LABORATÓRIO DE ARQUITETURA DE COMPUTADORES

Experimento 3

Memória de Dados

GRUPO: 5 TURMA: B

Caroline Aparecida de Paula 726506

Gabriel Rodrigues Rocha 726518

Henrique Shinki Kodama 726537

Isabela Sayuri Matsumoto 726539

# Resumo

O Experimento 3 baseou-se em prosseguir com a descrição, em VHDL, do processador MIPS simplificado. O objetivo do experimento foi adicionar, na parte inicial já descrita nos relatórios dos Experimentos 1 e 2, as operações de carregar (*Load*)*,* gravar (*Store*)e o salto condicional (*Beq*).

Tais operações são do tipo *I-format*: elas possuem 6 bits para código da operação, 5 bits para cada um dos dois registradores e 16 bits para endereços e valores imediatos. A unidade de controle (*Control*), além de habilitar um sinal *Reg\_write* que indica quando há escrita nos registradores (*R-format*), habilita também um sinal que indica que a instrução é do tipo *I-format*.O *Load* possuium sinal para leitura da memória e o *Store* possui um sinal de escrita na memória. Esses sinais são acionados em suas devidas instruções, não podendo estarem ativos alto ao mesmo tempo. Foi também adicionado um sinal específico *Branch* que é ativo alto quando a instrução em questão é de salto.

Na unidade de decodificação (*Idecode),* além das ações já descritas no Experimento 2, foi adicionado um multiplexador que seleciona entre o resultado do ULA (Unidade de Lógica Aritmética) e o dado da memória para escrever no registrador.

Na unidade de execução (*Execute*)*,* foi acrescentado um multiplexador que seleciona a entrada do somador entre os sinais *read\_data\_2* (*R-format*)e *SignExtend* (*I-format*). Além disso, para a instrução *beq* foi criado um sinal *Zero* que é ativo alto quando os conteúdos dos dois registradores são iguais e um sinal *Add\_result* recebe a soma do contador de programas (*PC*), já incrementado, com o número contido nos 16 bits menos significativos da instrução com a extensão de sinal.

Na unidade *Fetch*, foi adicionado um multiplexador que altera o valor do contador de programa (*PC*) para *Add\_result* quando os sinais *Branch* e *Zero* são ativos alto.

Também foi adicionado um módulo de memória de dados, chamado *DMemory*, cujo uso é abstraído no projeto.

O objetivo do experimento foi alcançado, pois as instruções adicionadas funcionaram corretamente, como foi demostrado durante a simulação e execução na placa, no laboratório.

