Especificação, síntese e implementação em VHDL de um processador MIPS Single Cycle Simplificado

Bernardo Silva

***Resumo* – Este artigo descreve a implementação de circuitos reconfiguráveis que simulem um sub-conjunto da arquitectura MIPS RISC Single Cycle. O processador MIPS pode ser descomposto em cinco fases funcionais: *Instruction Fetch*, *Instruction Decode*, *Execution*, *Data Memory* e *Write Back*. A unidade de controlo opera sobre todas estas fases gerindo as operações a executar em cada uma delas. Todos os componentes constituintes desta arquitectura foram especificados em VHDL, linguagem de descrição de hardware, o que proporciona fazer o paralelismo entre descrição comportamental de hardware e implementação de circuitos digitais. Foram criados cenários de simulação de modo a efectuar a análise da funcionalidade, tempos de execução e desempenho da arquitectura implementada. Futuramente este projecto terá uma interface gráfica que permitirá uma visualização em tempo real dos valores dos sinais que constituem a arquitectura do processador desenvolvido. O projecto em desenvolvimento poderá ser usado no âmbito das disciplinas Computação Reconfigurável (4º ano de MIECT), Sistemas Digitais Reconfiguráveis (opção de 5º ano, MIEET) e Modelação e Síntese de Processadores (opção de 5º ano, MIECT/MIEET).**

***Abstract* – This paper describes an implementation of reconfigurable circuits which emulate an instruction subset of a simplified MIPS RISC Single Cycle processor. The MIPS processor can be decomposed in five functional stages: Instruction Fetch, Instruction Decode, Execution, Data Memory, and Write Back. The Control Unit operates in all of these stages, managing the way each operation should be executed. All the components of the architecture were specified using VHDL, allowing to establish the parallelism between behavioral hardware description and circuit implementation. Different simulation scenarios were created to analyze the functionality of the designed system, execution times and performance. In the near future, a graphical interface is going to be developed, making it possible to visualize the values of the processor's signals in real time. The designed project can be successfully employed within Reconfigurable Computing (4th year of Computer Engineering curriculum), Reconfigurable Digital Systems (5th year, Electrical Engineering curriculum) and Processor Synthesis and Modeling (5th year, Computer/Electrical engineering curriculum) disciplines.**

I. INTRODUÇÃO

Ao longo dos anos, tem-se vindo a verificar um grande aumento de desempenho e integração nos componentes lógicos em FPGA permitindo implementar sistemas lógicos complexos num único *chip* [1]. O desenvolvimento de sistemas exigentes e complexos pode implicar a implementação de circuitos com milhões de portas lógicas que incluem memórias, interfaces rápidas e outros componentes de alto desempenho. Uma das abordagens para o projecto e implementação deste nível de complexidade é a utilização de linguagens de descrição de hardware, por exemplo VHDL (Very high speed integrated circuit Hardware Description Language). Esta permite a um projectista de sistemas desenvolver hardware de forma simples e rápida, com a grande vantagem de ser possível efectuar simulações e múltiplas implementações o que torna as fases de desenvolvimento mais rápidas e diminuem os custos que uma produção física de várias evoluções do sistema poderia ter até atingir o produto final [2].

A filosofia RISC (Reduced Instruction Set Computer) teve inícios nos anos setenta com a IBM e desde então foi desenvolvida e documentada em universidades e companhias de microprocessadores. O intuito desta filosofia é de optimizar e obter o máximo de desempenho de um processador e ao mesmo tempo simplificar o hardware de modo a conseguir custos razoáveis de produção [3]. Na arquitectura RISC o conjunto de instruções é baseado em abordagens *load/store*. Apenas o conjunto de instruções *load* e *store* permite o acesso à memória. Nenhuma das outras operações aritméticas ou de I/O interagem com a memória. A simplificação neste tipo de arquitectura resulta numa rápida descodificação de instruções e facilidade de implementação. Devido à sua robustez de desempenho, formatos simples de instruções, diversidade e suporte de produtos, o processador MIPS RISC, é neste momento um dos líderes de mercado [4].

Neste artigo, descreve-se uma abordagem de implementação do processador MIPS Single Cycle usando a linguagem de descrição de hardware VHDL. O objectivo deste trabalho é a demonstração de como VHDL é fiável e permite a implementação rápida de protótipos de microprocessadores. O uso de VHDL acresce de variadas vantagens para este estudo, nomeadamente como a possibilidade de implementação a vários níveis de

especificação e encadeamento das estruturas necessárias (arquitectura, registos, blocos de memória, barramentos, portas lógicas) durante todo o processo de desenvolvimento [5].

II. PRINCÍPIOS DE FUNCIONAMENTO E IMPLEMENTAÇÃO

A arquitectura do processador MIPS é baseada nos três tipos de instruções de 32 bits cada: Tipo-R, Tipo-J e Tipo-I, fazendo com que a arquitectura desenvolvida possua um *datapath* de 32 bits consistente com a dimensão das instruções (Tabela 1). No desenvolvimento deste projecto não foi implementado o Tipo-J, no entanto facilmente é acrescentada essa possibilidade.

