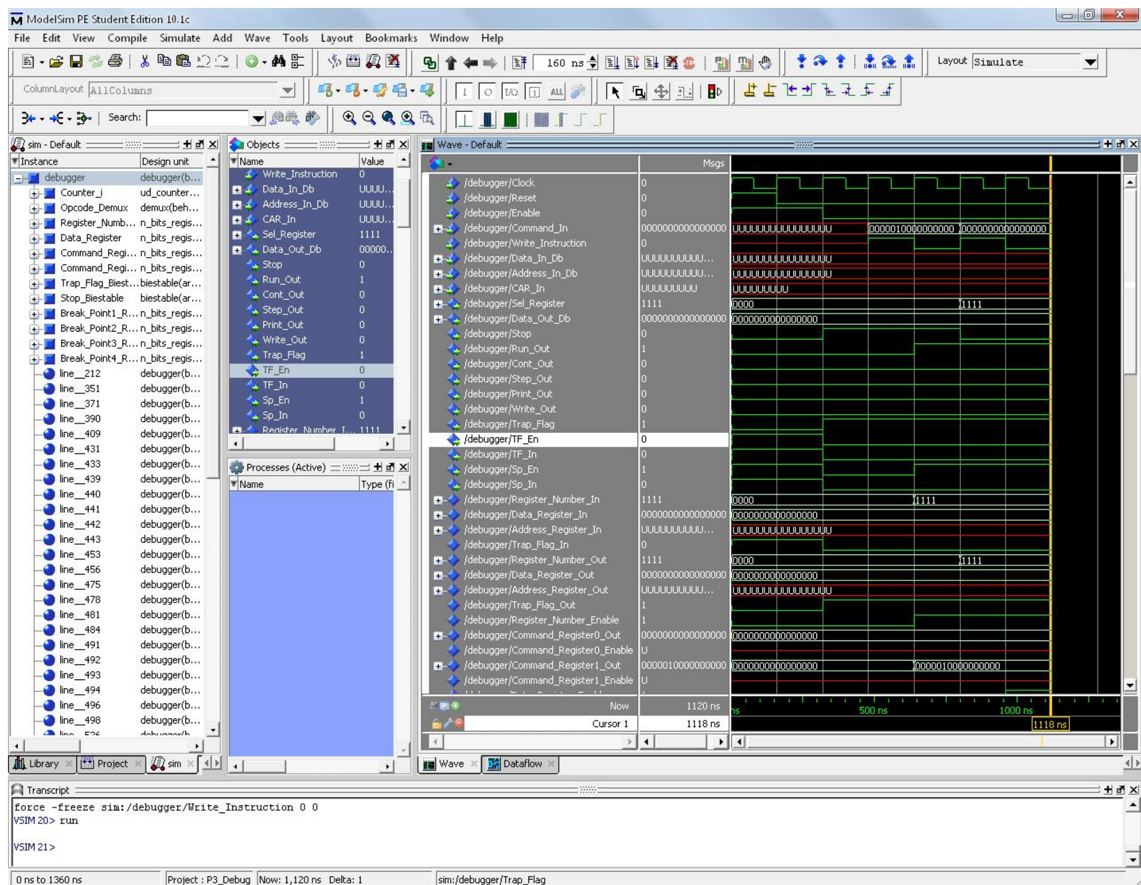


Figura 3.8: ModelSim PE Student Edition 10.1c *Screenshot*.

4

Arquitectura

Conteúdo

4.1 Funcionalidades a desenvolver **26**
4.1.1 O comando Run 26
4.1.2 O comando Cont 26
4.1.3 O comando Step 26
4.1.4 O comando List 27
4.1.5 O comando Code 27
4.1.6 O comando Br 27
4.1.7 O comando Del 28
4.1.8 O comando Print 28
4.1.9 O comando Write 28
4.1.10 O comando Exit 29
4.2 O *Debugger* **29**
4.2.1 Ativar e desativar o modo *Debugger* 29
4.2.2 Armazenamento dos Breakpoints 31
4.2.3 Decodificação dos Comandos 31
4.3 Modificações ao D2-SB **35**
4.4 Modificações ao Processador P3 . . . . **37**

4.1 Funcionalidades a desenvolver

O *Debugger* vai ter para armazenar até quatro *breakpoints*. Isto é facilmente escalável pelo que no futuro pode-se ampliar. Além disto, as principais funcionalidades vão ser os comandos que o utilizador vai poder inserir.

Por cada comando inserido com implementação em hardware — alguns dos comandos só vão ter implementação em software —, o *P3Loader* modificado vai enviar pela porta paralela quatro bytes em dois envios — dois bytes por envio; cada um de estes envios vai ser armazenados em uns registos que vamos ver na Seção 4.2.3. O primeiro dos envios vai ter o *Opcode* — 6 bits na parte mais significativa — do comando e o segundo vai ficar reservado para os comandos que tenham que enviar algum dado ao *Debugger*. Quando um comando não precisar de enviar nenhum dado, estes dois últimos bytes vão ficar a 0.

4.1.1 O comando Run

Este comando executa o código desde o início. Se há algum *breakpoint* a execução vai ficar parada na posição de memória que esteja inserida no *breakpoint* e ficará à espera de algum novo comando.

Para isso, o utilizador deve inserir o comando da seguinte maneira: *run*. Quando é inserido no programa *P3Loader*, este envia no primeiro dos envios os bytes 0x0400 para o reconhecimento do comando. O segundo envio vão ser dois bytes a 0.

4.1.2 O comando Cont

Continua com a execução do programa depois duma paragem. Se houver algum *breakpoint* a execução vai ficar parada na posição de memória que esteja inserida no *breakpoint* e ficará à espera de algum novo comando.

Com o comando *cont* o *P3Loader* vai funcionar de forma parecida ao *Run*. A principal diferença é que este comando não vai reiniciar a execução do programa pelo que vai seguir a correr desde onde ficou parado.

Neste caso os bytes enviados no primeiro envio vão ser 0x0800 mas, devido a não ter nenhum outro argumento, o segundo envio vão ser bytes a 0.

4.1.3 O comando Step

Com o P3 parado, este comando executa só uma instrução. Este comando é ótimo para ver como é que corre um programa instrução por instrução e ver onde é que falha.

Para isso, o utilizador vai inserir o comando *step* e o programa vai enviar 0x0C00 no primeiro envio e 0 no segundo.

4.1.4 O comando List

Mostra no ecrã todos os *breakpoints* armazenados neste momento. Para chamar corretamente a este comando deve escrever-se *list*.

Para implementar esta funcionalidade não foi preciso inserir hardware adicional devido a que quando um *breakpoint* é inserido o *P3Loader* armazena-o num *array*. Pelo que quando este comando é executado só percorremos o *array* mostrando cada um dos elementos do mesmo.

É por isso que também não é preciso nenhum argumento adicional.

4.1.5 O comando Code

O comando *code* tem dois argumentos: um endereço de memória e um número de posições. Vai mostrar no ecrã o conteúdo da memória desde o endereço até o endereço mais o número de posições.

Para executar este comando também não é preciso desenvolver nenhum hardware adicional devido a que a implementação inicial do dispositivo e do *P3Loader* já implementavam leituras e escritas nas memórias pelo que a implementação de este comando é só um ciclo a fazer leituras e apresentações dos dados no ecrã.

4.1.6 O comando Br

Tem um argumento, um endereço de memória. Introduz este endereço como novo *breakpoint* se anteriormente não foi inserido e ainda não há quatro *breakpoints* — só se pode inserir quatro *breakpoints* como máximo.

Uma grande parte do desenho desenvolvido tem como alvo a execução do comando *br* devido a que é uma das partes mais importantes do *Debugger*. A comparação dos *breakpoints* com o *Program Counter* do P3 que entretanto esteja a correr o programa, é uma das grandes vantagens que tem o dispositivo implementado em relação à solução antiga, o simulador.