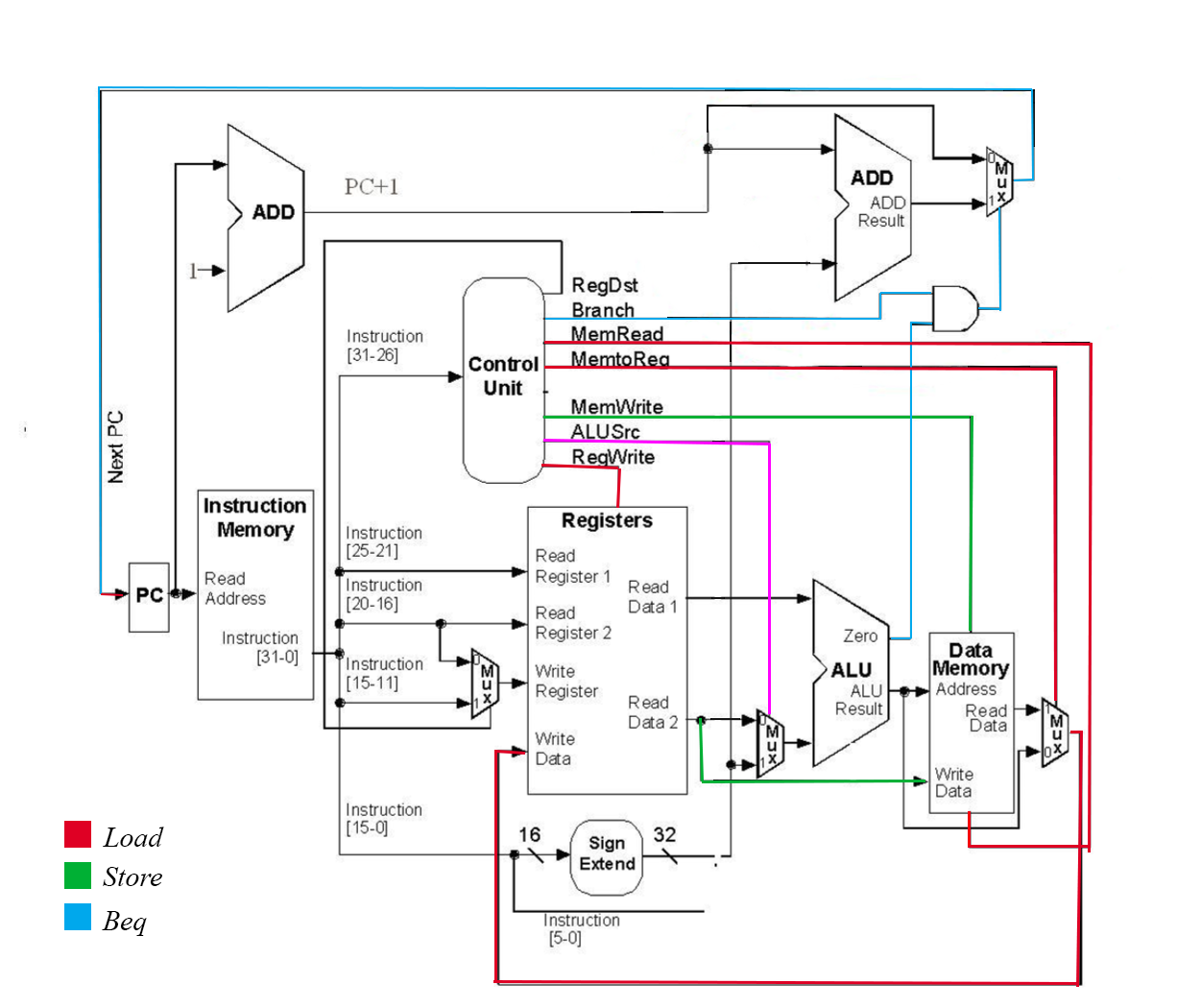


Figura 1: Diagrama do Projeto

# Código

Segue o código do módulo *Control*:

-- control module

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

ENTITY control IS

PORT( Opcode : IN STD\_LOGIC\_VECTOR( 5 DOWNTO 0 );

RegDst : OUT STD\_LOGIC;

RegWrite : OUT STD\_LOGIC;

MemToReg : OUT STD\_LOGIC;

MemRead : OUT STD\_LOGIC;

MemWrite : OUT STD\_LOGIC;

AluSrc : OUT STD\_LOGIC;

Branch : OUT STD\_LOGIC);

END control;

ARCHITECTURE behavior OF control IS

SIGNAL R\_format : STD\_LOGIC;

SIGNAL I\_format : STD\_LOGIC;

SIGNAL J\_format : STD\_LOGIC;

BEGIN

R\_format <= '1' WHEN Opcode = "000000" ELSE '0';

I\_format <= '1' WHEN Opcode = "100011" ELSE --LOAD

'1' WHEN Opcode = "101011" ELSE --STORE

'1' WHEN Opcode = X"4" ELSE --BEQ

'1' WHEN Opcode = X"5" ELSE --BNEQ

'1' WHEN Opcode = X"8" ELSE'0'; --ADDI

RegDst <= R\_format;

RegWrite <= '1' WHEN R\_format = '1' ELSE

'1' WHEN Opcode = "100011" ELSE

'0';

MemWrite <= '1' WHEN Opcode = "101011" ELSE '0'; --Store

MemRead <= '0' WHEN Opcode = "101011" ELSE '1';

MemToReg <= '1' WHEN Opcode = "100011" ELSE '0'; --Load

AluSrc <= I\_Format;

Branch <= '1' WHEN Opcode = "000100" ELSE '0'; -- BEQ

END behavior;

Segue o código do módulo *Ifetch*:

-- fetch module  
LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL; -- Tipo de sinal STD\_LOGIC e STD\_LOGIC\_VECTOR

USE IEEE.STD\_LOGIC\_ARITH.ALL; -- Operacoes aritmeticas sobre binarios

USE IEEE.STD\_LOGIC\_UNSIGNED.ALL;