Tipo-R						
Campo	Opcode	Rs	Rt	Rd	Shamt	Funct
Bits	31-26	25-21	20-16	15-11	10-6	5-0
Tipo-I						
Campo	Opcode	Rs	Rt	Address		
Bits	31-26	25-21	20-16	15-0		

Tabela 1. Formato dos tipos de instruções implementadas

A implementação de um processador deste tipo implica que num ciclo de relógio sejam efectuadas as fases de busca da instrução, a sua descodificação, execução ou acesso à memória e armazenamento dos dados produzidos. No entanto, como se pode verificar ao construir esta arquitectura, não é de todo possível num único ciclo de relógio efectuar todas as sincronizações necessárias. Deste modo o ciclo de relógio é dividido por dois, podendo assim usufruir de 4 flancos de sincronização distribuídos pelos vários componentes da arquitectura que os necessitam. Foi atribuído o nome de *CPUClock* ao sinal de relógio que por cada ciclo, efectua uma instrução e *MemClock* ao sinal de relógio que possui o dobro da frequência de *CPUClock*, sendo este associado às leituras e escritas das memórias (Figura 1).

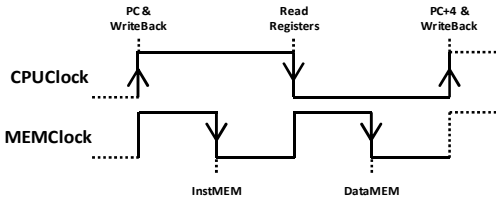


Figura 1. Diagrama representativo dos acessos efectuados nos flancos de relógio de CPUClock e MEMClock

A. Fase Instruction Fetch

A primeira fase da execução no MIPS RISC é a leitura da memória da instrução que se pretende executar (IF – *Intruction Fetch*). Esta operação de leitura é iniciada quando o *Program Counter* (PC), registo de 32 bits libera um endereço ao bloco *Instruction Memory* no flanco ascendente do *CPUClock*. O PC é incrementado de modo a indicar o próximo endereço de acesso à memória para a instrução seguinte (*PC+4*). No bloco *Instruction Memory*, lê-se da memória a instrução apontada pelo endereço recebido. Esta operação de leitura de memória é sincronizada com o flanco descendente de *MEMClock* e apenas efectuada quando *CPUClock* possui o valor '1' (ver transição *InstMEM* na Figura 1).

No caso de instruções de salto (Branch), o endereço da instrução seguinte poderá ser substituído pelo endereço gerado pela situação de salto. Para que tal aconteça com sucesso, é necessária lógica adicional que calcule o endereço de salto. Um multiplexer de duas entradas de 32 bits selecciona qual das suas entradas deve ser colocada à saída através do resultado gerado por uma porta lógica AND que avalia se a condição de salto é satisfeita.

A memória de instruções foi implementada com base nos blocos de memória embutida BlockRAM, disponíveis em FPGAs da família Spartan-3 [6]. Esta memória foi inicializada já com instruções que possibilitem posteriormente a análise dos resultados obtidos em simulação e detecção de erros [2,7,8].

Um diagrama de bloco desta fase é visualizado na Figura 2.

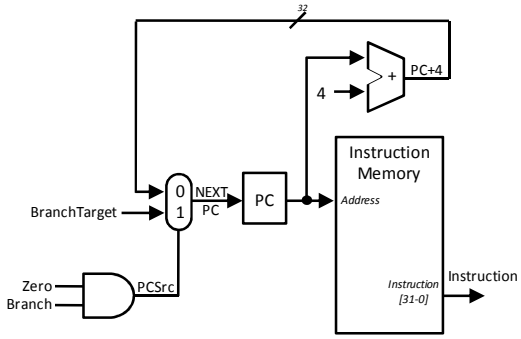


Figura 2. Diagrama da fase de Intruction Fetch

B. Instruction Decode

Quando a instrução é lida e colocada no barramento de saída do bloco *Instruction Memory*, os vários campos que constituem a instrução são processados. O *Opcode* da instrução é lido pela unidade de controlo do *datapath* e o campo *Funct* lido pela unidade de controlo da *ALU* (Arithmetic Logical Unit). Nesta fase a unidade de controlo (*Control Unit*) após ler o *Opcode* da instrução dispõe à sua saída todos os sinais de controlo do datapath (descrito mais detalhadamente à frente).

Os campos da instrução correspondentes aos registos que se pretendem ler/escrever são lidos pelo módulo *File Register*. Na implementação efectuada, as leituras de registos são assíncronas (podendo ser efectuadas de forma síncrona, ver transição *ReadRegisters* na Figura 1) e a escrita de dados no registo de destino é síncrona com o

flanco ascendente de *CPUClock* e apenas executada quando o sinal *RegWrite* se encontra activo.

Os campos lidos, cada um com 5 bits, endereçam registos de 32 bits que são lidos e colocados na saída do módulo. Existem 32 registos [r0-r31], todos disponíveis para guardar valores gerados pelas instruções à excepção do registo r0 que possui sempre o valor zero, norma implementada na arquitectura MIPS, visto que é um valor muitas vezes usado durante a execução de instruções.