Quando é executado este comando a primeira coisa que o programa faz é verificar que o *breakpoint* não foi inserido anteriormente e que ainda não temos 4 *breakpoints* inseridos. Se assim for, faz os envios da seguinte maneira: no primeiro dos envios o *P3Loader* vai enviar o código correspondente ao comando, 0x1400; no outro envio vão estar os dois bytes do *breakpoint* a inserir e que foram passados como argumento.

Além disso, o programa vai inserir o endereço num *array* para facilitar a implementação dos comandos *List*, *Del* e a do próprio comando *br* — deve verificar se o *breakpoint* a inserir não foi inserido anteriormente.

4.1.7 O comando Del

Tem um argumento, um endereço de memória. Este endereço vai indicar o *breakpoint* que estamos a querer remover da lista dos *breakpoints*.

Quando é inserido o comando *del* no programa, a primeira coisa que faz é a verificação de que o endereço que vem como argumento é um dos *breakpoints* inseridos anteriormente com ajuda do *array* implementado em software.

Se assim for, o *P3Loader* vai começar fazer os envios pela porta paralela. O primeiro deles vai ser 0x1800 correspondente ao *Opcode* do comando e no segundo vai enviar o *breakpoint* que estamos a remover.

4.1.8 O comando Print

Este comando vai mostrar algum dado concreto através da inserção da cadeia *print*. Estes dados podem ser dados extraídos da memória ou dos registos internos do P3. Tem dois argumentos: o primeiro vai ser uma cadeia de caracteres que vai poder ser *-r* ou *-a*; o segundo vai ser RX — sendo X o número de registo concreto — ou um endereço de memória respectivamente.

Com o *-r* como o primeiro dos argumentos quer dizer que o dado a mostrar vai ser um registo. Neste caso é preciso fazer envios ao *Debugger* já que não temos outra maneira de aceder a estes dados. O primeiro dos envios ao dispositivo vão ser, como sempre, dois bytes: o primeiro de estes bytes vai ser sempre o mesmo, 0x1C; no entanto, o segundo byte vai ser diferente por cada envio devido a que os seus últimos 4 bits vão indicar ao *Debugger* o número de registo que estamos a mostrar.

No caso no que o primeiro argumento seja *-a* o dado a obter vai ser um da memória. Desta maneira, o dado vai ser obtido diretamente através do software implementado anteriormente da mesma maneira que com o comando *Code*.

4.1.9 O comando Write

Este comando vai escrever um dado num concreto registo ou endereço de memória. É por isto que vai ter três argumentos: o seu primeiro argumento vai ser *-r* ou *-a*, o segundo argumento vai ser o registo — RX — ou o endereço de memória onde queremos escrever e o terceiro vai ser o dado a inserir.

O comando *write* vai ter mais um argumento que o *Print* mas os outros dois argumentos vão ter o mesmo significado que no comando anterior.

Assim, se a escrita for em memória — primeiro argumento *-a* — vai ser implementado em software usando as funções do *P3Loader* já implementadas e o hardware já desenvolvido anteriormente. No entanto, se a escrita for num dos registos do P3 — primeiro argumento *-r* — vai ser necessário fazer os envios correspondentes ao *Debugger*. No primeiro dos envios vai acontecer

o mesmo que com o comando anterior: o primeiro dos bytes vai ter um valor fixo — 0x20 — mas o segundo vai depender do número do registo que o utilizador quer escrever. O segundo envio vai ser o dado a inserir.

4.1.10 O comando Exit

Este comando vai sair do modo debug e deve ser executado com a palavra *exit*.

É preciso notificar ao *Debugger* que vamos fechar o modo debug. Para isto o *P3Loader* faz os seguintes envios pela porta paralela: O primeiro dos envios vai ter o valor 0x2400 e o segundo vai ser todo 0.

Depois de fazer os envios, o *P3Loader* para o P3 e mostra uma mensagem de despedida.

4.2 O *Debugger*

Como já temos visto, o *P3Loader* vai oferecer umas funcionalidades através do teclado e do ecrã e vai interagir com o hardware desenhado.

A seguir vamos ver pormenorizadamente o hardware desenvolvido para o correto funcionamento da interface software *P3Loader*.

4.2.1 Ativar e desativar o modo *Debugger*

Nesta parte da implementação do hardware vemos na figura 4.1 como os componentes principais são três registos, todos de só um bit:

Trap Flag Register Vai ser 1 quando o modo debug esteja ativo e 0 quando não.

Stop Register Vai ser 1 quando o *Debugger* precisar parar o P3 e 0 quando não.

RunCont Register É um registo que vai ficar a 1 sempre que os sinais *Run* ou *Cont* ativarem-se. Com o seu sinal de saída ativada, vai fazer ficar o *Stop Register* a 0. Quando é ativado, vai ficar assim até chegar um sinal que puder desativar-lho — um dos mesmos sinais que vão por a 1 o *Stop Register*.

Estes vão ser os registos mais importantes do *Debugger* devido à importância da sua responsabilidade.

O *Trap Flag Register* só vai ser 1 quando o *Enable* se ativar e vai ficar assim até que o comando *Exit* seja introduzido. A sua saída vai ligar o *Enabler* — Seção 4.2.3 — pelo que se não ficar ativo não vai ser possível o funcionamento do *Debugger*.

Por outro lado, o valor do *Stop Register* vai ser mais difícil de obter devido a que só deve ser ativado quando o *Debugger* precisar que o P3 ficar parado. Estes casos são concretamente os seguintes:

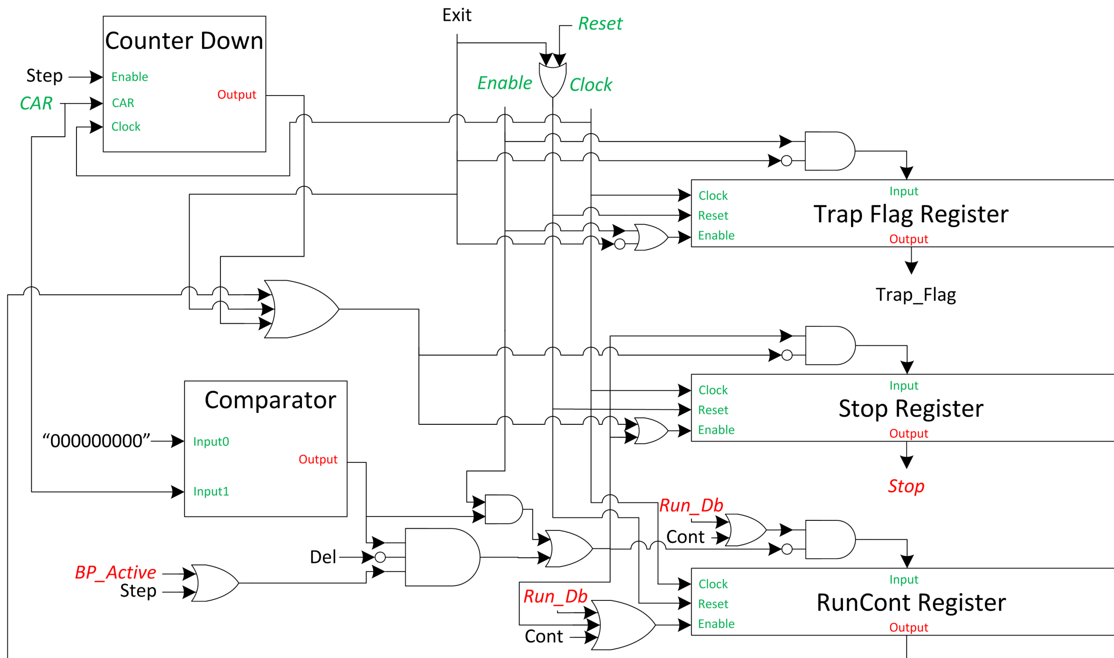


Figura 4.1: Hardware do ligado e desligado do *Debugger*.

