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.

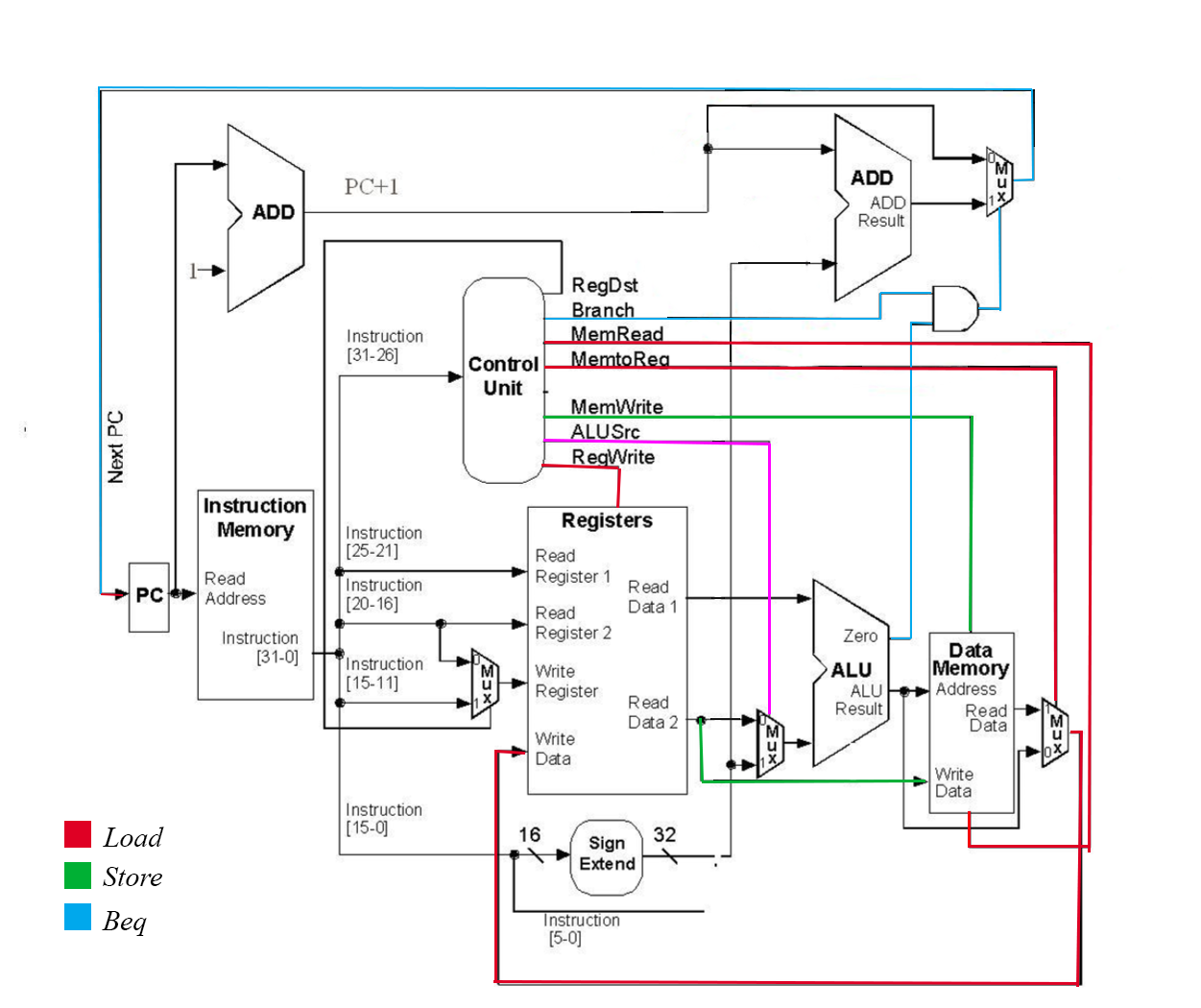


Figura 1: Diagrama do Projeto

# Código

Segue o código do módulo *Control*:

