Suporte hardware para um debugger para o Processador pedagógico P3

Juan José Rebollo Barranco E-mail: juarebbar@gmail.com

Abstract—Um Pequeno Processador Pedagógico — P3 — foi desenvolvido por professores do IST. 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 do presente trabalho 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.

Index Terms—FPGA, P3, Debugger, VHDL, Xilinx, Trap Flag.

1 INTRODUÇÃO

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

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.

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 o programa P3Loader que até agora só carregava o código assembly no P3 ou dados nas memórias que o processador tem. Este programa 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.

Este artigo encontra-se organizado em 6 seções de forma que na Seção 2 é realizada uma revisão do estado da arte para pôr em situação dos debuggers que há no mercado atualmente. Mais adiante, na Seção 3 vemos uma série de elementos que precisamos para o desenvolvimento do trabalho — o processador P3, o hardware tecnológico usado, a versão inicial do desenho hardware do P3 e as ferramentas software. As funcionalidades e a arquitectura do debugger desenvolvido vo ser vistas na Seção 4 para, na Seção 5, apresentar os resultados obtidos e vermos algumas tabelas comparativas. Por último, na Seção 6, vamos ver as conclusões e o trabalho futuro do trabalho.

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 Trabalho Relacionado

Para a implementação do debugger, tivemos que aprender como é que outros debbugers funcionam.

Um dos debuggers mais conhecidos do mercado é o *GDB*. Oferece uma grande funcionalidade enquanto a paragens e visualização de posições de memória e registos. Tem uma licença totalmente livre e pode executar programas escritos em *Ada*, *C*, *C++*, *Objective-C* e *Pascal* de maneira nativa ou em remota.

Outro debugger que investigamos foi o *OllyDbg*. Este 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 está disponível o código fonte do programa. É frequentemente usado para fazer engenharia reversa e não precisa de instalação.

O terceiro dos debuggers vistos é o *SoftICE*. 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. Excepcionalmente útil na programação de drivers e é compatível com as últimas versões do sistema operativo de *Microsoft*.

Por último, temos o debugger *MacsBug*. Este é um depurador de baixo nível — linguagem assembly/nível máquina — para o sistema operativo de *Macintosh* e oferece muitos comandos para desmontagem, procura e visualização de dados.

Na Tabela 1 podemos ver um resumo/comparativa das características dos debuggers analizados nesta seção.

3 Hardware e Ferramentas de Desenvolvimento

3.1 O Processador P3

Como o P3 [2] 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*.

3.2 Hardware

Para o objectivo do projecto, vamos usar umas placas interligadas. O desenho foi feito pelos professores do IST e no seguinte vamos ver a função de cada uma das placas.

A placa Digilent D2-SB System Board [3] é o elemento central. Contém uma FPGA *Xilinx Spartan2E XC2S200E PQ208* [4] 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.

A Digilent DIO5 Peripheral Board [5] contém uma série de 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.

Com a Digilent PIO1 Parallel I/O Board [6] podemos comunicar a placa com outros dispositivos usando uma porta paralela via SPP ou EPP [7].

Por último, a placa Digilent MEM1 Memory Module 1 [8] 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 [9] — e uma memória flash *28F004B3* — também vai ter uma capacidade de 512KB; 512K endereços com 1 byte por endereço [10].

3.3 Arquitectura original do sistema

No esquema da Figura 1 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 o dispositivo principal vai ser o *P3*. Aquí estão o processador, o controlador de entradas/saídas e o gestor de interrupções.

Também está o dispositivo *Clock Control* que é o responsável por gerar os diversos sinais de relógio necessários e os sinais de inicialização.

Por último, vemos que temos os dispositivos interfaces: o *MEM1 Interface* — possibilita a leitura/escrita na memória externa —, o *PIO1 Interface* — possibilita a leitura/escrita de dados através da porta paralela — e o *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.

Nesta arquitectura, além de inserir o dispositivo *Debugger*, vamos ter que fazer algumas

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

TABLE 1: Resumo/Comparativa das características dos debuggers analizados.