Breakpoint ativado Quando o endereço do *Addr_In_Db* é o mesmo que o endereço dum dos armazenados nos *Breakpoint Register* e, consequentemente, é ativado o sinal *BP_Active*.

O sinal Step Estando o P3 parado, um sinal *Step* deve ativar o funcionamento do P3 — o *Stop Register* deve ficar a 0 — mas só na execução duma instrução, depois o *Stop Register* deve voltar ser 1.

Os sinais Run, Cont e Exit Estando o P3 parado, qualquer destas sinais vão pôr o *Stop Register* a 0 e, consequentemente, o P3 a correr.

Como foi dito anteriormente, cada instrução tem várias microinstruções pelo que para o primeiro dos casos é preciso ver o valor do CAR — registo que armazena o endereço da seguinte microinstrução a executar da ROM de controlo — para que o P3 só parar quando esteja a "000000000", que é o endereço da primeira microinstrução de todas as instruções.

Também temos o dispositivo *Counter Down* que ativa o seu sinal de saída durante os ciclos de relógio que durar a execução da seguinte instrução quando o seu sinal *Enable* fica ativada. Isto é preciso para quando o sinal *Step* seja ativada, o *Stop Register* fique a 0 o tempo suficiente para que o CAR avance e seja diferente de "000000000" até que volte ficar todo a 0. O sinal do *Stop Register* vai voltar a ser 1 devido ao sinal *Step* quando a saída do *Counter Down* fique a 0 depois do CAR voltar a ser todo 0 como foi dito anteriormente.

4.2.2 Armazenamento dos Breakpoints

Esta parte vai ser das mais complexas do *Debugger* devido, principalmente, à quantidade de combinatória que temos que inserir. É por esta razão que só vamos detalhar o desenho do armazenamento de um *breakpoint*. Os outros três vão ser iguais.

Como podemos ver na figura 4.2, o armazenamento de cada *breakpoint* vai ter associado os seguintes componentes:

Breakpoint Register É o elemento principal deste desenho devido a que é onde se armazena o *breakpoint*. Tem 17 bits: 16 para o armazenamento do *breakpoint* e o bit mais significativo vai ser o bit de válido que vai indicar quando é que o *Breakpoint Register* tem um endereço válido.

Mux Address Vai estar ligado à entrada *Addr_In_Db* e ao *Command Register 0*. Normalmente a saída vai ser o sinal *Addr_In_Db* mas quando os sinais *Br* ou *Del* se ativarem, a saída vai ser o *Command Register 0* devido a que estamos a armazenar ou a remover um *breakpoint*, respectivamente.

Comparator Compara o endereço do *Breakpoint Register* com a saída do *Mux Address*. A sua saída vai ser 1 no caso que estes sejam iguais e 0 noutro caso.

Mux Breakpoint Vai controlar a entrada do *Breakpoint Register*. Esta entrada pode ficar a 0 se o sinal *Del* estiver ativa e a saída do *Mux Address* noutro caso.

Além de todo isto, também vemos como se gera a saída *BP_Active* que vai ficar a 1 sempre que alguma das saídas dos comparadores seja 1 e o bit de válido do *Breakpoint Register* correspondente seja também 1.

4.2.3 Decodificação dos Comandos

Nesta seção vamos ver o desenho da parte destinada ao armazenamento e decodificação dos comandos do dispositivo *Debugger*. Na figura 4.3 podemos ver o desenho desta parte do projecto. Estes são os seus componentes:

Command Registers Estes registos estão destinados ao armazenamento dos códigos enviados pelo *P3Loader* através da porta paralela correspondentes aos comandos que o utilizador está a inserir. São dois registos de 16 bits cada um que serão habilitados alternativamente pelo *Enabler*.

Enabler A sua entrada vai ficar a 1 cada vez que o programa *P3Loader* faz um envio de dados ao *Debugger*. Como vamos ver mais adiante, quando o utilizador inserir um comando, o programa vai fazer dois envios pela porta paralela de dois bytes cada um. É por isto que

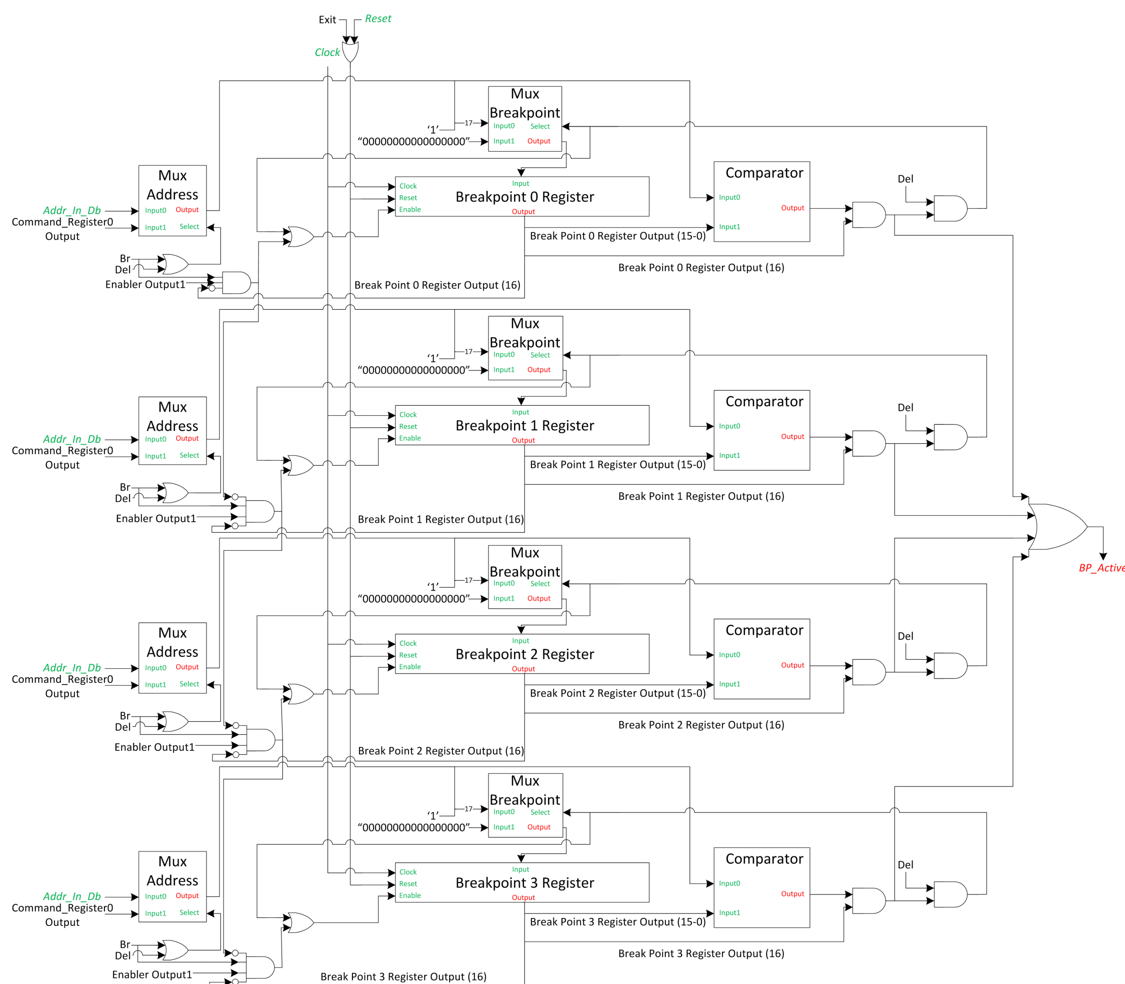


Figura 4.2: Hardware do armazenamento dos *Breakpoints* do *Debugger*.

cada vez que a entrada do dispositivo seja 1, vai ativar-se uma das suas saídas que vão ficar ligadas aos sinais *Enable* dos *Command Register*. Como já sabemos, nestes registos vão armazenar-se os dados enviados desde o programa pelo que vai ser preciso que as saídas do *Enabler* ativem-se alternativamente quando a sua entrada se ativar.