LIBRARY altera\_mf;

USE altera\_mf.altera\_mf\_components.ALL;

ENTITY Ifetch IS

PORT( reset : in STD\_LOGIC;

clock : in STD\_LOGIC;

Branch : in STD\_LOGIC;

Zero : in STD\_LOGIC;

ADDResult : in STD\_LOGIC\_VECTOR(7 DOWNTO 0);

PC\_out : out STD\_LOGIC\_VECTOR(7 DOWNTO 0);

Instruction : out STD\_LOGIC\_VECTOR(31 DOWNTO 0));

END Ifetch;

ARCHITECTURE behavior OF Ifetch IS

SIGNAL PC : STD\_LOGIC\_VECTOR(7 DOWNTO 0);

SIGNAL Next\_PC : STD\_LOGIC\_VECTOR(7 DOWNTO 0);

SIGNAL PC\_inc : STD\_LOGIC\_VECTOR(7 DOWNTO 0);

SIGNAL Mem\_Addr : STD\_LOGIC\_VECTOR(7 DOWNTO 0);

BEGIN

--Descricao da Memoria

data\_memory: altsyncram -- Declaracao do compomente de memoria

GENERIC MAP(

operation\_mode => "ROM",

width\_a => 32, -- tamanho da palavra (Word)

widthad\_a => 8, -- tamanho do barramento de endereco

lpm\_type => "altsyncram",

outdata\_reg\_a => "UNREGISTERED",

init\_file => "program.mif", -- arquivo com estado inicial

intended\_device\_family => "Cyclone")

PORT MAP(

address\_a => Mem\_Addr,

q\_a => Instruction,

clock0 => clock); -- sinal de clock da memoria

-- Descricao do somador

PC\_inc <= PC+1;

-- Descricao do registrador

PROCESS

BEGIN

WAIT UNTIL (clock'event AND clock='1');

IF reset='1' THEN

PC <= "00000000";

ELSE

PC <= Next\_PC;

END IF;

END PROCESS;

-- Usar o Next\_PC ao inves do PC porque a memoria tem um registrador de entrada interno

-- Entao o PC tem que ser atualizado simultaneamente com o reg interno da memoria

Mem\_Addr <= Next\_PC;

Next\_PC <= "00000000" WHEN reset='1' ELSE

ADDResult WHEN (Branch = '1' AND Zero = '1') ELSE

PC\_inc;

PC\_out <= PC;

END behavior;

Segue o código do módulo *Idecode*:

--decode module  
LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

USE IEEE.STD\_LOGIC\_ARITH.ALL;

USE IEEE.STD\_LOGIC\_UNSIGNED.ALL;

ENTITY Idecode IS

PORT( read\_data\_1 : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

read\_data\_2 : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

Instruction : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

ALU\_Result : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

Data\_Mem : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

RegWrite : IN STD\_LOGIC;

RegDst : IN STD\_LOGIC;

Sign\_extend : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

clock,reset : IN STD\_LOGIC;

MemToReg : IN STD\_LOGIC;

MemAddr : OUT STD\_LOGIC\_VECTOR( 7 DOWNTO 0));

END Idecode;

ARCHITECTURE behavior OF Idecode IS

TYPE register\_file IS ARRAY ( 0 TO 31 ) OF STD\_LOGIC\_VECTOR(31 DOWNTO 0);

SIGNAL reg\_bank: register\_file;

SIGNAL write\_reg\_ID: STD\_LOGIC\_VECTOR(4 DOWNTO 0);

SIGNAL write\_data: STD\_LOGIC\_VECTOR(31 DOWNTO 0);

SIGNAL read\_Rs\_ID: STD\_LOGIC\_VECTOR(4 DOWNTO 0);

SIGNAL read\_Rt\_ID: STD\_LOGIC\_VECTOR(4 DOWNTO 0);

SIGNAL write\_Rd\_ID: STD\_LOGIC\_VECTOR(4 DOWNTO 0);

SIGNAL write\_Rt\_ID: STD\_LOGIC\_VECTOR(4 DOWNTO 0);

SIGNAL Immediate\_value: STD\_LOGIC\_VECTOR(15 DOWNTO 0);

SIGNAL readDataSignal1 : STD\_LOGIC\_VECTOR(31 DOWNTO 0);