1. -- control module
2. LIBRARY IEEE;
3. USE IEEE.STD\_LOGIC\_1164.ALL;
4. ENTITY control IS
5. PORT( Opcode : IN STD\_LOGIC\_VECTOR( 5 DOWNTO 0 );
6. RegDst : OUT STD\_LOGIC;
7. RegWrite : OUT STD\_LOGIC;
8. MemToReg : OUT STD\_LOGIC;
9. MemRead : OUT STD\_LOGIC;
10. MemWrite : OUT STD\_LOGIC;
11. AluSrc : OUT STD\_LOGIC;
12. Branch : OUT STD\_LOGIC);
13. END control;
14. ARCHITECTURE behavior OF control IS
15. SIGNAL R\_format : STD\_LOGIC;
16. SIGNAL I\_format : STD\_LOGIC;
17. SIGNAL J\_format : STD\_LOGIC;
18. BEGIN
19. R\_format <= '1' WHEN Opcode = "000000" ELSE '0';
20. I\_format <= '1' WHEN Opcode = "100011" ELSE --LOAD
21. '1' WHEN Opcode = "101011" ELSE --STORE
22. '1' WHEN Opcode = X"4" ELSE --BEQ
23. '1' WHEN Opcode = X"5" ELSE --BNEQ
24. '1' WHEN Opcode = X"8" ELSE'0'; --ADDI
26. RegDst <= R\_format;
27. RegWrite <= '1' WHEN R\_format = '1' ELSE
28. '1' WHEN Opcode = "100011" ELSE
29. '0';
31. MemWrite <= '1' WHEN Opcode = "101011" ELSE '0'; --Store
32. MemRead <= '0' WHEN Opcode = "101011" ELSE '1';
33. MemToReg <= '1' WHEN Opcode = "100011" ELSE '0'; --Load
34. AluSrc <= I\_Format;
36. Branch <= '1' WHEN Opcode = "000100" ELSE '0'; -- BEQ
38. END behavior;

Segue o código do módulo *Ifetch*:

1. -- fetch module  
   LIBRARY IEEE;
2. USE IEEE.STD\_LOGIC\_1164.ALL; -- Tipo de sinal STD\_LOGIC e STD\_LOGIC\_VECTOR
3. USE IEEE.STD\_LOGIC\_ARITH.ALL; -- Operacoes aritmeticas sobre binarios
4. USE IEEE.STD\_LOGIC\_UNSIGNED.ALL;
5. LIBRARY altera\_mf;
6. USE altera\_mf.altera\_mf\_components.ALL;
7. ENTITY Ifetch IS
8. PORT( reset : in STD\_LOGIC;
9. clock : in STD\_LOGIC;
10. Branch : in STD\_LOGIC;
11. Zero : in STD\_LOGIC;
12. ADDResult : in STD\_LOGIC\_VECTOR(7 DOWNTO 0);
13. PC\_out : out STD\_LOGIC\_VECTOR(7 DOWNTO 0);
14. Instruction : out STD\_LOGIC\_VECTOR(31 DOWNTO 0));
15. END Ifetch;
16. ARCHITECTURE behavior OF Ifetch IS
17. SIGNAL PC : STD\_LOGIC\_VECTOR(7 DOWNTO 0);
18. SIGNAL Next\_PC : STD\_LOGIC\_VECTOR(7 DOWNTO 0);
19. SIGNAL PC\_inc : STD\_LOGIC\_VECTOR(7 DOWNTO 0);
20. SIGNAL Mem\_Addr : STD\_LOGIC\_VECTOR(7 DOWNTO 0);
21. BEGIN
22. --Descricao da Memoria
23. data\_memory: altsyncram -- Declaracao do compomente de memoria
24. GENERIC MAP(
25. operation\_mode => "ROM",
26. width\_a => 32, -- tamanho da palavra (Word)
27. widthad\_a => 8, -- tamanho do barramento de endereco
28. lpm\_type => "altsyncram",
29. outdata\_reg\_a => "UNREGISTERED",
30. init\_file => "program.mif", -- arquivo com estado inicial
31. intended\_device\_family => "Cyclone")
32. PORT MAP(
33. address\_a => Mem\_Addr,
34. q\_a => Instruction,
35. clock0 => clock); -- sinal de clock da memoria
37. -- Descricao do somador
38. PC\_inc <= PC+1;
40. -- Descricao do registrador
41. PROCESS
42. BEGIN
43. WAIT UNTIL (clock'event AND clock='1');
44. IF reset='1' THEN
45. PC <= "00000000";
46. ELSE
47. PC <= Next\_PC;
48. END IF;
49. END PROCESS;
50. -- Usar o Next\_PC ao inves do PC porque a memoria tem um registrador de entrada interno
51. -- Entao o PC tem que ser atualizado simultaneamente com o reg interno da memoria
52. Mem\_Addr <= Next\_PC;
53. Next\_PC <= "00000000" WHEN reset='1' ELSE
54. ADDResult WHEN (Branch = '1' AND Zero = '1') ELSE
55. PC\_inc;
56. PC\_out <= PC;
57. END behavior;