Demux Este desmultiplexador vai ter a missão da decodificação dos 6 bits mais significativos do *Comand Register 1*, que são os que indicam o comando que está a inserir o utilizador.

Number Register Vai ser um registo de 4 bits que vai armazenar o número do registo do P3 que o *Debugger* quer ler o escrever através das instruções *Print* e *Write* respectivamente.

Data Register É um registo de 16 bits que pode armazenar: valor que o *Debugger* quer escrever num registo do P3 ou o valor lido de um registo do processador antes de enviar-lho pela porta paralela.

Mux Data Register Controla a entrada de dados no *Data Register* que pode ser a saída do

Command Register 0 — nos casos nos que o *Debugger* quer escrever num registo através do comando *Write* — ou *Data_In_Db* — nos casos que o *Debugger* esteja a ler um registo do processador através do comando *Print*.

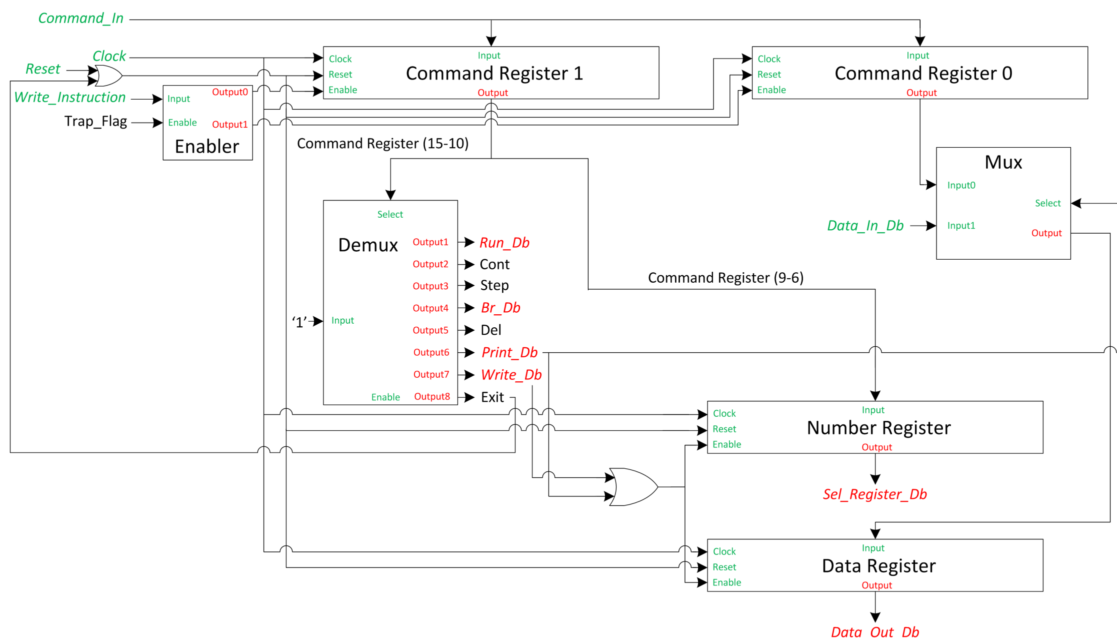


Figura 4.3: Hardware do armazenamento e decodificação dos comandos do *Debugger*.

Na Seção 4.1 vimos todos os comandos que vamos a desenvolver. Os comandos que vão precisar de implementação hardware vão ser decodificados nos *Command Register* — através dos envios que faz o *P3Loader* ao *Debugger*. Agora vamos ver o que é que acontece dentro do *Debugger* quando cada um de estes comandos são inseridos. Nesta parte, vamos ver detalhes do desenho que são pormenorizados nas Seções 4.3 e 4.4:

Run Vai fazer um reset ao *Register Bank* para que todo ficar a 0 e reiniciar o programa. Além disso, põe o *RunCont Register* a 1 para que o *Stop Register* seja 0 até que o utilizador através do botão 15 o pare ou chegar a um *breakpoint*.

Cont Só vai ter efeito no *RunCont Register* da mesma maneira que no comando anterior.

Step Neste caso, o *Debugger* vai ativar o dispositivo *Counter Down* que vai pôr o *Stop Register* a 0 para que comece a execução da seguinte instrução. No entanto, a saída do *Counter Down* só vai ficar a 1 durante os ciclos de relógio que durar a execução da instrução — quando o CAR volte ser 0 o sinal de saída do *Counter Down* também vai ficar a 0 — para que depois o *Stop Register* volte ficar a 1 e, consequentemente, o processador ficar parado.

Br Quando este comando é decodificado nos *Command Register*, o *Debugger* vai inserir o *breakpoint* num dos *Breakpoint Register* que ainda ficam livres ativando o bit de válido do

mesmo. Para isso, a saída do próprio *Command Register* onde fica armazenado o *breakpoint* vai estar ligada às entradas dos *Mux_Breakpoint*. Através da combinatória que podemos ver nos *Enable* dos *Breakpoint Register* — Figura 4.2 —, o armazenamento de um novo *breakpoint* vai ser sempre nos *Breakpoint Register* mais baixos — em relação ao seu índice.

Del Neste caso, o registo onde estava inserido antes o *breakpoint* que estamos a apagar vai ficar todo a 0, incluindo o bit de válido. Para isso o primeiro que o *Debugger* vai fazer é armazenar o endereço do *breakpoint* a apagar no *Data Register* que também vai estar ligado às entradas dos comparadores que verificam qual é o *breakpoint* a remover. O dispositivo vai saber que estamos a apagar um *breakpoint* devido a que, além de que o endereço seja igual que um dos *breakpoints* inseridos — sabemos esto pelos comparadores —, o bit de válido e o sinal *Del* vão ficar a 1.

Print Quando é decodificado no *Debugger* — deve ser inserido o comando com a opção *-r* — ativa-se o sinal *Print_Db* dentro do próprio dispositivo e os 4 bits que indicam o registo alvo armazenam-se no *Number Register* que vai estar ligado a uma das entradas do multiplexador que o sinal *SelAD* do *Register Bank* tem na sua entrada. O sinal *Print_Db* vai chegar até ao P3, ativando os sinais pertinentes para obter o dado e armazenar-lho no *Data Register*. Quando temos o dado neste registo, o próprio sinal *Print_Db* vai gerir a saída do dado para que o *P3Loader* possa ler pela porta paralela. Depois, o programa vai apresentar o dado pelo ecrã. O sinal *Print_Db* vai fornecer tanto a leitura como o envio do dado ativando as saídas corretas dos multiplexadores para a execução do comando.

Write Com a inserção deste comando e os envios feitos, ativa-se a sinal *Write_Db* que chega até o P3 e gere o processador para realizar a escrita. Como no comando anterior, os 4 bits que indicam o registo alvo vão ser armazenados no *Number Register* que vai estar ligado a uma das entradas do multiplexador que tem a entrada *SelAD* do *Register Bank*. Por outro lado, o dado a inserir vai ficar armazenado no *Data Register* que também vai estar ligado a uma das entradas do multiplexador que gere a entrada *D* do mesmo Register Bank. Da mesma maneira que no comando *Print*, neste comando o sinal *Write_Db* vai fornecer a escrita do dado no registo do processador ativando as saídas corretas dos multiplexadores para a execução do comando.

Exit Quando o dispositivo decodifica o envio, o *Trap Flag Register* vai ficar a 0. Isto afeta no *Enabler* que não vai pôr nenhuma das suas saídas a 1 entretanto o *Trap Flag Register* esteja a 0, pelo que a decodificação fica parada e, consequentemente, o *Debugger* desativado. Este sinal também vai afetar ao *Reset* de todos os registos do *Debugger* devido a que quando ativar-se, todos os registos devem ficar a 0 para que não haja nenhum dado

em nenhum dos registos no caso que o utilizador querer voltar ativar o modo debug — os *Breakpoint Register* também vão ficar a 0 depois de inserir este comando.