SIGNAL readDataSignal2 : STD\_LOGIC\_VECTOR(31 DOWNTO 0);

BEGIN

-- Os sinais abaixo devem receber as identificacoes dos registradores

-- que estao definidos na instrucao, ou seja, o indice dos registradores

-- a serem utilizados na execucao da instrucao

read\_Rs\_ID <= Instruction(25 DOWNTO 21);

read\_Rt\_ID <= Instruction(20 DOWNTO 16);

write\_Rd\_ID <= Instruction(15 DOWNTO 11);

write\_Rt\_ID <= Instruction(20 DOWNTO 16);

Immediate\_value <= Instruction(15 DOWNTO 0);

readDataSignal1 <= reg\_bank(CONV\_INTEGER(read\_Rs\_ID));

readDataSignal2 <= reg\_bank(CONV\_INTEGER(read\_Rt\_ID));

-- Os sinais abaixo devem receber o conteudo dos registradores, reg(i)

-- USE "CONV\_INTEGER(read\_Rs\_ID)" para converser os bits de indice do registrador

-- para um inteiro a ser usado como indice do vetor de registradores.

-- Exemplo: dado um sinal X do tipo array de registradores,

-- X(CONV\_INTEGER("00011")) recuperaria o conteudo do registrador 3.

read\_data\_1 <= readDataSignal1;

read\_data\_2 <= readDataSignal2;

-- Crie um multiplexador que seleciona o registrador de escrita de acordo com o sinal RegDst

write\_reg\_ID <= write\_Rd\_ID WHEN RegDst = '1' ELSE write\_Rt\_ID;

-- Ligue no sinal abaixo os bits relativos ao valor a ser escrito no registrador destino.

-- adicionar um multiplex que seleciona entre o dado da memoria ou o ULA

write\_data <= ALU\_Result WHEN memToReg = '0' ElSE Data\_Mem;

-- Estenda o sinal de instrucoes do tipo I de 16-bits to 32-bits

-- Faca isto independente do tipo de instrucao, mas use apenas quando

-- for instrucao do tipo I.

Sign\_extend <= X"0000" & Immediate\_value

WHEN Immediate\_value(15) = '0'

ELSE X"FFFF" & Immediate\_value;

MemAddr <= readDataSignal1( 7 DOWNTO 0) + Immediate\_value (7 DOWNTO 0);

PROCESS

BEGIN

WAIT UNTIL clock'EVENT AND clock = '1';

IF reset = '1' THEN

-- Inicializa os registradores com seu numero

FOR i IN 0 TO 31 LOOP

reg\_bank(i) <= CONV\_STD\_LOGIC\_VECTOR( i, 32 );

END LOOP;

ELSIF (RegWrite = '1' OR MemToReg = '1') AND write\_reg\_ID /= "00000" THEN

reg\_bank(CONV\_INTEGER(write\_reg\_ID)) <= write\_data;

END IF;

END PROCESS;

END behavior;

Segue o código da entidade *Execute*:

--execute module  
LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

USE IEEE.STD\_LOGIC\_ARITH.ALL;

USE IEEE.STD\_LOGIC\_SIGNED.ALL;

ENTITY Execute IS

PORT( Read\_data1 : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

Read\_data2 : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

PC : IN STD\_LOGIC\_VECTOR (7 DOWNTO 0);

ALU\_Result : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0);

Signal\_Ext : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

Alu\_Src : IN STD\_LOGIC;

Zero : OUT STD\_LOGIC;

ADDResult : OUT STD\_LOGIC\_VECTOR (7 DOWNTO 0));

END Execute;

ARCHITECTURE behavior OF Execute IS

SIGNAL iAux : STD\_LOGIC\_VECTOR(31 DOWNTO 0);

BEGIN

iAux <= Read\_data2 WHEN Alu\_src = '0' ELSE Signal\_Ext;

ALU\_Result <= Read\_data1 + iAux;

Zero <= '0' WHEN (Read\_data1 /= Read\_data2) ELSE '1';

ADDResult <= PC + 1 + Signal\_Ext (7 DOWNTO 0);

--- multiplex ---

END behavior;

Por fim, o código da TLE, *Exp03*:

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

ENTITY Exp03 IS

PORT( reset : IN STD\_LOGIC;

clock48MHz : IN STD\_LOGIC;