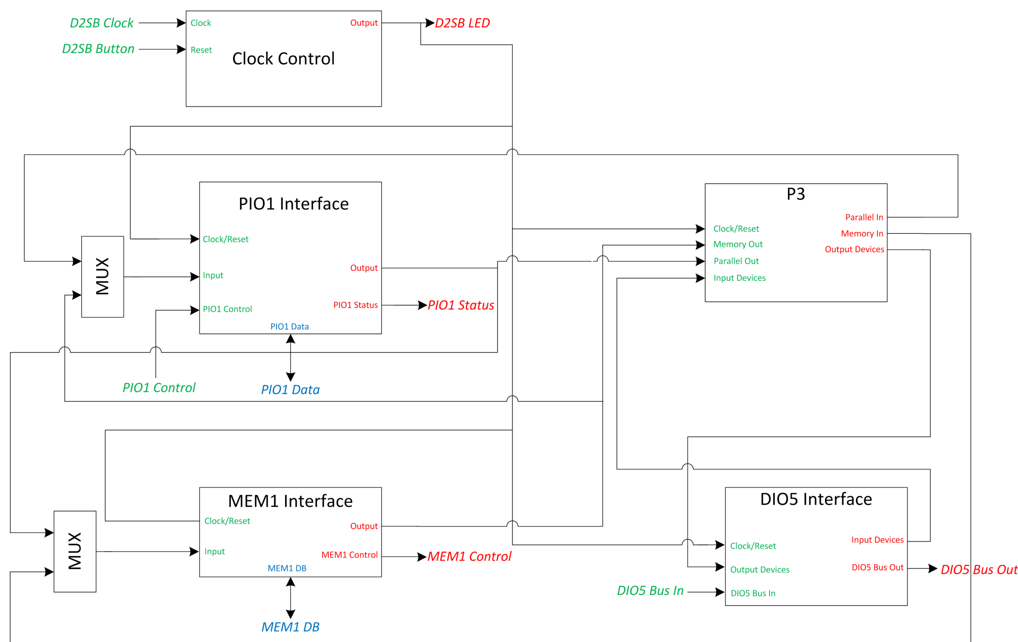

Fig. 1: Arquitectura original do sistema.

modificações tanto nas ligações entre os dispositivos como dentro do próprio P3 e alguns dos seus componentes.

3.4 Ferramentas de Desenvolvimento

Tanto para o desenvolvimento hardware do desenho como para a modificação do programa de carregamento das memórias do P3 — o *P3Loader* — foi preciso a utilização de diversos programas.

O software principal foi o *Xilinx ISE 10.1 [11]* que está constituído por diversas ferramentas de desenvolvimento de sistemas digitais incluído o desenvolvimento e depuração do código VHDL. Também foi importante o *ModelSim PE Student Edition 10.1c* que foi preciso para a simulação das partes separadas do desenho do trabalho.

Além destes programas, também foi preciso usar um compilador para a modificação do *P3Loader*. Qualquer compilador C++ podia ser válido pelo que foi escolhido o *Microsoft Visual C++ 2008 Express Edition* devido, além da alta funcionalidade que tem, que neste compilador realizou-se as primeiras versões do programa *P3Loader*, pelo que já sabemos que corria bem.

4 Implementação

Para o desenvolvimento do projecto foi preciso a inserção de algumas funcionalidades novas. Estas vão ser implementadas tanto em software — para interagir com o utilizador, através da inserção de comandos no programa *P3Loader* — como em hardware.

Os comandos desenvolvidos são:

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.

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.

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.

List
: Mostra no ecrã todos os *breakpoints* armazenados neste momento.

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.

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.

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

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.

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.

Exit
: Este comando vai sair do modo debug e vai fazer que o *P3Loader* tenha o comportamento anterior à modificação.

Para desenvolver estas funcinalidades foi desenhado o debugger hardware que vamos ver a seguir. A explicação do mesmo vai ser feita em três partes separadas para que seja mais fácil a sua compreensão.

Para começar vamos ver a parte da implementação do hardware na qual estão os registos auxiliares. Podemor ver na figura 2 como os componentes principais são três registos de só um bit: o *Trap Flag Register* — vai ser 1 quando o modo debug esteja ativo e 0 quando não —, o *Stop Register* — vai ser 1 quando o *Debugger* precisar parar o P3 e 0 quando não — e o *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 e vai ficar assim até chegar um sinal que puder desativar-lho.

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

A seguinte parte que vamos ver 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 3, o armazenamento de cada *breakpoint* vai ter associado os seguintes componentes: o *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 —, o *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*