4.3 Modificações ao D2-SB

Como consequência da inserção do dispositivo *Debugger*, as ligações ficam agora um bocado alteradas relativamente ao que eram. O *Debugger* vai precisar estar ligado principalmente ao P3 — para conseguir parar o processador e poder manipular os sinais de entrada do *Register Bank*. Também vai estar ligado aos dispositivos PIO5 —para a comunicação do debugger com a porta paralela e, consequentemente, com o utilizador — e MEM1 Interface — para a comunicação com a memória — para o seu correto funcionamento.

Para o utilizador poder usar o *Debugger*, deve iniciar o programa *P3Loader* no qual vai ter uma opção para iniciar o modo debug — *startD*. Uma vez iniciado, o programa vai ficar à espera de que o utilizador carregue o botão 15 do DIO5 que vai ser o botão de paragem. É por isto que também vamos ver como alguma das modificações afetam à saída correspondente de este botão da interface do DIO5 já que, no momento de carregar o botão, deve enviar-se uma mensagem pela porta paralela para que o programa *P3Loader* saiba que o *Debugger* parou o processador e o utilizador possa começar interagir com o dispositivo inserindo qualquer dos comandos que tem à sua disposição.

Na Figura 4.4 podemos ver como é que vão ligar-se os dispositivos depois da inserção do *Debugger*.

Pelo que agora a arquitectura do D2-SB vai ficar com todos os módulos mencionados antes mais a inserção de:

Debugger Módulo que, uma vez o P3 esteja no modo debug, vai controlar o modo como o P3 executa as instruções, o conteúdo dos seus registos e da sua memória assim como vai ter uma comunicação com o utilizador através da porta paralela.

Como vemos na Figura 4.4, a entrada do MEM1 Interface também vai ficar modificada. Isto deve-se a que quando o programa *P3Loader* quer escrever na memória, esta entrada deve ficar ligada à saída dos dados do PIO1.

Todas as entradas inseridas ao P3 vão ser precisas para controlar a execução das instruções no processador e arranjar os dados do *Register Bank*. As saídas novas devem-se à necessidade do *Debugger* de saber o conteúdo do *Program Counter* para os *breakpoints* ou os dados que se precisarem.

Os novos multiplexadores da entrada do PIO1 foram precisos devido aos três possíveis eventos nos quais o *Debugger* precisa enviar algum dado pela porta paralela. Estes três eventos são: o carregamento do botão 15, a execução do comando *Print* e a ativação da linha *BP_Active* — indica quando um *breakpoint* é encontrado. Assim, quando o botão 15 se ativar vai ser enviada

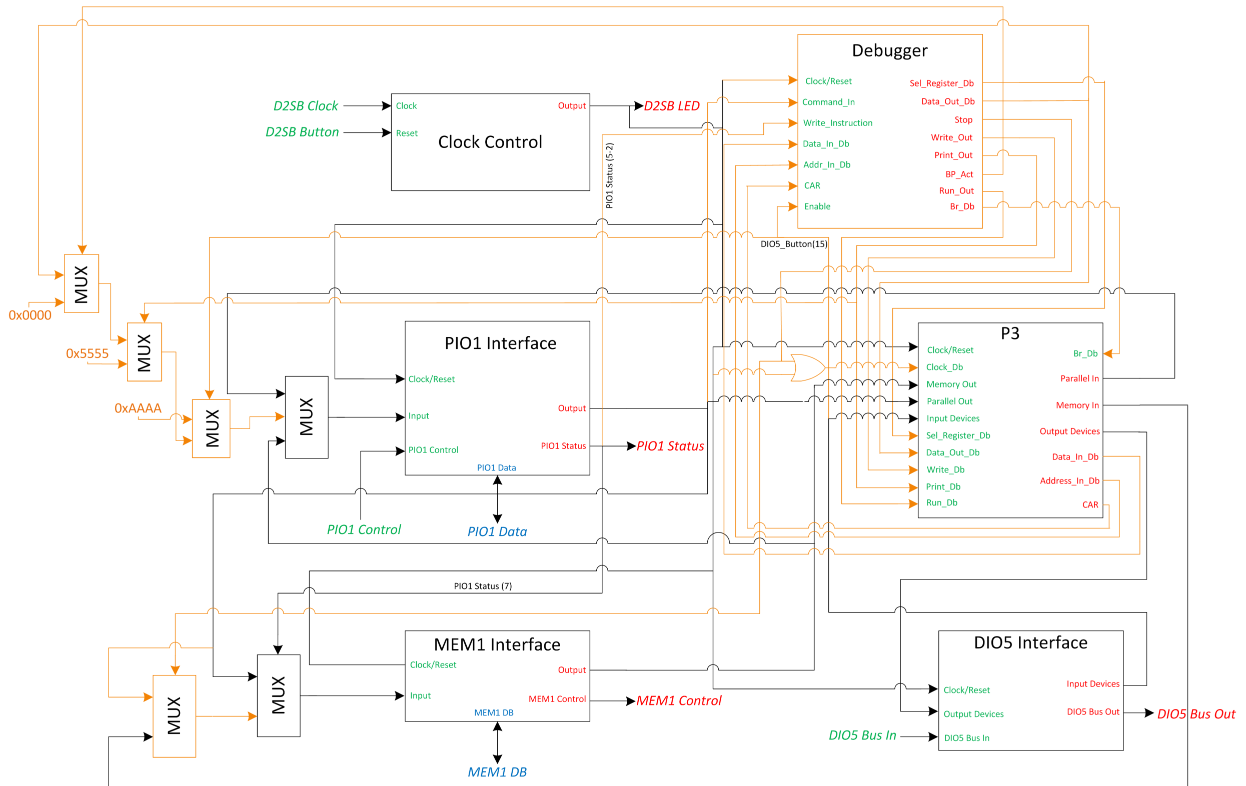


Figura 4.4: Arquitectura modificada do sistema.

a mensagem 0xAAAA pela porta paralela; 0x5555 quando o sinal *BP_Active* se ativar; e o dado extraído do P3 quando o comando *Print* seja inserido. Como podemos ver, todos estes multiplexadores vão ficar ligados em cascata até que, finalmente, o multiplexador do botão 15 fique ligado a um multiplexador que já estava na implementação antiga. Neste, o *Select* vai ser os três sinais que vêm do PIO1 e que vão ser os que selecionam o registo do PIO1 a ler ou a escrever pelo utilizador e que, no caso de querer algum dado do *Debugger* vai ser “101" — o PIO1 só tem 5 registos pelo que o último endereço vai ser “100" —, código que vai ativar a entrada dos novos multiplexadores.

Da mesma maneira, quando o comando *Print* é inserido pelo utilizador, é preciso que a entrada do MEM1 fique ligada à saída do P3. É por isso que inserimos esse novo multiplexador.

Para finalizar, também podemos ver como o P3 tem agora outra entrada de relógio que vai afetar vários registos que devem ficar parados quando o utilizador precisa parar o P3. Isto vai conseguir-se através de um sinal que vai sair do *Debugger*, o sinal *Stop*, que vai ficar a 1 sempre que o *Debugger* precisar parar o P3 e que vai fazer que a entrada *Clock_Db* do processador fique a 1 todo o tempo que o sinal *Stop* esteja ativo.

4.4 Modificações ao Processador P3

Como podemos ver na Figura 4.5, agora o P3 vai ter mais entradas e saídas que antes. Isto é principalmente devido aos sinais que vêm do *Debugger* para controlar a paragem do processador e a escrita e leitura dos registos — a escrita e leitura da memória vai ser implementada principalmente em software através das funções já implementadas no *P3Loader* e o hardware desenvolvido anteriormente — e aos registos que o debugger precisa para saber quando parar o processador.