O módulo *File Register* possui então quatro entradas: correspondentes aos endereços de dois registos de leitura e um de escrita, um porto de entrada de 32 bits dos dados a escrever e duas saídas de valores contidos nos dois registos lidos: *RegData1* e *RegData2*, ambos de 32 bits. Este módulo foi desenvolvido em VHDL de modo a que após sua síntese, os registos são implementados em LUTs (*LookUp Tables*) [3,8].

Os 16 bits menos significativos da instrução passam pelo *SignExtended*, módulo que replica o bit mais significativo de forma a gerar uma saída de 32 bits em complemento para 2. Este valor, servirá posteriormente para cálculo de endereços de salto, acesso à memória para leitura/escrita ou leitura de valores imediatos em instruções contendo valores imediatos.

O diagrama de blocos da figura 3, correspondente a esta fase, representa o descrito anteriormente.

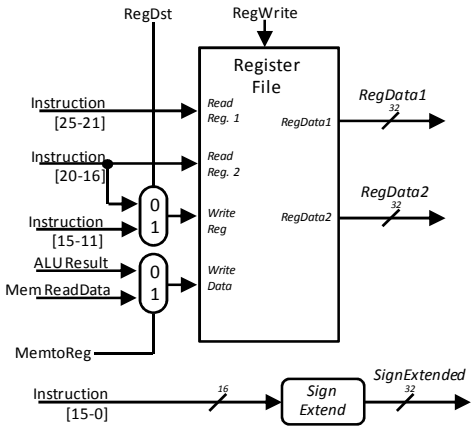


Figura 3. Diagrama de blocos da fase de *Intruction Decode*

C. Execution

Após a descodificação da instrução na fase anterior, dá-se início à fase *Execution* que produz todos os cálculos necessários. Os componentes nesta fase são a *ALU*, *ALU Control* e os blocos necessários para cálculo do endereço de salto: um somador de 32 bits e bloco combinatório de deslocamento *SLL2*. Na unidade de *ALU* efectuam-se os deslocamentos, cálculos aritméticos e lógicos e utiliza-se o somador de 32 bits para determinar o endereço relativo de salto duma instrução *branch*. Para efectuar o cálculo do endereço é necessário antes efectuar um deslocamento lógico à esquerda de dois bits do sinal *SignExtended* e depois usar esse resultado e o valor de *PC+4* como entradas do somador de 32 bits. O diagrama de blocos da figura 4 demonstra a arquitectura desta fase.

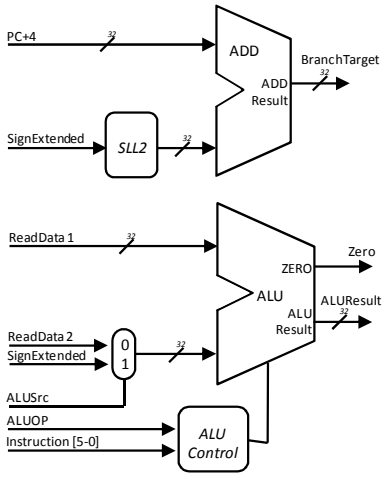


Figura 4. Diagrama de blocos da fase *Execution*

A ALU possui 3 entradas, sendo duas correspondentes aos valores dos registos no caso de instruções do Tipo-R ou o valor do primeiro registo e os 32 bits do *SignExtended* para instruções do Tipo-I. A selecção da segunda entrada é efectuada através de um multiplexer controlado pela unidade de controlo. A terceira entrada é correspondente aos bits de controlo. Estes bits são gerados pela *ALU Control* que analisa os 6 bits do campo *Funct* das instruções e os 2 bits gerados pelo *Control Unit*, decifrando assim que tipo de operação se pretende efectuar na *ALU*. A saída da *ALU Control* é um sinal de 3 bits que permitem gerar uma das seguintes combinações mostradas na Tabela 2.

Instrução	Tipo de instrução	ALU OP	Operação a executar	Campo Funct	Operação da ALU	Controlo da ALU
LW	Tipo-I	00	LOAD WORD	XXXXXX	ADD	010
SW	Tipo-I	00	STORE WORD	XXXXXX	ADD	010
BEQ	Tipo-I	01	BRANCH EQUAL	XXXXXX	SUBTRACT	110
ADD	Tipo-R	10	ADD	100000	ADD	010
SUB	Tipo-R	10	SUBTRACT	100010	SUBTRACT	110
AND	Tipo-R	10	AND	100100	AND	000
OR	Tipo-R	10	OR	100101	OR	001
SLT	Tipo-R	10	SET ON LESS THAN	101010	SET ON LESS THAN	111

Tabela 2. Sinais gerados pelo ALU Control em função do ALUOP e Funct recebidos

O uso de VHDL, fornece grandes vantagens na descrição das condições que se pretendem verificar, usando declarações *CASE-IS* ou *IF-THEN-ELSE* pode-se mesmo quase transcrever a leitura da Tabela 2, exemplo disso é a Figura 5 que representa a implementação usada em VHDL para o módulo *ALUControl*.

```
process(Funct, ALUOp)
begin
       case ALUOp is
       when "00" => Output <= "010";
       when "01" => Output <= "110";
       when others =>
                case Funct is
                when "100100" => Output <= "000";
                when "100101" => Output <= "001";
                when "100000" => Output <= "010";
                when "100010" => Output <= "110";
                when others   => Output <= "111";
                end case;
       end case;
end process;
```