Segue o código do módulo *Idecode*:

1. --decode module  
   LIBRARY IEEE;
2. USE IEEE.STD\_LOGIC\_1164.ALL;
3. USE IEEE.STD\_LOGIC\_ARITH.ALL;
4. USE IEEE.STD\_LOGIC\_UNSIGNED.ALL;
5. ENTITY Idecode IS
6. PORT( read\_data\_1 : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
7. read\_data\_2 : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
8. Instruction : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
9. ALU\_Result : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
10. Data\_Mem : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
11. RegWrite : IN STD\_LOGIC;
12. RegDst : IN STD\_LOGIC;
13. Sign\_extend : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
14. clock,reset : IN STD\_LOGIC;
15. MemToReg : IN STD\_LOGIC;
16. MemAddr : OUT STD\_LOGIC\_VECTOR( 7 DOWNTO 0));
17. END Idecode;
18. ARCHITECTURE behavior OF Idecode IS
19. TYPE register\_file IS ARRAY ( 0 TO 31 ) OF STD\_LOGIC\_VECTOR(31 DOWNTO 0);
21. SIGNAL reg\_bank: register\_file;
22. SIGNAL write\_reg\_ID: STD\_LOGIC\_VECTOR(4 DOWNTO 0);
23. SIGNAL write\_data: STD\_LOGIC\_VECTOR(31 DOWNTO 0);
24. SIGNAL read\_Rs\_ID: STD\_LOGIC\_VECTOR(4 DOWNTO 0);
25. SIGNAL read\_Rt\_ID: STD\_LOGIC\_VECTOR(4 DOWNTO 0);
26. SIGNAL write\_Rd\_ID: STD\_LOGIC\_VECTOR(4 DOWNTO 0);
27. SIGNAL write\_Rt\_ID: STD\_LOGIC\_VECTOR(4 DOWNTO 0);
28. SIGNAL Immediate\_value: STD\_LOGIC\_VECTOR(15 DOWNTO 0);
29. SIGNAL readDataSignal1 : STD\_LOGIC\_VECTOR(31 DOWNTO 0);
30. SIGNAL readDataSignal2 : STD\_LOGIC\_VECTOR(31 DOWNTO 0);
32. BEGIN
33. -- Os sinais abaixo devem receber as identificacoes dos registradores
34. -- que estao definidos na instrucao, ou seja, o indice dos registradores
35. -- a serem utilizados na execucao da instrucao
36. read\_Rs\_ID <= Instruction(25 DOWNTO 21);
37. read\_Rt\_ID <= Instruction(20 DOWNTO 16);
38. write\_Rd\_ID <= Instruction(15 DOWNTO 11);
39. write\_Rt\_ID <= Instruction(20 DOWNTO 16);
40. Immediate\_value <= Instruction(15 DOWNTO 0);
41. readDataSignal1 <= reg\_bank(CONV\_INTEGER(read\_Rs\_ID));
42. readDataSignal2 <= reg\_bank(CONV\_INTEGER(read\_Rt\_ID));
43. -- Os sinais abaixo devem receber o conteudo dos registradores, reg(i)
44. -- USE "CONV\_INTEGER(read\_Rs\_ID)" para converser os bits de indice do registrador
45. -- para um inteiro a ser usado como indice do vetor de registradores.
46. -- Exemplo: dado um sinal X do tipo array de registradores,
47. -- X(CONV\_INTEGER("00011")) recuperaria o conteudo do registrador 3.
48. read\_data\_1 <= readDataSignal1;
49. read\_data\_2 <= readDataSignal2;
51. -- Crie um multiplexador que seleciona o registrador de escrita de acordo com o sinal RegDst
52. write\_reg\_ID <= write\_Rd\_ID WHEN RegDst = '1' ELSE write\_Rt\_ID;
54. -- Ligue no sinal abaixo os bits relativos ao valor a ser escrito no registrador destino.
55. -- adicionar um multiplex que seleciona entre o dado da memoria ou o ULA
56. write\_data <= ALU\_Result WHEN memToReg = '0' ElSE Data\_Mem;
58. -- Estenda o sinal de instrucoes do tipo I de 16-bits to 32-bits
59. -- Faca isto independente do tipo de instrucao, mas use apenas quando
60. -- for instrucao do tipo I.
61. Sign\_extend <= X"0000" & Immediate\_value
62. WHEN Immediate\_value(15) = '0'
63. ELSE X"FFFF" & Immediate\_value;
65. MemAddr <= readDataSignal1( 7 DOWNTO 0) + Immediate\_value (7 DOWNTO 0);
66. PROCESS
67. BEGIN
68. WAIT UNTIL clock'EVENT AND clock = '1';
69. IF reset = '1' THEN
70. -- Inicializa os registradores com seu numero
71. FOR i IN 0 TO 31 LOOP
72. reg\_bank(i) <= CONV\_STD\_LOGIC\_VECTOR( i, 32 );
73. END LOOP;
74. ELSIF (RegWrite = '1' OR MemToReg = '1') AND write\_reg\_ID /= "00000" THEN
75. reg\_bank(CONV\_INTEGER(write\_reg\_ID)) <= write\_data;
77. END IF;
78. END PROCESS;
80. END behavior;