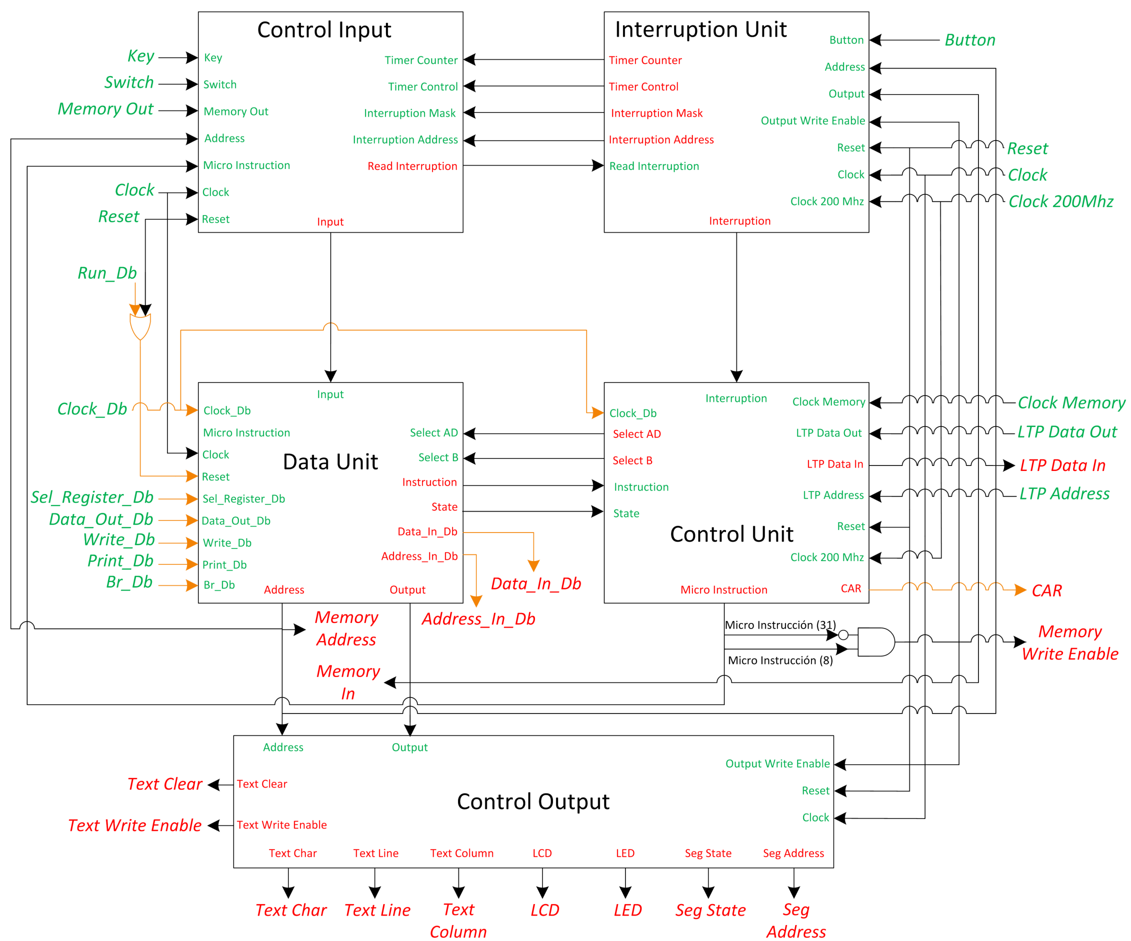


Figura 4.5: Arquitectura modificada do P3.

Também ficou modificada a entrada *Reset* da *Data Unit* devido a que quando o sinal *Run* é ativo, são reinicializados todos os registos de maneira que fique como ao início da execução.

De qualquer maneira, todas estas entradas e saídas, uma vez dentro do P3, vemos que vão chegar diretamente à *Data Unit*. Isto é devido a que é onde se encontram os dados que vamos precisar ou alterar. Na Figura 4.6 vemos como fica a *Data Unit* com as modificações necessárias. Vai ter os seguintes multiplexadores e saídas novas:

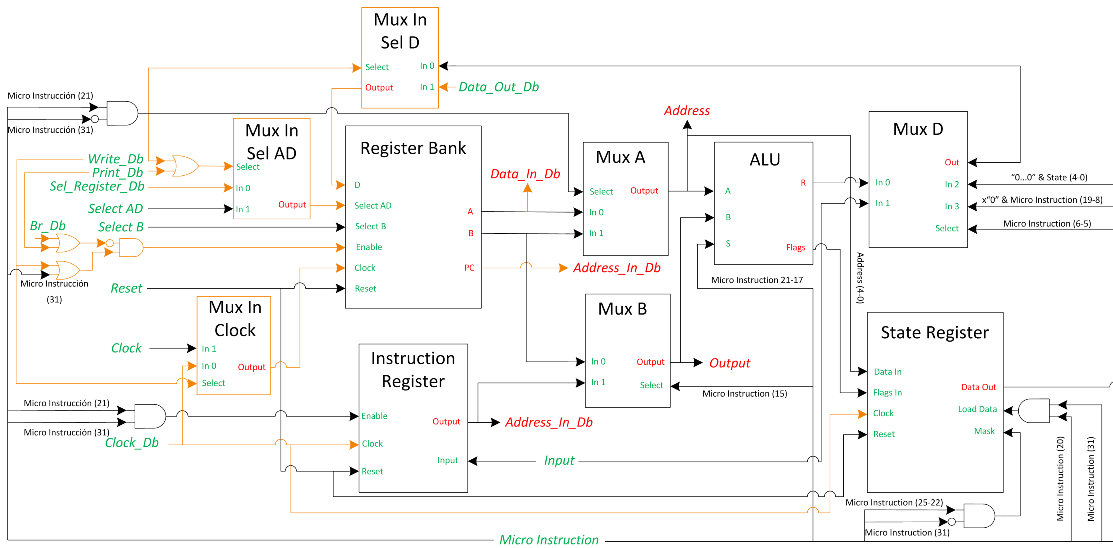


Figura 4.6: Arquitectura modificada do *Data Unit*.

Porta_In_Write Vai modificar o comportamento do sinal de escrita do *Register Bank*. Isto é devido a que já não só vamos ter que permitir a ativação do *Regiser Bank* quando o bit 31 do *Micro Instruction Register* seja 1, senão também quando o *Write_Db* seja 1 para poder escrever um dado em qualquer dos registos. Quando o *Debugger* para o P3 o sinal 31 do *Micro Instruction Register* pode ficar a 1 pelo que também devemos garantir que o sinal de escrita no *Register Bank* é 0 nos casos dos comandos *Br* e *Print* devido a que nestes casos escrevem-se dados no *Data Register* do *Debugger* — este registo vai estar diretamente ligado ao *Mux_In_D*.

Mux_In_SelAD Se os sinais *Print_Db* ou *Write_Db* são ativadas na entrada *SelAD* do *Register Bank* deve entrar a saída do *Number Register* do *Debugger* que vai ser o sinal *Sel_Register_Db*.

Mux_In_D Se o sinal *Write_Db* esteja ativa, pela entrada *D* do *Register Bank* deve entrar a saída do *Data Register* — *Data_Out_Db* — do *Debugger*, onde vai estar armazenado o novo valor do registo a modificar.

Mux_In_Clock O *Register Bank* também deve ficar parado mas quando o comando *Write* é ativado o seu relógio tem que correr para que este comando possa escrever o novo dado em algum dos registos. Pelo que quando o *Write* seja 0 o relógio do *Register Bank* vai ser *Clock_Db* e *Clock* se for 1 — entretanto esteja o processador parado pelo *Debugger*

Sinal A do *Register Bank* Sinal ligado à saída *Data_In_Db* para levar os dados dos registos que o *Debugger* precisar do P3.