LCD\_RS, LCD\_E : OUT STD\_LOGIC;

LCD\_RW, LCD\_ON : OUT STD\_LOGIC;

DATA : INOUT STD\_LOGIC\_VECTOR(7 DOWNTO 0);

clock : IN STD\_LOGIC;

InstrALU : IN STD\_LOGIC);

END Exp03;

ARCHITECTURE exec OF Exp03 IS

COMPONENT LCD\_Display

GENERIC(NumHexDig: Integer:= 11);

PORT( reset, clk\_48Mhz : IN STD\_LOGIC;

HexDisplayData : IN STD\_LOGIC\_VECTOR((NumHexDig\*4)-1 DOWNTO 0);

LCD\_RS, LCD\_E : OUT STD\_LOGIC;

LCD\_RW : OUT STD\_LOGIC;

DATA\_BUS : INOUT STD\_LOGIC\_VECTOR(7 DOWNTO 0));

END COMPONENT;

COMPONENT Ifetch

PORT( reset : in STD\_LOGIC;

clock : in STD\_LOGIC;

Branch : in STD\_LOGIC;

Zero : in STD\_LOGIC;

ADDResult : in STD\_LOGIC\_VECTOR(7 DOWNTO 0);

PC\_out : out STD\_LOGIC\_VECTOR(7 DOWNTO 0);

Instruction : out STD\_LOGIC\_VECTOR(31 DOWNTO 0));

END COMPONENT;

COMPONENT Idecode

PORT( read\_data\_1 : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

read\_data\_2 : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

Instruction : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

ALU\_Result : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

Data\_Mem : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

RegWrite : IN STD\_LOGIC;

RegDst : IN STD\_LOGIC;

Sign\_extend : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

clock,reset : IN STD\_LOGIC;

MemToReg : IN STD\_LOGIC;

MemAddr : OUT STD\_LOGIC\_VECTOR( 7 DOWNTO 0));

END COMPONENT;

COMPONENT Control

PORT( Opcode : IN STD\_LOGIC\_VECTOR( 5 DOWNTO 0 );

RegDst : OUT STD\_LOGIC;

RegWrite : OUT STD\_LOGIC;

MemToReg : OUT STD\_LOGIC;

MemRead : OUT STD\_LOGIC;

MemWrite : OUT STD\_LOGIC;

AluSrc : OUT STD\_LOGIC;

Branch : OUT STD\_LOGIC);

END COMPONENT;

COMPONENT Execute

PORT( Read\_data1 : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

Read\_data2 : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

PC : IN STD\_LOGIC\_VECTOR (7 DOWNTO 0);

ALU\_Result : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0);

Signal\_Ext : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

Alu\_Src : IN STD\_LOGIC;

Zero : OUT STD\_LOGIC;

ADDResult : OUT STD\_LOGIC\_VECTOR (7 DOWNTO 0));

END COMPONENT;

COMPONENT Dmemory

PORT( read\_data : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

address : IN STD\_LOGIC\_VECTOR( 7 DOWNTO 0 );

write\_data : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );

MemRead, Memwrite : IN STD\_LOGIC;

clock,reset : IN STD\_LOGIC );

END COMPONENT;

SIGNAL DataInstr : STD\_LOGIC\_VECTOR(31 DOWNTO 0);

SIGNAL DisplayData : STD\_LOGIC\_VECTOR(31 DOWNTO 0);

SIGNAL PCAddr : STD\_LOGIC\_VECTOR(7 DOWNTO 0);

SIGNAL RegDst : STD\_LOGIC;

SIGNAL RegWrite : STD\_LOGIC;

SIGNAL ALUResult : STD\_LOGIC\_VECTOR(31 DOWNTO 0);

SIGNAL SignExtend : STD\_LOGIC\_VECTOR(31 DOWNTO 0);

SIGNAL readData1 : STD\_LOGIC\_VECTOR(31 DOWNTO 0);

SIGNAL readData2 : STD\_LOGIC\_VECTOR(31 DOWNTO 0);

SIGNAL HexDisplayDT : STD\_LOGIC\_VECTOR(43 DOWNTO 0);

SIGNAL auxAluSrc : STD\_LOGIC;

SIGNAL auxMemWrite : STD\_LOGIC;

SIGNAL auxMemToReg : STD\_LOGIC;