Segue o código da entidade *Execute*:

1. --execute module  
   LIBRARY IEEE;
2. USE IEEE.STD\_LOGIC\_1164.ALL;
3. USE IEEE.STD\_LOGIC\_ARITH.ALL;
4. USE IEEE.STD\_LOGIC\_SIGNED.ALL;
5. ENTITY Execute IS
6. PORT( Read\_data1 : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);
7. Read\_data2 : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);
8. PC : IN STD\_LOGIC\_VECTOR (7 DOWNTO 0);
9. ALU\_Result : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0);
10. Signal\_Ext : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);
11. Alu\_Src : IN STD\_LOGIC;
12. Zero : OUT STD\_LOGIC;
13. ADDResult : OUT STD\_LOGIC\_VECTOR (7 DOWNTO 0));
14. END Execute;
15. ARCHITECTURE behavior OF Execute IS
16. SIGNAL iAux : STD\_LOGIC\_VECTOR(31 DOWNTO 0);
17. BEGIN
19. iAux <= Read\_data2 WHEN Alu\_src = '0' ELSE Signal\_Ext;
20. ALU\_Result <= Read\_data1 + iAux;
22. Zero <= '0' WHEN (Read\_data1 /= Read\_data2) ELSE '1';
24. ADDResult <= PC + 1 + Signal\_Ext (7 DOWNTO 0);
26. --- multiplex ---
27. END behavior;

Por fim, o código da TLE, *Exp03*:

1. LIBRARY IEEE;
2. USE IEEE.STD\_LOGIC\_1164.ALL;
3. ENTITY Exp03 IS
4. PORT( reset : IN STD\_LOGIC;
5. clock48MHz : IN STD\_LOGIC;
6. LCD\_RS, LCD\_E : OUT STD\_LOGIC;
7. LCD\_RW, LCD\_ON : OUT STD\_LOGIC;
8. DATA : INOUT STD\_LOGIC\_VECTOR(7 DOWNTO 0);
9. clock : IN STD\_LOGIC;
10. InstrALU : IN STD\_LOGIC);
11. END Exp03;
12. ARCHITECTURE exec OF Exp03 IS
13. COMPONENT LCD\_Display
14. GENERIC(NumHexDig: Integer:= 11);
15. PORT( reset, clk\_48Mhz : IN STD\_LOGIC;
16. HexDisplayData : IN STD\_LOGIC\_VECTOR((NumHexDig\*4)-1 DOWNTO 0);
17. LCD\_RS, LCD\_E : OUT STD\_LOGIC;
18. LCD\_RW : OUT STD\_LOGIC;
19. DATA\_BUS : INOUT STD\_LOGIC\_VECTOR(7 DOWNTO 0));
20. END COMPONENT;
21. COMPONENT Ifetch
22. PORT( reset : in STD\_LOGIC;
23. clock : in STD\_LOGIC;
24. Branch : in STD\_LOGIC;
25. Zero : in STD\_LOGIC;
26. ADDResult : in STD\_LOGIC\_VECTOR(7 DOWNTO 0);
27. PC\_out : out STD\_LOGIC\_VECTOR(7 DOWNTO 0);
28. Instruction : out STD\_LOGIC\_VECTOR(31 DOWNTO 0));
29. END COMPONENT;
30. COMPONENT Idecode
31. PORT( read\_data\_1 : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
32. read\_data\_2 : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
33. Instruction : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
34. ALU\_Result : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
35. Data\_Mem : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
36. RegWrite : IN STD\_LOGIC;
37. RegDst : IN STD\_LOGIC;
38. Sign\_extend : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
39. clock,reset : IN STD\_LOGIC;
40. MemToReg : IN STD\_LOGIC;
41. MemAddr : OUT STD\_LOGIC\_VECTOR( 7 DOWNTO 0));
42. END COMPONENT;
43. COMPONENT Control
44. PORT( Opcode : IN STD\_LOGIC\_VECTOR( 5 DOWNTO 0 );
45. RegDst : OUT STD\_LOGIC;
46. RegWrite : OUT STD\_LOGIC;
47. MemToReg : OUT STD\_LOGIC;
48. MemRead : OUT STD\_LOGIC;
49. MemWrite : OUT STD\_LOGIC;
50. AluSrc : OUT STD\_LOGIC;
51. Branch : OUT STD\_LOGIC);
52. END COMPONENT;
54. COMPONENT Execute
55. PORT( Read\_data1 : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);
56. Read\_data2 : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);
57. PC : IN STD\_LOGIC\_VECTOR (7 DOWNTO 0);
58. ALU\_Result : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0);
59. Signal\_Ext : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);
60. Alu\_Src : IN STD\_LOGIC;
61. Zero : OUT STD\_LOGIC;
62. ADDResult : OUT STD\_LOGIC\_VECTOR (7 DOWNTO 0));
64. END COMPONENT;
65. COMPONENT Dmemory
66. PORT( read\_data : OUT STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
67. address : IN STD\_LOGIC\_VECTOR( 7 DOWNTO 0 );
68. write\_data : IN STD\_LOGIC\_VECTOR( 31 DOWNTO 0 );
69. MemRead, Memwrite : IN STD\_LOGIC;