Sinal PC do *Register Bank* É uma saída do *Register Bank* que está ligada à nova saída do processador *Address_In_Db*. Vai ser encarregada de levar até o *Debugger* o endereço no que o processador está em cada momento para poder comparar este endereço com os armazenados nos *Breakpoint Register*.

Também podemos ver como o *Control Unit* tem uma entrada e uma saída novas:

Clock_Db Usado para parar o relógio dos registos — CAR e SBR — quando o *Debugger* enviar uma sinal de paragem ao P3.

CAR Saída do registo CAR. É precisa para que o Debugger possa verificar os endereços da instrução em execução só uma vez e não em cada microinstrução da instrução.

5

Implementação e Resultados

Conteúdo

5.1 Metodologia de Desenvolvimento **42**
5.2 Resultados **42**
5.3 Observações **46**

5.1 Metodologia de Desenvolvimento

Ao início fizeram-se os desenhos com lápis e folhas mas quando já tinha a arquitectura mais o menos clara, foi descrita diretamente em código VHDL. Também no início foi feita uma primeira versão das modificações necessárias ao *P3Loader* em C++.

Para depurar o *P3Loader* e saber que estava a funcionar bem criou-se um *Dummy* em software para simular o comportamento dos envios pela porta paralela do P3. Também se programaram alguns dos comandos diretamente em software devido a que com a interface já feita não é necessário interagir com o P3 desde o *Debugger* para a realização de algumas destas instruções. Estas são: *List* — mostra os breakpoints inseridos até esse momento; *Code* — precisa interagir com o P3 mas com os métodos já feitos no *P3Loader* pode-se ver dados da memória sem necessidade de interagir com o *Debugger*; *Write* ou *Print* quando interagem com a memória — ainda que o *Write* alterar a memória, é sempre através do software já implementado e do hardware já desenvolvido; o *Print* interage da mesma maneira que o comando *Code* — e *help*.

Terminada a implementação em VHDL, iniciou-se o processo de depuração com o *ModelSim* mas, como foi dito antes, este programa tinha algumas limitações. Além disso, pôde-se simular os elementos em separado o que ajudou melhorar a implementação com a descoberta de bugs no hardware desenvolvido até o momento.

Uma vez depurados todos os elementos por separado, começou-se a depuração na placa. Através dos LEDs pude ver sinais internos para saber onde é que falhava o dispositivo. Para isto, esteve-se experimentando comando por comando até que estiveram a correr todos e assim conseguir o correto funcionamento do *Debugger*.

Nesta última fase também foram fixadas algumas falhas no *P3Loader* que ainda não foram descobertas como a correta sincronização do programa com o dispositivo *Debugger* que até o momento não estava a correr bem.

5.2 Resultados

O *Debugger* foi testado com o projecto que se fez na cadeira *Arquitectura de Computadores* no ano 2011-2012. Este é um projecto válido para testar o desenho porque é um projecto mais o menos grande no qual se usam muitas das funcionalidades da placa devido a que é um *Pacman* — Figura 5.1 — que interage com o utilizador através do ecrã e do teclado.

Como vemos na screenshot da figura 5.2, os comandos foram todos implementados mas alguns só em software devido a que foi usado para as suas implementações as funções que já estavam implementadas no *P3Loader* inicial e o hardware desenvolvido no D2-SB inicial. Estes comandos são *Help*, *List*, *Code* e os comandos *Write* e *Print* quando interagem com a memória. Os outros também têm implementação nova em hardware e comunicam-se com o *P3Loader* pela porta paralela.

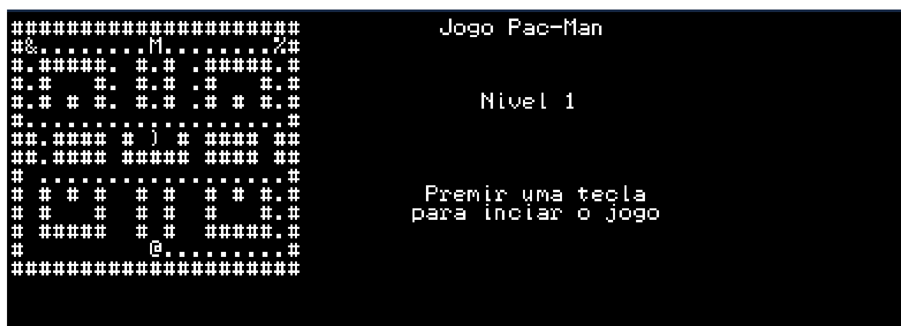


Figura 5.1: Projecto AC 2011-2012, *Pacman*.

```
P3 FPGA Controler.
Type "help" for more information.

>>startD
Wellcome to Debug Mode.
Type "help" for more information.

Waiting to STOP signal (Buttom 15) or Breakpoint signal.

P3 stopped by Debugger.
DB>>help

P3 FPGA Debug Controler commands:
        - help                              : Display all available debugger commands.
        - run                               : Run the P3 from the start.
        - cont                              : Continue the P3 execution.
        - step                              : With the P3 stopped, execute just an instruction.
        - list                              : Display all breakpoints inserted already.
        - code [address] [numPos]           : Display [numPos] positions of the extern memory of P3 since the [address].
        - br [address]                      : Insert a new breakpoint at [address].
        - del [address]                     : Delete the breakpoint in [address].
        - print [-r|-a] [RX|address]        : Display the content of number X register or [address].
                                              Options:
                                                      [-r] for registers.
                                                      [-a] for addresses.
        - write [-r|-a] [RX|address] [value] : Write in number X register or [address] [value].
                                              Options:
                                                      [-r] for registers.
                                                      [-a] for addresses.
        - exit                              : Terminate Debug Mode.

DB>>
```

Figura 5.2: P3Loader - Debugger: Operações disponíveis.

A Figura 5.3 apresenta o carregamento do programa e o funcionamento dos comandos *Print* e *Write*. Como vemos, o dado do endereço 8000h é modificado e apresentado mais uma vez.

Note-se que o *Debugger* parou o programa devido ao carregamento do utilizador no *Botão 15*.

Na seguinte Figura 5.4 vemos como estão a correr os comandos *Run*, *Br* e *Step*. Vemos como depois de inserir um breakpoint e pôr a correr o programa o *Debugger* pára a execução do programa no mesmo endereço do breakpoint que foi inserido. Também podemos ver como, depois de inserir o comando *Step*, o *Program Counter* — Registo 15 — aumenta em um valor.

A diferença para a Figura 5.3 é que nesta vemos que a paragem do *Debugger* é devida à chegada do *Program Counter* a um endereço que está inserido como breakpoint.

Por último, apresenta-se nas tabelas 5.1 e 5.2 uma comparação em relação às linhas de código VHDL e ao área com respeito ao dispositivo inicial. O contagem das linhas do *Debugger* foi feito de todos os ficheiros .vhd que foram precisos para o seu desenvolvimento completo.

Também, na Tabela 5.3 vemos o aumento de frequência máxima entre o dispositivo inicial e final.

```
P3 FPGA Controler.
Type "help" for more information.

>>load c:/projac.exe
Reading file "c:/projac.exe".
Loading P3 memory...

Address | Content
-----------------
    0000 | c020
    0001 | 0382
    0002 | 5001
    0003 | ae60
    0004 | 0001
    0005 | ac70
    0006 | 8023
    0007 | 5401
    0008 | 1c00
. . .
    9281 | 0020
    9282 | 0020
    9283 | 0020
    9284 | 0020
    9285 | 0000
    fe00 | 0009
    fe01 | 0010
    fe02 | 0017
    fe03 | 001e
    fe0f | 0002

Done!

P3 is now running.

>>startD
Wellcome to Debug Mode.
Type "help" for more information.

Waiting to STOP signal (Buttom 15) or Breakpoint signal.

P3 stopped by Debugger.

DB>>print -a 8000h
Loading P3 Extern Memory address: 0x8000

Address | Content
-----------------
 0x8000 | 0xa5a5

DB>>write -a 8000h 5a5ah
Writing in P3 Extern Memory address: 0x8000 Data: 0x5a5a
Writing succesfull

DB>>print -a 8000h
Loading P3 Extern Memory address: 0x8000

Address | Content
-----------------
 0x8000 | 0x5a5a

DB>>_
```

Figura 5.3: P3Loader - Debugger: Funcionamento dos comandos *Print* e *Write*.