Figura 5 – Código VHDL do módulo *ALUControl* usando declarações do tipo *CASE-IS*

A ALU possui dois sinais de saída, em que num coloca o resultado da operação aritmética/lógica efectuada (ALUResult) e noutro um sinal de um bit chamado Zero, que fica activo quando o resultado da ALU for igual a zero. Este último sinal é ligado à porta lógica AND que recebe também um sinal do Control Unit denominado de Branch (activo numa situação de salto conforme ilustrado na Figura 2). Quando estes dois sinais estão activos a condição de salto é validada e o valor do próximo endereço de memória de instruções que deve ser lido é o apontando pela instrução de salto.

D. Data Memory

A fase *Data Memory*, lê ou escreve valores da ou para a memória. Esta fase numa situação de *Branch* não produz qualquer tipo de efeito. A *Data Memory* (Figura 6) recebe dois valores à entrada correspondentes aos dados a serem escritos (*WriteData*) e o endereço de memória ao qual se pretende aceder (*Address*). Este módulo possui ainda um porto de saída no qual coloca o valor lido da memória (*ReadData*). Dois sinais separados, *MemWrite* e *MemRead*, são usados para garantir a escrita ou a leitura da memória, sendo esta acção sincronizada com o *MEMClock* e *CPUClock* (ver transição *DataMem* na Figura 1).

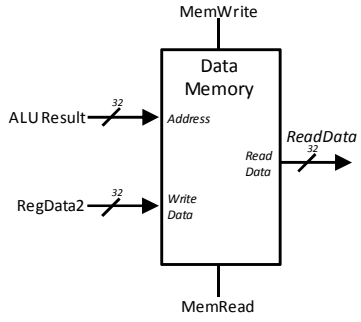


Figura 6. Diagrama do bloco da fase de *Data Memory*

Para ler da memória o sinal *MemRead* encontra-se activo e *MemWrite* desactivo. Para escrita estes sinais encontram-se invertidos, sendo por isso possível, se pretendido, uma optimização de recursos usando apenas um bit de controlo para escrita e leitura. Durante o desenvolvimento, uma pequena parcela de memória foi inicializada para efeitos de simulação e verificação de erros. A implementação deste tipo de Memória foi feita em módulos BlockRAM, usufruindo das vantagens das estruturas disponibilizadas pela FPGA. Em VHDL uma BlockRAM pode ser instanciada directamente ou usando uma sequência de instruções VHDL que a ferramenta de síntese (XST da Xilinx) consegue interpretar como sendo objectivo do projectista a implementação em BlockRAM do componente descrito. De seguida encontra-se uma parte parcial do código que demonstra a descrição em VHDL da memória de dados (Figura 7).

```
signal Memory : T_DataMem := DataMemory;


process (Clock, Enable)
begin
   if FALLING_EDGE(Clock)  then
        if MemRead = '1' and Enable = '0' then
           ReadData <= Memory(CONV_INTEGER(Address));
        elsif MemWrite = '1' and Enable = '0' then
           Memory(CONV_INTEGER(Address)) <= WriteData;
        end if;
   end if;
end process;
```

Figura 7 – Código VHDL que descreve a implementação de uma memória

E. Write Back

Nesta fase de uma instrução MIPS RISC, o resultado produzido em fases anteriores como *Execution* ou *DataMemory* pode ser escrito num registo. Um multiplexer determina qual dos resultados obtidos deve ser escrito no registo de destino. Esta operação de selecção é gerida pelo *Control Unit* que através do sinal *MemToReg* efectua o controlo sobre este multiplexer. As instruções envolvidas nesta fase são operações de Registo-Registo ou *Loadword*. Para operações Registo-Registo, o resultado da ALU é passado para escrita sendo necessário para isso o bit de selecção do multiplexer conter o valor '0' e para valores lidos da memória, encontra-se no sinal de selecção o valor activo "1", escrevendo assim no registo o conteúdo do sinal *ReadData* (pode-se visualizar o diagrama desta situação na Figura 8).

F. Control Unit

Em todas as fases do MIPS RISC, existem sinais de controlo como já referidos anteriormente. Estes, estão descritos na Tabela 3, e possuem a capacidade de alterar o funcionamento do *datapath* dependente da instrução que se pretende executar.

Fase do MIPS RISC	Sinais de controlo	Efeito dos sinais
Instruction Fetch	PCSrc	O valor do PC é substituído pelo endereço calculado pela instrução de salto ou pelo valor de PC+4.
Instruction Decode	RegDst	O número do registo de destino provém do valor dos bits 15-11 ou dos bits 20-16 da instrução.
	RegWrite	O registo de destino escolhido é escrito com o conteúdo do sinal ligado à entrada do porto WriteData.
Execution	ALUSrc	O segundo operando da ALU é os 16 bits menos significativos da instrução extendidos de sinal até 32 bits. Caso contrário é o valor do segundo registo lido (ReadData2).
Data Memory	MemWrite	O conteúdo da memória apontado pelo endereço é escrito com o valor à entrada do porto WriteData.
	MemRead	O conteúdo da memória apontado pelo endereço é lido e colocado no porto de saída (ReadData).
Write Back	MemtoReg	O valor que se encontra para escrita no registo provém da memória ou do resultado da ALU.