SIGNAL auxMemRead : STD\_LOGIC;

SIGNAL DMemoryOut : STD\_LOGIC\_VECTOR(31 DOWNTO 0);

SIGNAL auxMemAddr : STD\_LOGIC\_VECTOR(7 DOWNTO 0);

SIGNAL auxBranch : STD\_LOGIC;

SIGNAL auxZero : STD\_LOGIC;

SIGNAL auxAddResult : STD\_LOGIC\_VECTOR (7 DOWNTO 0);

BEGIN

LCD\_ON <= '1';

-- Inserir MUX para DisplayData

displayData <= DataInstr WHEN InstrALU = '1' ELSE AluResult;

HexDisplayDT <= "0000" & PCAddr & DisplayData;

lcd: LCD\_Display

PORT MAP(

reset => reset,

clk\_48Mhz => clock48MHz,

HexDisplayData => HexDisplayDT,

LCD\_RS => LCD\_RS,

LCD\_E => LCD\_E,

LCD\_RW => LCD\_RW,

DATA\_BUS => DATA);

IFT: Ifetch

PORT MAP(

reset => reset,

clock => clock,

PC\_out => PCAddr,

Instruction => DataInstr,

Branch => auxBranch,

Zero => auxZero,

AddResult => auxAddResult);

CTR: Control

PORT MAP(

Opcode => DataInstr(31 DOWNTO 26),

RegDst => RegDst,

RegWrite => RegWrite,

MemToReg => auxMemToReg,

MemRead => auxMemRead,

MemWrite => auxMemWrite,

AluSrc => auxAluSrc,

Branch => auxBranch);

IDEC: Idecode

PORT MAP(

read\_data\_1 => readData1,

read\_data\_2 => readData2,

Instruction => DataInstr,

ALU\_Result => ALUResult,

Data\_Mem => DMemoryOut,

RegWrite => RegWrite,

RegDst => RegDst,

Sign\_extend => SignExtend,

clock => clock,

reset => reset,

MemToReg => auxMemToReg,

MemAddr => auxMemAddr);

EXE: Execute

PORT MAP(

Read\_data1 => readData1,

Read\_data2 => readData2,

ALU\_Result => ALUResult,

Signal\_Ext => signExtend,

PC => PCAddr,

Alu\_Src => auxAluSrc,

Zero => auxZero,

AddResult => auxAddResult);

DMEM: Dmemory

PORT MAP(

read\_data => DMemoryOut,

address => auxMemAddr,

write\_data => readData2,

MemRead => auxMemRead,

Memwrite => auxMemWrite,

clock => clock,

reset => reset);

END exec;

O código da nova entidade de memória, *DMemory*, será omitido pois não é relevante para a análise de funcionamento.