70. clock,reset : IN STD\_LOGIC );
71. END COMPONENT;
72. SIGNAL DataInstr : STD\_LOGIC\_VECTOR(31 DOWNTO 0);
73. SIGNAL DisplayData : STD\_LOGIC\_VECTOR(31 DOWNTO 0);
74. SIGNAL PCAddr : STD\_LOGIC\_VECTOR(7 DOWNTO 0);
75. SIGNAL RegDst : STD\_LOGIC;
76. SIGNAL RegWrite : STD\_LOGIC;
77. SIGNAL ALUResult : STD\_LOGIC\_VECTOR(31 DOWNTO 0);
78. SIGNAL SignExtend : STD\_LOGIC\_VECTOR(31 DOWNTO 0);
79. SIGNAL readData1 : STD\_LOGIC\_VECTOR(31 DOWNTO 0);
80. SIGNAL readData2 : STD\_LOGIC\_VECTOR(31 DOWNTO 0);
81. SIGNAL HexDisplayDT : STD\_LOGIC\_VECTOR(43 DOWNTO 0);
82. SIGNAL auxAluSrc : STD\_LOGIC;
83. SIGNAL auxMemWrite : STD\_LOGIC;
84. SIGNAL auxMemToReg : STD\_LOGIC;
85. SIGNAL auxMemRead : STD\_LOGIC;
86. SIGNAL DMemoryOut : STD\_LOGIC\_VECTOR(31 DOWNTO 0);
87. SIGNAL auxMemAddr : STD\_LOGIC\_VECTOR(7 DOWNTO 0);
88. SIGNAL auxBranch : STD\_LOGIC;
89. SIGNAL auxZero : STD\_LOGIC;
90. SIGNAL auxAddResult : STD\_LOGIC\_VECTOR (7 DOWNTO 0);
91. BEGIN
92. LCD\_ON <= '1';
94. -- Inserir MUX para DisplayData
95. displayData <= DataInstr WHEN InstrALU = '1' ELSE AluResult;
97. HexDisplayDT <= "0000" & PCAddr & DisplayData;
98. lcd: LCD\_Display
99. PORT MAP(
100. reset => reset,
101. clk\_48Mhz => clock48MHz,
102. HexDisplayData => HexDisplayDT,
103. LCD\_RS => LCD\_RS,
104. LCD\_E => LCD\_E,
105. LCD\_RW => LCD\_RW,
106. DATA\_BUS => DATA);
108. IFT: Ifetch
109. PORT MAP(
110. reset => reset,
111. clock => clock,
112. PC\_out => PCAddr,
113. Instruction => DataInstr,
114. Branch => auxBranch,
115. Zero => auxZero,
116. AddResult => auxAddResult);
118. CTR: Control
119. PORT MAP(
120. Opcode => DataInstr(31 DOWNTO 26),
121. RegDst => RegDst,
122. RegWrite => RegWrite,
123. MemToReg => auxMemToReg,
124. MemRead => auxMemRead,
125. MemWrite => auxMemWrite,
126. AluSrc => auxAluSrc,
127. Branch => auxBranch);
128. IDEC: Idecode
129. PORT MAP(
130. read\_data\_1 => readData1,
131. read\_data\_2 => readData2,
132. Instruction => DataInstr,
133. ALU\_Result => ALUResult,
134. Data\_Mem => DMemoryOut,
135. RegWrite => RegWrite,
136. RegDst => RegDst,
137. Sign\_extend => SignExtend,
138. clock => clock,
139. reset => reset,
140. MemToReg => auxMemToReg,
141. MemAddr => auxMemAddr);
142. EXE: Execute
143. PORT MAP(
144. Read\_data1 => readData1,
145. Read\_data2 => readData2,
146. ALU\_Result => ALUResult,
147. Signal\_Ext => signExtend,
148. PC => PCAddr,
149. Alu\_Src => auxAluSrc,
150. Zero => auxZero,
151. AddResult => auxAddResult);
152. DMEM: Dmemory
153. PORT MAP(
154. read\_data => DMemoryOut,
155. address => auxMemAddr,
156. write\_data => readData2,
157. MemRead => auxMemRead,
158. Memwrite => auxMemWrite,
159. clock => clock,
160. reset => reset);
161. END exec;