Tabela 3. Sinais de controlo em cada fase e seu efeito quando activos

A unidade de controlo (*Control Unit*) recebe à entrada o *Opcode* da instrução e gera na sua saída os sinais necessários para controlo de registos, memória, multiplexers e *ALUControl*.

Como só depende do *Opcode*, em cada ciclo de relógio os sinais de controlo apenas podem possuir o valor '1', '0' ou ser irrelevantes (*don't-care*) como se pode verificar na Tabela 4. No desenvolvimento em VHDL, o módulo *Control Unit* foi construído utilizando *CASE-IS* declarando de uma forma explícita todos os sinais gerados para cada *Opcode* de entrada.

Instruction	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp 1	ALUOp 0
Tipo-R	1	0	0	1	0	0	0	1	0
LW	0	1	1	1	1	0	0	0	0
SW	X	1	X	0	0	1	0	0	0
BEQ	X	0	X	0	0	0	1	0	1

Tabela 4. Valores atribuídos aos sinais de controlo dependendo do tipo de instrução

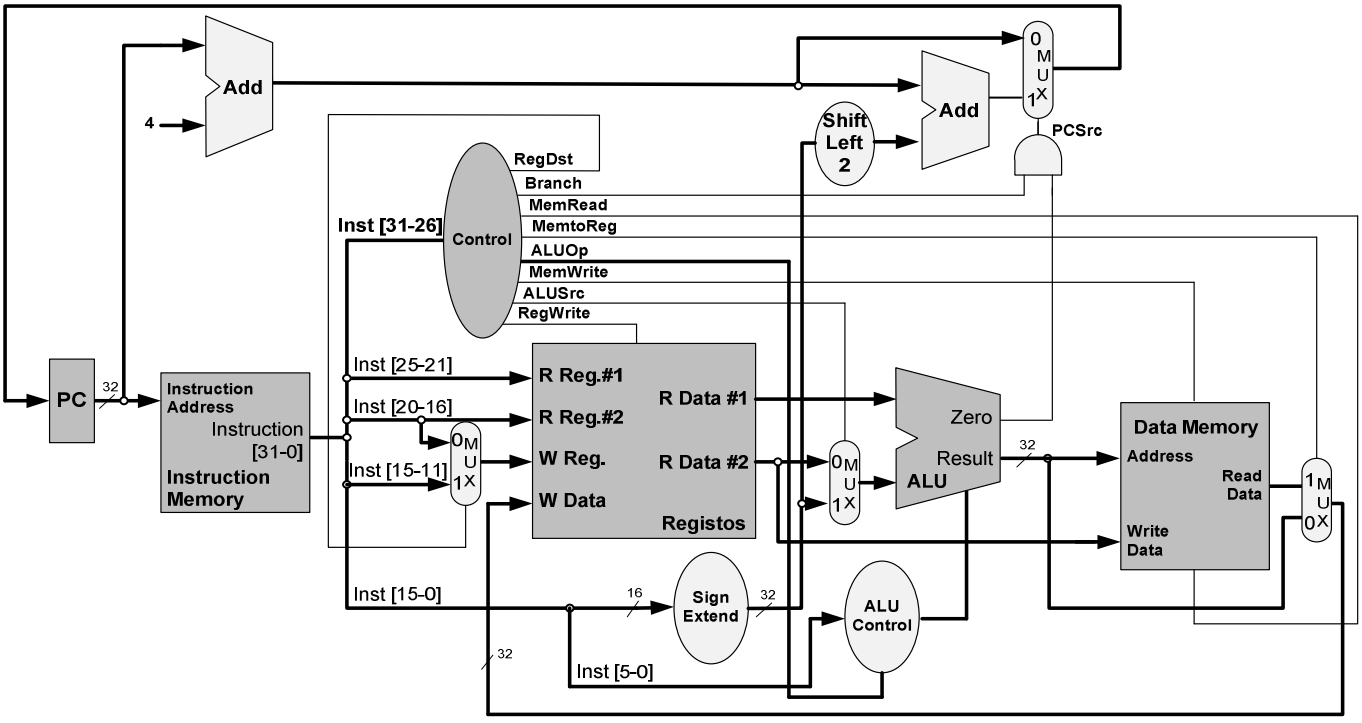


Figura 8. Arquitectura completa do MIPS RISC de 32 bits simplificado com a unidade de controlo e a fase de Write Back (imagem adapatada de [9])

III. SIMULAÇÃO E ANÁLISE

Para síntese e simulação deste projecto usou-se a ferramenta ISE Xilinx 9.2.04i [10]. Esta ferramenta possui um simulador integrado (ISE Simulator) no qual permitiu executar um trecho de código assembly (apresentado na Tabela 5) na arquitectura projectada e analisar os resultados obtidos com os valores esperados. Uma parte da memória de dados foi inicializada com alguns valores conforme se pode ver na Tabela 6.