# Simulação e testeC:\Users\Gabriel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\screenshot.190.png

Figura 2: Conteúdo de program.mif

Figura 3: Conteúdo de DMEMORY.mif

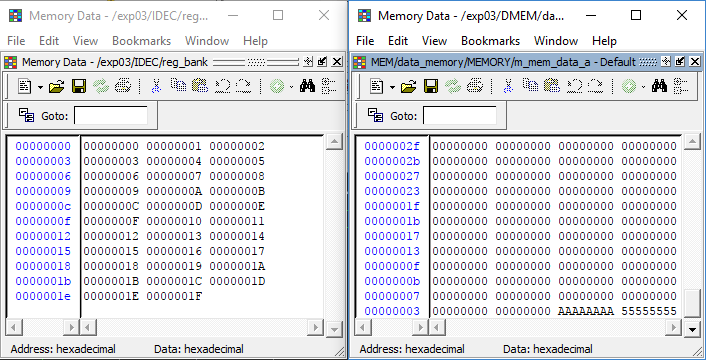


Figura 4: Estado inicial dos registradores e memória de dados

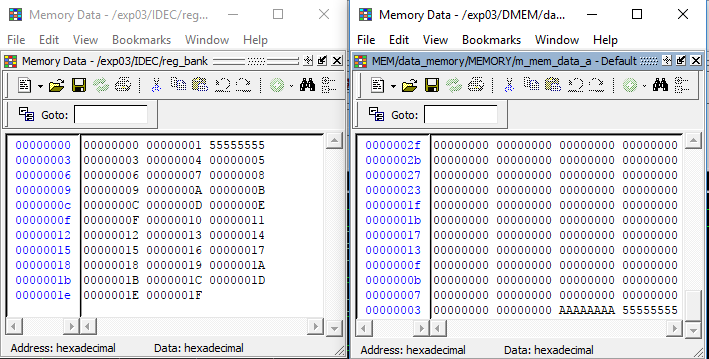


Figura 5: Registradores e memória de dados após 1ª instrução

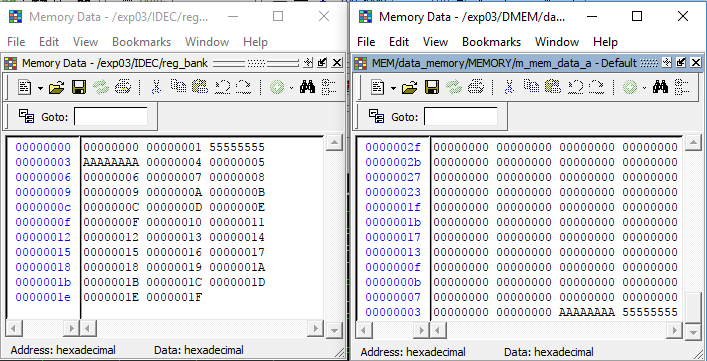


Figura 6: Registradores e memória de dados após 2ª instrução

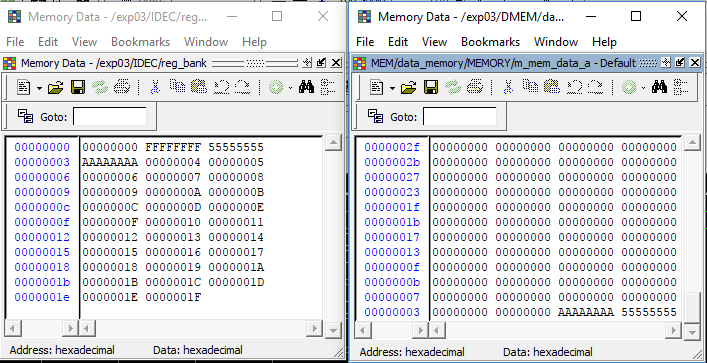


Figura 7: Registradores e memória de dados após 3ª instrução

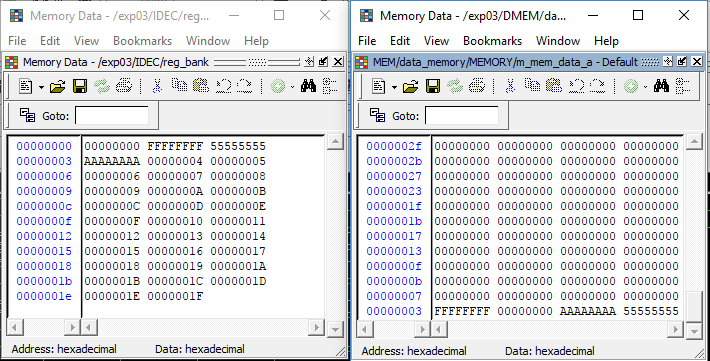


Figura 8: Registradores e memória de dados após 4ª instrução

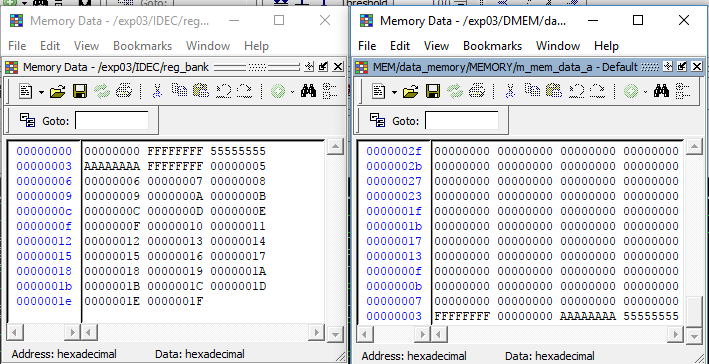


Figura 9: Registradores e memória de dados após 5ª instrução

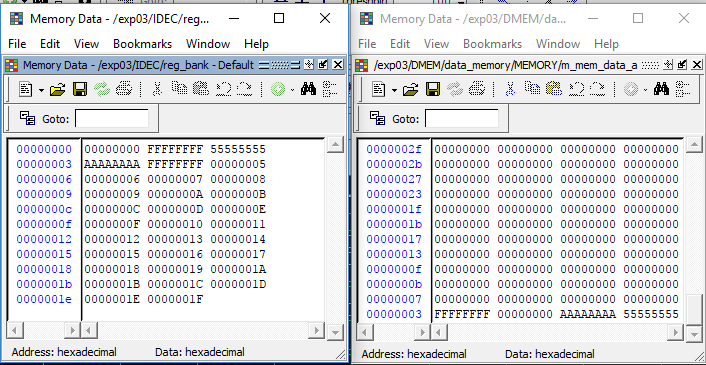


Figura 10: Registradores e memória de dados após 6ª instrução