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

# Simulação e teste

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

Figura 2: Conteúdo de program.mif

Figura 3: Conteúdo de DMEMORY.mif

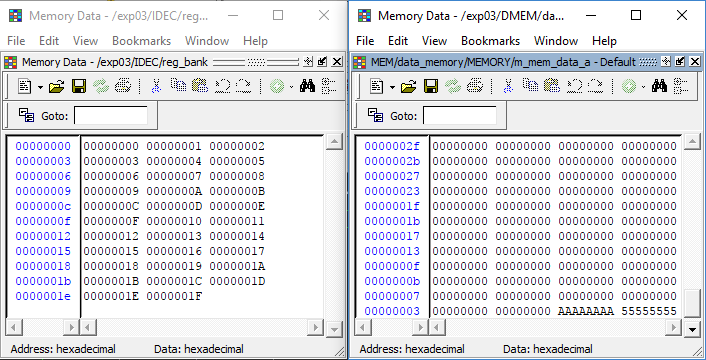


Figura 4: Estado inicial dos registradores e memória de dados

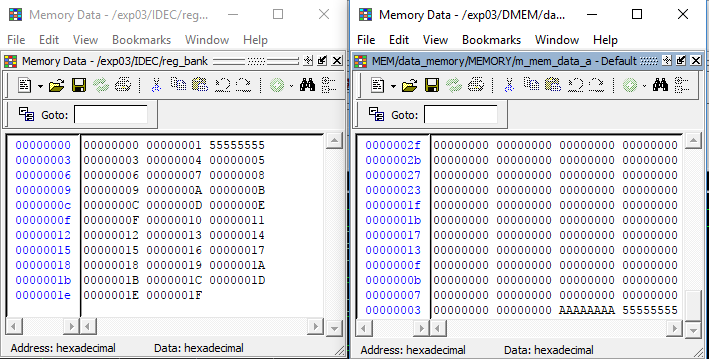


Figura 5: Registradores e memória de dados após 1ª instrução

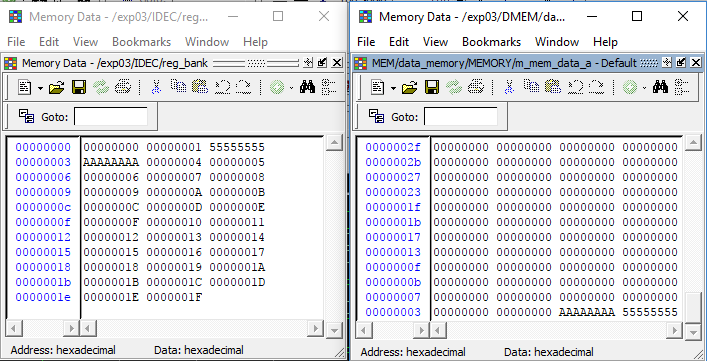


Figura 6: Registradores e memória de dados após 2ª instrução

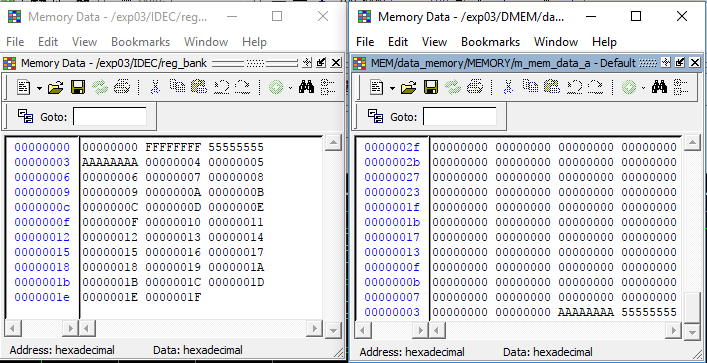


Figura 7: Registradores e memória de dados após 3ª instrução

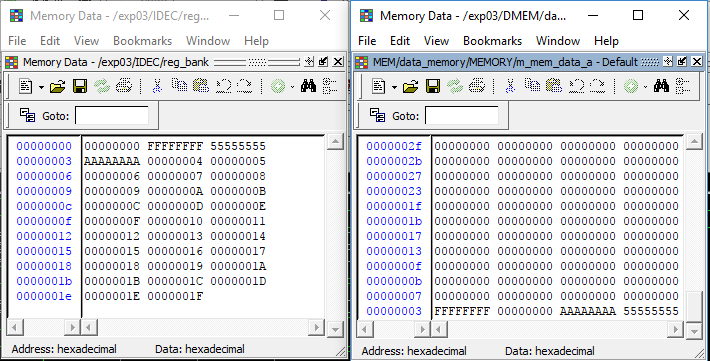


Figura 8: Registradores e memória de dados após 4ª instrução

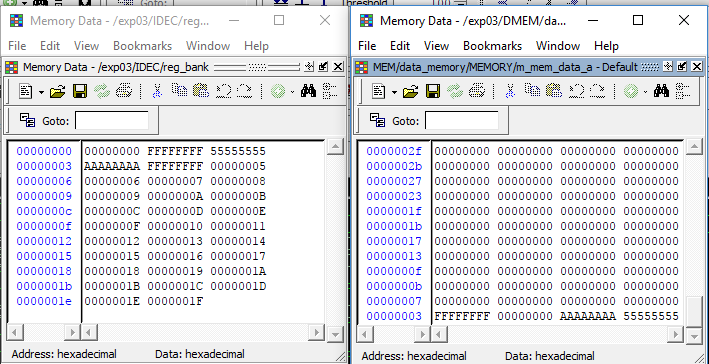


Figura 9: Registradores e memória de dados após 5ª instrução

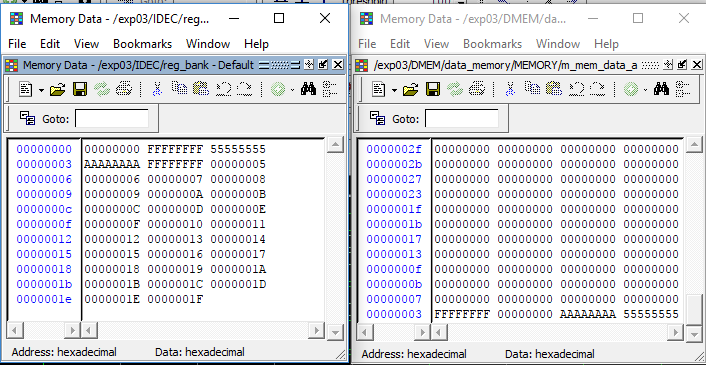


Figura 10: Registradores e memória de dados após 6ª instrução

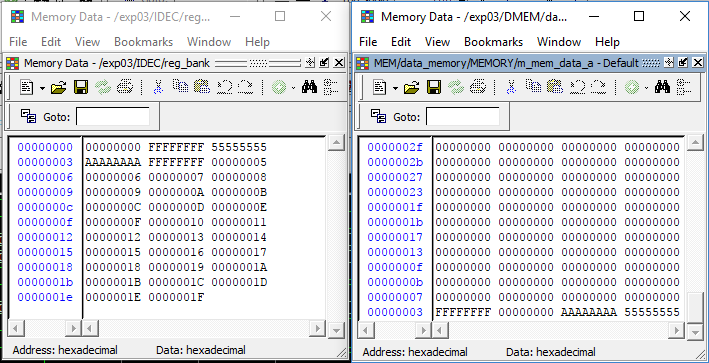


Figura 11: Registradores e memória de dados ficam constantes após a execução da 6ª instrução