	Inicial	Final	Aumento
D2-SB	309	450	45,6%
P3	210	251	19,5%
Data Unit	149	191	28,2%
Control Unit	146	148	1,4%
Debugger	0	811	-

Tabela 5.1: Comparativa do número de linhas inicial e final.

	Inicial	Final	Aumento
D2-SB	1.004	1.130	12,5%

Tabela 5.2: Comparativa do número de *Slices* ocupados inicial e final.

```
P3 FPGA Controler.
Type "help" for more information.

>>load c:/projac.exe
Reading file "c:/projac.exe".
Loading P3 memory...

Address ¦ Content
-----------------
    0000 ¦ c020
    0001 ¦ 0382
    0002 ¦ 5001
    0003 ¦ ae60
    0004 ¦ 0001
    0005 ¦ ac70
    0006 ¦ 8023
    0007 ¦ 5401
    0008 ¦ 1c00
                              . . .
    9282 ¦ 0020
    9283 ¦ 0020
    9284 ¦ 0020
    9285 ¦ 0000
    fe00 ¦ 0009
    fe01 ¦ 0010
    fe02 ¦ 0017
    fe03 ¦ 001e
    fe0f ¦ 0002

Done!

P3 is now running.

>>startD
Wellcome to Debug Mode.
Type "help" for more information.

Waiting to STOP signal (Buttom 15) or Breakpoint signal.

P3 stopped by Debugger.

DB>>br bbcch

DB>>run
Waiting to STOP signal (Buttom 15) or Breakpoint signal.

P3 stopped by a Breakpoint.

DB>>print -r R15
Loading P3 Register 15

Register: 15, Value: 0xbbcc

DB>>step

DB>>print -r R15
Loading P3 Register 15

Register: 15, Value: 0xbbcd

DB>>_
```

Figura 5.4: P3Loader - Debugger: Funcionamento dos comandos *Br* e *Step*.

	Inicial	Final	Aumento
D2-SB	36,76 MHz	34,48 MHz	-6,61%

Tabela 5.3: Comparativa da frequência máxima inicial e final.

5.3 Observações

Além do processo de depuração, uma das maiores complexidades do projecto foi a utilização da porta paralela como meio de comunicação. A escolha desta maneira de comunicar o dispositivo com o utilizador foi feita em 2003, ano em que ainda a porta paralela tinha muita aceitação no mercado mas não se viu a possibilidade da substituição deste meio por outros mais modernos como o USB.

O *ModelSim* e as suas simulações ajudaram muito na fase de depuração mas quando tive que simular todo o projecto completo não deu e comecei depurar com os dispositivos da placa, principalmente com os LED, os quais foram ligados a distintos sinais para ver o que é que acontecia nas diferentes entradas e saídas tanto das portas como dos registros e em outros dispositivos.

Inicialmente, o sinal *Stop* devia vir simplesmente da saída da combinatória do armazenamento dos *Breakpoint Register* do *Debugger*, mas este sinal também tinha o alvo de parar o P3 pelo que em alguns dos elementos do processador agora entra um relógio diferente, resultado duma porta OR entre o antigo relógio e este sinal. Por isto é necessário que o sinal *Stop* seja um sinal limpo, sem picos, e optou-se por um biestável para marcar bem quando o sinal estiver ativo.

Devido à grande conectividade dos elementos, foi difícil no processo de depuração atingir bem, principalmente, os sinais de *Enable* dos registros e biestáveis pelo que foi preciso pôr alguns multiplexadores nas entradas.

No processo de pensar como é que ia ser a melhor maneira de parar o P3 pensaram-se três possíveis soluções: pôr um registo intermediário entre a memória e o *Instruction Register* para controlar o passo das instruções a este registo; controlar os dados que saem das memórias ROM internas do P3; e parar o relógio através de uma sinal *Stop* que entrar numa porta OR com o antigo relógio. Finalmente, foi escolhida a última opção devido, principalmente, à facilidade da sua implementação em combinação com a eficácia do sistema.

O *Opcode* de cada comando com implementação em hardware é de 6 bits pelo que pode haver até um máximo de 64 combinações das quais só foram utilizadas 8 para os comandos implementados. É por isto que ainda se pode acrescentar comandos com outras opções não pensadas no contexto do desenvolvimento de esta tese.

6

Conclusões

No final da tese, as funcionalidades expostas foram totalmente desenvolvidas e tanto o dispositivo *Debugger* como o programa *P3Loader* modificado ficaram a correr corretamente.

No início pensou-se em que o carregamento do botão 15 ativar uma interrupção no próprio processador, e que for o mesmo P3 quem se encarregar da sua própria paragem. Esta maneira talvez fosse a mais correta mas optou-se pela implementação desenvolvida devido a que o comportamento ia ser igual e que desta maneira ia ser muito menos complexa a sua implementação.

Como trabalho futuro sempre fica a opção de acrescentar o número de comandos para conseguir desenvolver mais funcionalidades das implementadas. Além disso, também se pode trocar a placa PIO1 por outra que forneça ao dispositivo de um meio de comunicação mais rápido e moderno. Como sugestão, a placa *Digilent PmodUSB2 Module* [16] — fornece de uma ligação USB 2.0 — pode desenvolver estas funcionalidades mas teria ser modificada a implementação atual do dispositivo.

Apesar de ter algumas limitações — o número máximo de breakpoints, por exemplo —, os objectivos do trabalho foram cumpridos e o *Debugger* permete efectuar a depuração do programa que esteja a correr no P3.

Bibliografia

[1] V. Brito, “Extensão do ambiente hardware do processador p3,” Grau de Mestre em Engenharia Electrotécnica e de Computadores, IST, 2011.

[2] S. S. e. a. Richard Stallman, Roland Pesch, Debugging with gdb: The gnu Source-Level Debugger, 10th ed. Free Software Foundation, 2012.

[3] O. Yuschuk, “OllyDbg Website,” http://www.ollydbg.de/, 2012, [Online; accessed 10-October-2012].

[4] Wikipedia, “SoftICE Wikipedia Website,” http://en.wikipedia.org/wiki/SoftICE, 2012, [Online; accessed 10-October-2012].

[5] ——, “MacsBug Wikipedia Website,” http://en.wikipedia.org/wiki/MacsBug, 2012, [Online; accessed 10-October-2012].

[6] G. Arroz, J. Monteiro, and A. Oliveira, Arquitectura de Computadores: dos Sistemas Digitais aos Microprocessadores, 1st ed. IST Press, 2007.

[7] Digilent D2-SB System Board Reference Manual, Digilent, 2003.

[8] Spartan-IIE FPGA Family Datasheet, Xilinx, 2008.

[9] Digilent DIO5 Peripheral Board Reference Manual, Digilent, 2003.

[10] Digilent Parallel I/O 1 Board Reference Manual, Digilent, 2004.

[11] IEEE Standard Signaling Method for a Bidirectional Parallel Peripheral Interface for Personal Computers, IEEE Std, 2000.

[12] Digilent Memory Module 1 Reference Manual, Digilent, 2005.

[13] IS61LV5128AL, ISSI, 2005.

[14] FLASH MEMORY MT28F004B3, MT28F400B3, Micron, 2003.

[15] ISE 10.1 Quick Start Tutorial, Xilinx, 2008.

[16] Digilent PmodUSB2 Module Reference Manual, Digilent, 2005.