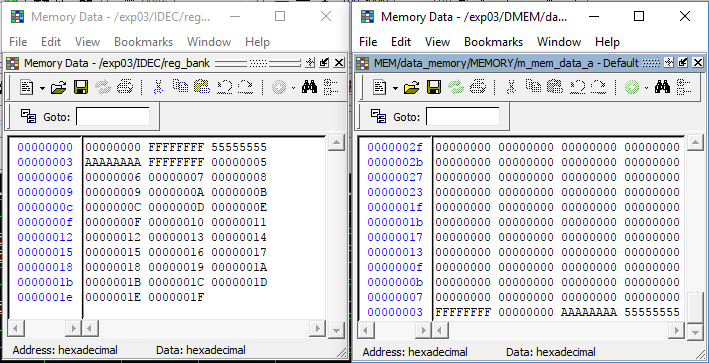
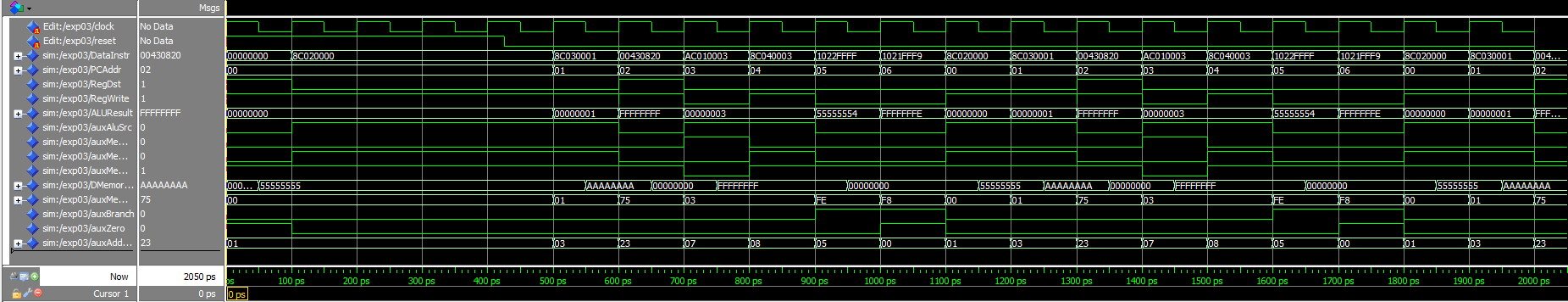


Figura 11: Registradores e memória de dados ficam constantes após a execução da 6ª instrução

Figura 12: Waves resultantes da simulação



## Discussão

Explicação pontual **RELACIONANDO** o código VHDL com as mudanças nos sinais de saída para a imagem de simulação apresentada (item 3), associe também ao conteúdo de memória. **Esta explicação deve deixar claro que a simulação representa o funcionamento correto do projeto descrito**.

|  |  |  |
| --- | --- | --- |
|  | **Instrução Hexadecimal** | **Instrução MIPS** |
| 00 | 8C020000 | LW $v0, 0x0000($zero) |
| 01 | 8C030001 | LW $v1, 0x0001($zero) |
| 02 | 00430820 | ADD $at, $v0, $v1 |
| 03 | AC010003 | SW $at, 0x0003($zero) |
| 04 | 8C040003 | LW $a0, 0x0003($zero) |
| 05 | 1022FFFF | BEQ $at, $v0, 0xFFFF |
| 06 | 1021FFF9 | BEQ $at, $at, 0xFFF9 |