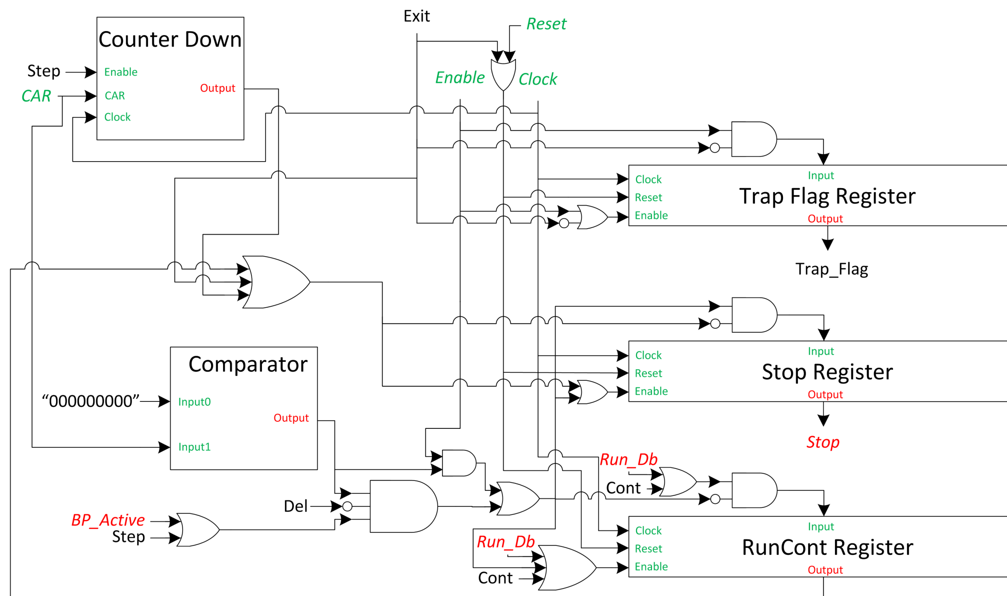

Fig. 2: Hardware do ligado e desligado do *Debugger*.

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 —, o *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 — e o *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.

Por último vamos ver o desenho da parte destinada ao armazenamento e decodificação dos comandos do dispositivo *Debugger*.

Na figura 4 podemos ver o desenho desta parte do projecto e os seus componentes: o *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* —, o *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 e é por isto que 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* —, o *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 —, o *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 —, o *Data Register* — é um registo de 16 bits que pode armazenar tanto valor que o *Debugger* quer escrever num registo do P3 como o valor lido de um registo do processador antes de enviar-lho pela porta paralela — e o *Mux Data Register* — controla a entrada de dados no *Data Register* que pode ser a saída do *Command Register 0* ou *Data_In_Db*.

Para a realização de isto, o primeiro que se fez foram os desenhos com lápis e folhas mas quando já se 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

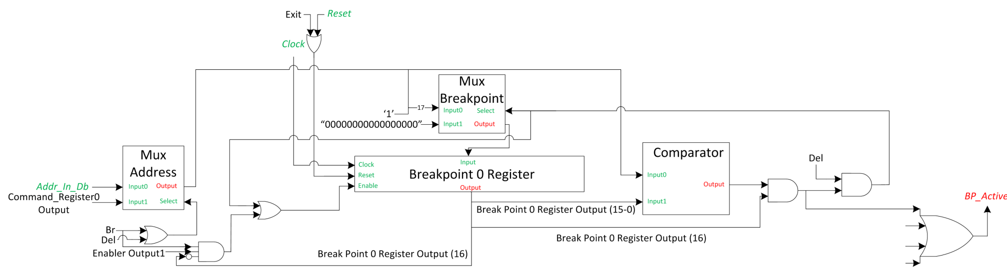

Fig. 3: Hardware do armazenamento de um *Breakpoint* do *Debugger*.