Nº da Instrução	Endereço de Memória (Hex)	Instrução Assembly (valores em hexadecimal)
1.	00	lw r1, 0(r0)
2.	04	lw r3, 4(r0)
3.	08	lw r4, 8(r0)
4.	0C	lw r5,20(r0)
5.	10	lw r7,10(r0)
6.	14	Loop: lw r2, 0(r3)
7.	18	add r4,r4,r2
8.	1C	add r3,r3,r1
9.	20	slt r6,r3,r5
10.	24	beq r6,r7,Loop
11.	28	sw r4,24(r0)
12.	2C	lw r8,24(r0)

Tabela 5. Conteúdo da *Instruction Memory*

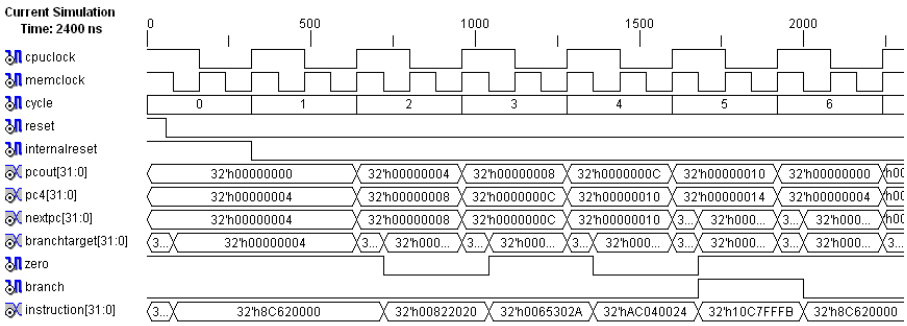
O código assembly consiste num conjunto inicial de instruções de leitura de valores da memória, seguido de um ciclo de 4 iterações que efectua o somatório dos elementos de um array de valores inteiros.

Endereço de Memória (Hex)	Conteúdo da Memória de Dados (Hex)
00	0x04
04	0x10
08	0x00
0C	0x08
10	0x01
14	0x02
18	0x03
1C	0x04
20	0x20
24	0x00

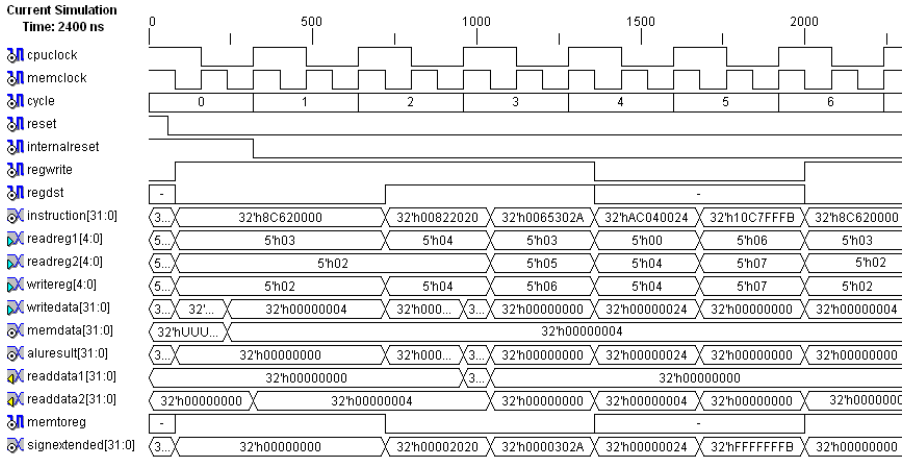
Tabela 6. Conteúdo da *DataMemory* após a inicialização

Por fim, o valor do somatório é escrito em memória, e novamente lido. Em seguida são mostrados alguns dos resultados obtidos correspondentes à simulação sequencial das instruções número 6, 7, 9, 11 e 10 (a troca das instruções 10 e 11 foi propositada para efeitos de demonstração das simulações). Deste modo pode-se verificar o PC a iniciar com o valor 0x00 na instrução 6 até ao valor 0x10 da instrução 10, voltando novamente a 0x00 devido à instrução 10 corresponder a um branch taken, neste subconjunto de instruções simuladas. O subconjunto de instruções escolhido permite visualizar os valores dos sinais no datapath para diferentes formatos de instrução.