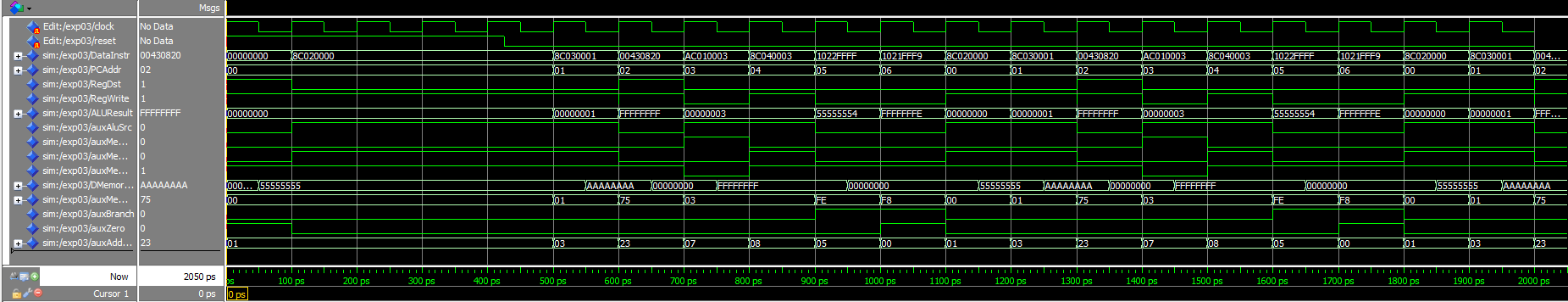


Figura 12: Waves resultantes da simulação

## Discussão

Ao todo, foram realizadas sete instruções no experimento, como indica o arquivo program.mif da Figura 2. O banco de registradores foi inicializado segundo seus índices no módulo de decodificação (*Idecode*), nas linhas 71 e 72 do respectivo código. A Figura 4 representa o estado dos registradores antes da execução das Instruções.

A primeira instrução realiza simplesmente um load da posição zero da memória, no registrador 0x02 ($v0), que agora possui o valor 0x55555555 (Figura 5). Logo após, há outro *load* da posição $zero, mas agora com *offset* de 0x0001, no registrador 0x03 ($v1), com o valor 0xAAAAAAAA (Figura 6).

A terceira instrução faz uma soma entre os valores carregados anteriormente (0xAAAAAAAA + 0x55555555), e o resultado (0xFFFFFFFF) foi armazenado no registrador 0x01 ($at), como mostra a Figura 7. A próxima instrução armazena esse valor na posição 0x03 da memória. (Figura 8).

Coerentemente, as waves de simulação apresentam o resultado da soma (ALUResult) na terceira instrução (PCAddr = 02). O sinal RegWrite é ativo alto nas intruções do tipo R-format e no *load,* e passa a ser ativo baixo na instrução de store (PCAddr = 03), como descrito na unidade de controle (*Control*), nas linhas 28 a 30. Já o sinal RegDst é ativo alto nas instruções de R-format, que, por enquanto, só realiza a soma, como descrito na linha 27.

A instrução 04 é do tipo *I-format* e carrega ao registrador 0x04 ($v2) o conteúdo que está presente no endereço 0x0003 (0xFFFFFFFF). Nota-se que foi obtido o resultado desejado, de acordo com a Figura 9 e a simulação (Figura 12), na qual o sinal *MemToReg* passa a ser alto durante a instrução, como descrito na linha 34 do *Control*, e a variável *DMemoryOut* contém o dado da memória desejado.

As instruções 05 e 06 são ambas do tipo *I-format* e realizam um salto condicional. Durante a instrução 05 é verificado se o conteúdo do registrador 0x01 ($at) é igual ao do registrador 0x02 ($v0) e caso seja, realiza o salto para a própria instrução, pois 0xFFFF corresponde ao valor decimal -1. Na Figura 10 o primeiro registrador contém o valor 0xFFFFFFFF e o segundo 0x55555555, portanto o salto não é realizado e *Next\_PC* recebe *PC\_inc* ao invés de *ADDResult,* como descrito nas linhas 53 a 55 do *Ifetch*. Também é possível evidenciar que durante a instrução 05 o sinal *Branch* é ativo alto, porém o sinal *Zero* é ativo baixo, uma vez que é ativo alto apenas quando os conteúdo dos registradores são iguais, linha 22 do *Execute*. Por outro lado, a instrução 06 realiza uma comparação entre dois registradores idênticos, 0x01 ($at), que contém o valor 0xFFFFFFFF*.* Nota-se na simulação que durante a instrução 06 o sinal *Zero* é ativo alto e como ambos os sinais, *Branch* e *Zero,* são ativos altos, o salto é realizado. O valor 0xFFFA corresponde ao -6 em decimal, isto faz com que o *endereço* passe a ser 00 novamente e o ciclo seja retomado. Por último, vale notar que os registradores e memória de dados permanecem constantes (Figura 11) após a instrução 06.