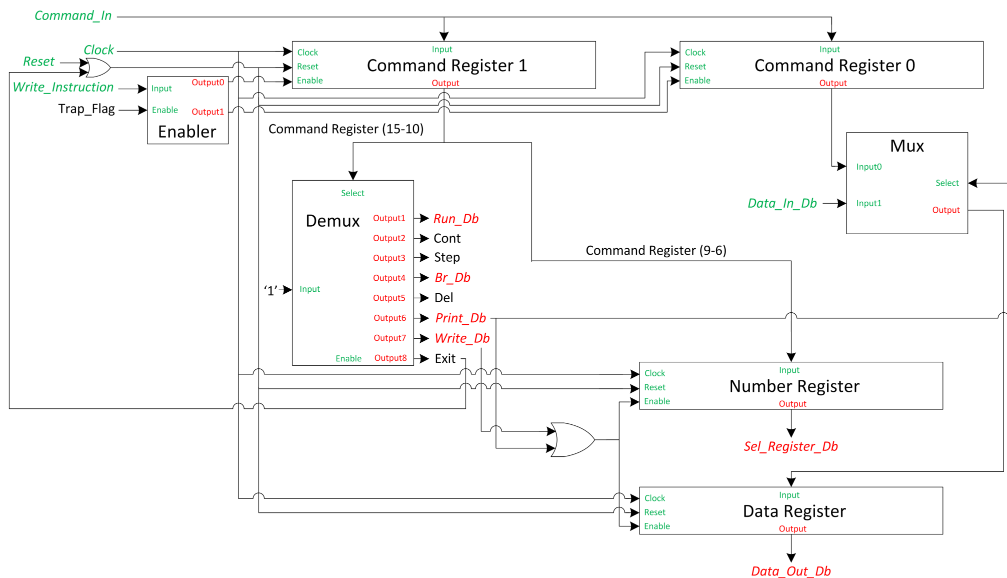

Fig. 4: Hardware do armazenamento e decodificação dos comandos do *Debugger*.

em software devido a que com a interface já feita não é necessário interagir com o P3 desde o *Debugger* para a realização de 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 este programa tinha algumas limitações e não foi possível simular o desenho completo — ainda que sim separadamente. 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 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 — que interage com o utilizador através do ecrã e do teclado.

Como vemos na screenshot da Figura 6, 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.

Por último, apresenta-se nas tabelas 2 e 3 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 4 vemos o aumento de frequência máxima entre o dispositivo inicial e final.

	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	-

TABLE 2: Comparativa do número de linhas inicial e final.

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

TABLE 3: Comparativa do número de *Slices* ocupados inicial e final.

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

TABLE 4: Comparativa da frequência máxima inicial e final.

6 CONCLUSÇÕES E TRABALHO FUTURO

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

Como podemos ver na Tabela 2, o aumento em alguns partes do desenho anterior foi importante, além do ficheiro *Debugger* que foi desenvolvido completamente. Vemos que a parte que mais aumentou foi a do *DS-2B*, isto foi devido a que é aí onde inserimos o dispositivo *Debugger*.

Na Tabela 3 vemos que o aumento enquando aos *Slices* ocupados no foi demais. É por isto que ainda fica muita FPGA livre.

Por último, vemos na Tabela 4 que o aumento da frequência foi negativo. Isto foi principalmente devido à inserção da nova lógica no dispositivo.

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* [12] — 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.

REFERENCES

[1] V. Brito: *Extensão do Ambiente Hardware do Processador P3*, Grau de Mestre em Engenharia Electrotécnica e de Computadores, IST. 2011.

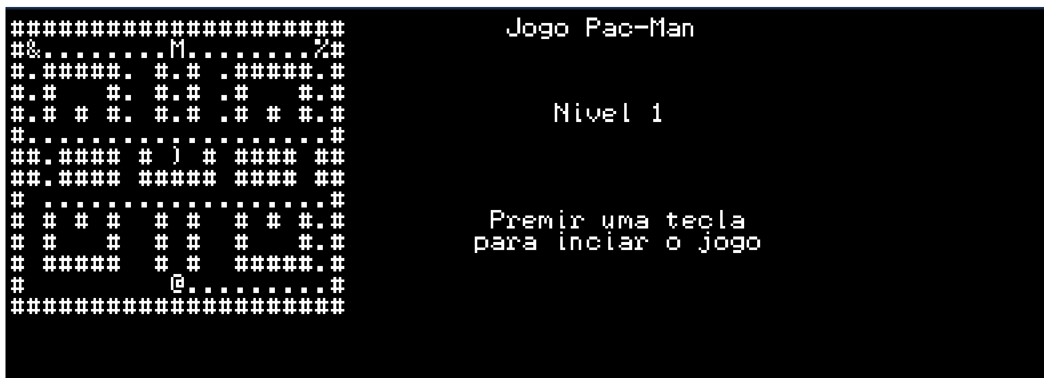

Fig. 5: Projecto AC 2011-2012, *Pacman*.

```
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>>
```

Fig. 6: P3Loader - Debugger: Operações disponíveis.

[2] Guilherme Arroz, José Monteiro and Arlindo Oliveira: *Arquitectura de Computadores: dos Sistemas Digitais aos Microprocessadores*, 1st ed. IST Press, 2007.

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

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

[5] *Digilent DIO5 Peripheral Board Reference Manual*, Digilent, 2003.

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

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

[8] *Digilent Memory Module 1 Reference Manual*, Digilent, 2005.

[9] *IS61LV5128AL*, ISSI, 2005.

[10] *FLASH MEMORY MT28F004B3, MT28F400B3*, Micron, 2003.

[11] *ISE 10.1 Quick Start Tutorial*, Xilinx, 2008.

[12] *Digilent PmodUSB2 Module Reference Manual*, Digilent, 2005.