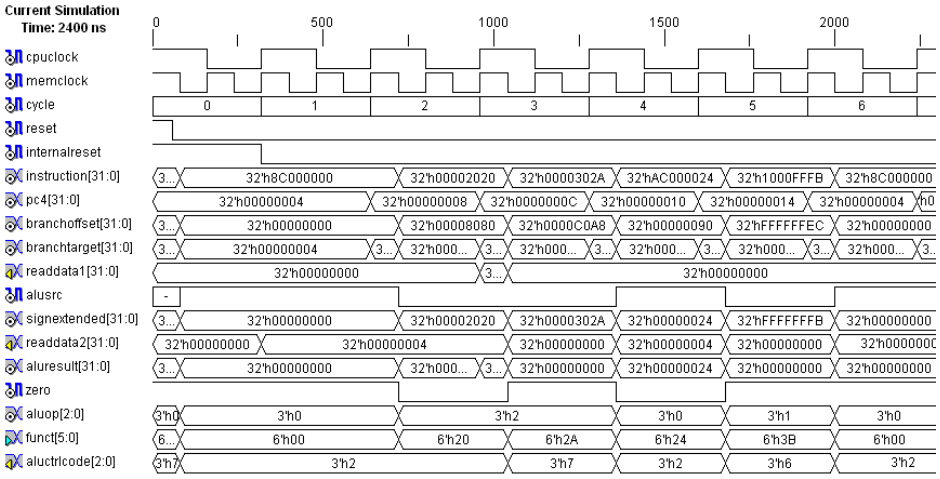
SIMULAÇÃO DA FASE INSTRUCTION FETCH:



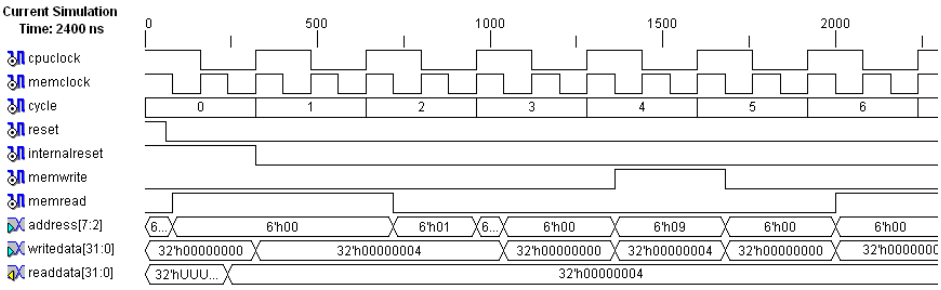
SIMULAÇÃO DA FASE INSTRUCTION DECODE:



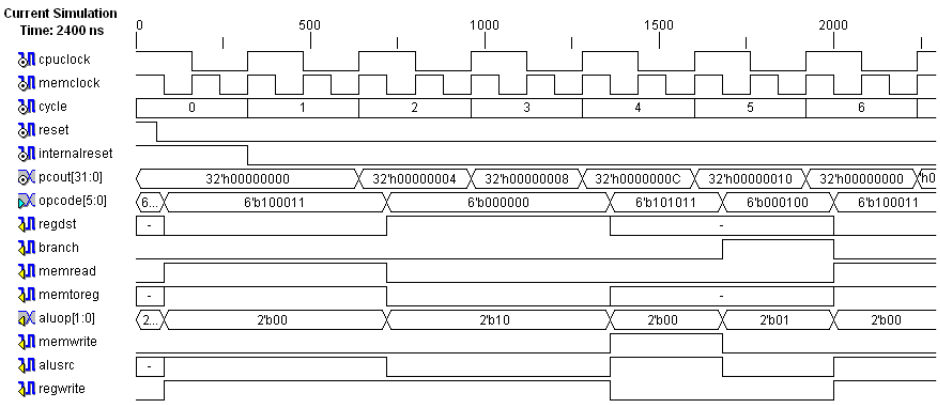
SIMULAÇÃO DA FASE EXECUTION:



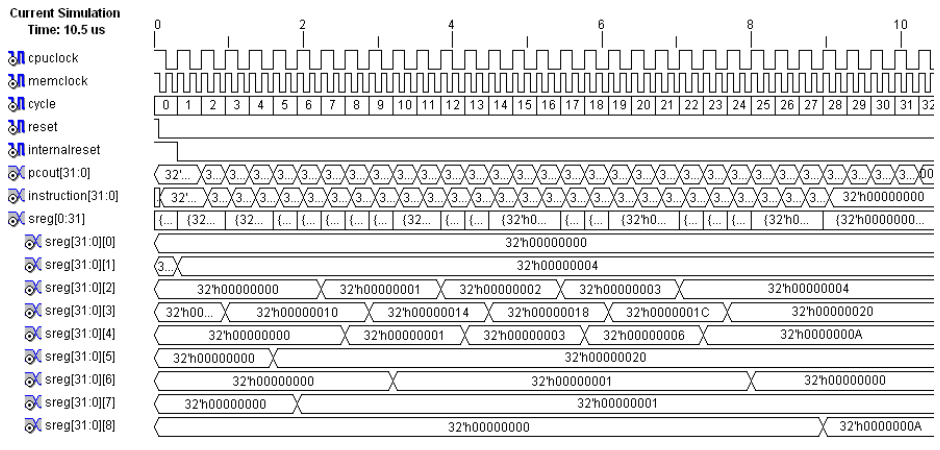
SIMULAÇÃO DA FASE DATAMEMORY E WRITEBACK:



SIMULAÇÃO DA UNIDADE DE CONTROLO:



SIMULAÇÃO DA ARQUITECTURA DESENVOLVIDA PARA TODAS AS INSTRUÇÕES DA TABELA 5:



No fim desta execução de código é possível verificar que os valores finais correspondem aos pretendidos, demonstrando assim que todos os valores foram devidamente gerados e calculados.

Foi também implementada uma versão ligeiramente alterada, no âmbito da disciplina de opção do 5º ano para MIECT e MIEET, Modelação e Síntese de Processadores, que permitiu visualizar o resultado final iluminando 8 LEDs correspondentes aos 8 bits menos significativos do resultado pretendido para o registo r8. Isto foi possível usando o bit 7 do resultado da ALU para decisão de escrita para os LEDs. Quando a '1' escreve na memória de dados e também para os LEDs da placa *Spartan 3 Starter Board* da *Digilent Inc* [10], iluminando-os.

IV. FUTURO TRABALHO

Um dos objectivos pretendidos neste projecto é que de uma forma fácil e construtiva, um aluno possa ir aprendendo VHDL. Para o efeito, será de todo interessante o aluno poder efectuar uma implementação pessoal deste tipo de arquitectura e posteriormente possuir uma ferramenta que permita visualizar os valores contidos nos sinais em cada ciclo de relógio durante a execução de cada instrução na arquitectura MIPS RISC desenvolvida. Uma interface gráfica está a ser desenvolvida em C#, de modo a tirar partido do facto de a placa DETIUA-S3 [12] já possuir uma ligação USB pela qual se efectua toda a transmissão de dados de sua configuração. Assim, será criada uma máquina de estados que implemente um protocolo de comunicação entre o módulo MIPS desenvolvido e a memória flash da placa DETIUA-S3. Utilizando este protocolo de comunicação será possível que o utilizador possua uma maior interactividade no que diz respeito a carregamento de instruções, inicialização da memória de dados, controlo de ciclo de relógio e visualização de valores da implementação do seu próprio código VHDL de uma versão MIPS. Se tais objectivos forem alcançados, quem sabe, um futuro aluno da Universidade de Aveiro poderá aprender e ter uma maior proximidade com código assembly, Arquitectura de Computadores, VHDL, síntese de processadores e circuitos digitais numa única ferramenta.

V. CONCLUSÕES

Ao longo do desenvolvimento deste projecto foram adquiridos diversificados conhecimentos em VHDL. Estes conhecimentos contribuíram para sucessivas optimizações de alguns dos módulos. Em título de referência, a utilização de um pacote em VHDL para declarar as dimensões do barramento do *datapath*, tamanho das memórias e inicializações das mesmas e mesmo alguns tipos de dados. Desta forma o MIPS implementado possui a característica de possuir um código VHDL simples, legível e (apesar de limitado) parametrizável.

Este tipo de projecto veio também a proporcionar a aplicação de vários conhecimentos adquiridos no âmbito de múltiplas disciplinas do curso de Mestrado Integrado em Engenharia de Computadores e Telemática.

Por simulação após a verificação dos resultados obtidos corresponderem aos pretendidos, e utilização de outras sequências de instruções, pequenos programas em assembly, pode-se afirmar que o processador MIPS Single Cycle simplificado, implementado em VHDL foi desenvolvido e testado à frequência de relógio de 50 MHz na placa *Spartan-3 Starter Board* com sucesso.

REFERÊNCIAS

[1] Linley Gwennap , FPGA Evolution, online: http://www.linleygroup.com/columns/nikkei1006.html

[2] Material e trabalho desenvolvido no âmbito da disciplina Modelação e Síntese de Processadores 2007/08:
- WIKI de VHDL desenvolvido pelos alunos: http://wsl.cemed.ua.pt/vhdl/index.php
- The *XST User Guide:* http://wsl.cemed.ua.pt/moodle/file.php/4/docs/XSTUserGuide.pdf

[3] William N. Joy, Co-Founder, and Vice President for Research on Sun Microsystems, Inc. online: http://www.cs.washington.edu/homes/lazowska/cra/risc.html

[4] Informações genéricas disponíveis online a partir do site: http://www.mips.com/

[5] A single clock cycle MIPS RISC processor design using VHDL Reaz M.B.I, Islam M.S., Sulaiman, M.S., Semiconductor Electronics, Dec. 2002, pp 199 – 203.

[6] Xilinx, online: http://www.xilinx.com/products/silicon_solutions/fpgas/spartan_series/spartan3_fpgas/

[7] Tutoriais online disponíveis no âmbito da disciplina Sistemas Digitais Reconfiguráveis: http://www.ieeta.pt/~skl/SDR_V.html

[8] P. J. Ashenden, VHDL Tutorial, EDA Consultant, Ashenden Designs Pty. LTD.

[9] Material didáctico da disciplina Arquitectura de Computadores I 2006/2007, acetatos teóricos [17-19]

[10] ISE Xilinx 9.2i homepage online: http://www.xilinx.com/ise/logic_design_prod/foundation.htm

[11] Manual da placa Digilent Inc, Spartan-3 Starter Board online: http://www.digilentinc.com/Data/Products/S3BOARD/S3BOARD-rm.pdf

[12] Manual de Utilizador da Placa DETIUA-S3, Versão 1.0 por Manuel Almeida, online: www.ieeta.pt/~skl/Research/Projects/Manual_Utilizador_pt.pdf

[13] MIPS Instruction Reference online: http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

[14] Wikipédia Online: http://en.wikipedia.org/wiki/MIPS_architecture