Leonardo Courbassier Martins

Laboratório de Arquitetura e Organização de Computadores: CoreBassier

São José dos Campos - Brasil

Maio de 2018

Leonardo Courbassier Martins

Laboratório de Arquitetura e Organização de Computadores: CoreBassier

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores.

Docente: Prof. Dr. Sérgio Ronaldo

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Maio de 2018

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

A arquitetura e a organização de um computador é fundamental e única em cada sistema que existe. A partir deles, é possível comparar diferentes processadores e sabendo a filosofia de cada um deles, é possível escolher um que mais se adequa a um determinado projeto.

É indiscutível que o processador é uma das partes mais importantes de um computador e, por isso, deve ser projetada com cautela e com técnicas cada vez mais aprimoradas para construir processadores cada vez mais rápidos e completos.

A mudança do tamanho de computadores é um bom exemplo de como uma boa organização do computador pode fazer diferença; E é necessário, cada vez mais, um desenvolvimento de ténicas para uma tecnologia cada vez mais avançada.

Atualmente, com o crescimento da Internet das Coisas, as diferentes topologias emergentes têm processadores. Isso só reforça o quão importante esse componente computacional é. Com isso, saber projetar um processador para devidos tipos de problemas é fundamental na computação.

2 Objetivos

2.1 Geral

O objetivo desse projeto é desenvolver um processador e testá-lo na placa FPGA. O processador deve ser capaz de rodar qualquer algoritmo já visto nas disciplinas anteriores.

2.2 Específico

Descrição da arquitetura do processador e do conjunto de instruções; Desenvolvimento e implementação da ULA (Unidade Lógica Aritmética), Banco de Registradores, Memória de Dados, módulo Contador de Programa, Unidade de Controle e módulo de Entrada e Saída.

3 Fundamentação Teórica

3.1 Processador

O processador é uma parte de um sistema computacional, que utiliza de um conjunto de instruções para executar a aritmética básica, lógica, e a entrada e saída de dados. O processador tem um papel parecido ao cérebro no computador, pois ele controla todo os periféricos do computador e é o núcleo dele.
Além de saber o para quê serve, uma pergunta mais interessante é: Como funciona um processador?
Como é feita a implementação de um processador? Além dessas perguntas, é necessário conhecer o público-alvo do processador. O processador será projetado para programadores (dando várias instruções diferentes e diversas opções) ou será projetado para usuários comuns? A resposta para essas perguntas dividem o processador em várias técnicas de *design*.
As suas implementações vem mudando drásticamente desde o primeiro processador (Intel 4004, em 1971), assim como a tecnologia também vem desenvolvendo-se, surgiram diversas arquiteturas, organizações, entre outros. Já foram testados várias combinações de organizações até a que compreende-se como a melhor atualmente.

3.2 Arquitetura

Dentre os diversos tipos de arquitetura existentes, as mais conhecidas são as arquiteturas RISC (*Reduced Instruction Set Computer*) e CISC (*Complex Instruction Set Computer*).
A estratégia de *design* RISC foca-se em um número pequeno de instruções, diminuindo o CPI (*Clock per Instruction*) e a complexidade do processador.
Em contrapartida, a estratégia CISC foca-se em um número grande de instruções, o que facilita muito a programação, com várias instruções complexas, e com isso, a complexidade dos programas feitos em máquinas CISC são menores (em relação ao número de instruções). Geralmente, as instruções tem um maior CPI, mas isso é compensado com uma funcionalidade não trivial. Foi escolhido o modelo de arquitetura RISC para este relatório.

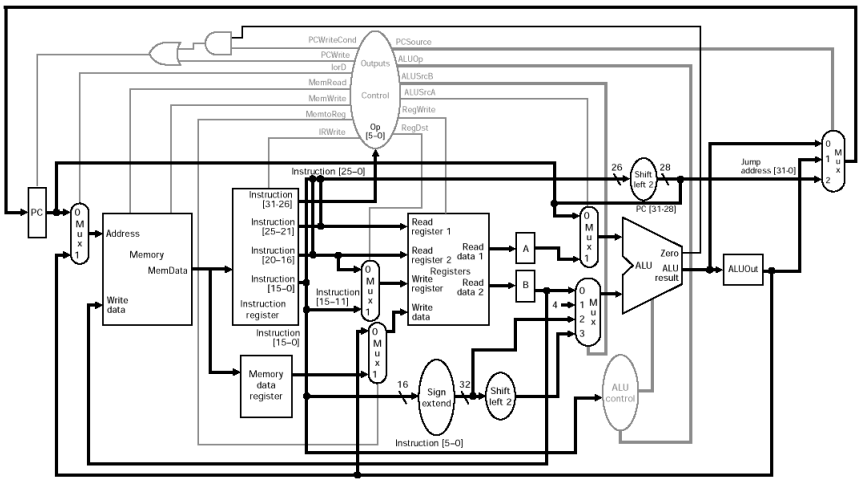
3.3 MIPS

Como referência de processador, foi utilizado o MIPS. O MIPS é um estilo de processador que não tem um único processador físico.
Quando foi comercializado em 1981, era comercializado a ideia do processador e outras empresas faziam a implementação delas. Sua característica principal é ser muito simples e didático e ele foi um grande influenciador das próximas arquiteturas RISC.
A sua implementação com *pipeline* é bem rápida e não é tão complexa, e seus programas não ficam muito complexos.
O MIPS contém 32 registradores em sua implementação e alguns deles são reservados para o sistema. Além disso ele possui uma unidade de controle que facilita todo o controle do processador e seus componentes.

3.3.1 Arquitetura do MIPS

Na Figura 1, vemos o esquemático da arquitetura MIPS.
Como o MIPS é RISC, vemos que sua arquitetura é simples de ser entendida. Cada módulo na figura tem um papel específico, como descrito a seguir:
O módulo do PC tem a função de atualizar o Contador de Programa para buscar a próxima instrução do programa atual. Esse endereço entra no módulo da memória de Instruções, onde a próxima instrução é buscada. A instrução atual é decodificada então, e as *flags* são definidas pela Unidade de Controle. Os operadores da instrução são carregados do Banco de Registradores e as entradas corretas são enviadas para a Unidade Lógica e Aritmética para realizar a operação explicitado pela instrução. O resultado da ALU é escrito no Banco de Registradores ou na Memória de Dados, de acordo com a instrução atual. Após isso o valor do PC é atualizado e o ciclo continua.

Figura 1 – Esquematico do MIPS



Fonte: Computer Organization and Design (1)

3.3.2 Tipos de Instrução

O MIPS possui três tipos de instruções que são caracterizados como: Tipo R, que é ilustrado na Tabela 1:

Tabela 1 – Formato das Instruções do Tipo R

Campo	OpCode	RS	RT	RD	Shamt	Funct
Tamanho (bits)	6	5	5	5	5	6

Fonte: O Autor

E a seguir são descritas as instruções do tipo R: add, addu, and, div, jr, mfhi, mflo, mfc0, mult, xor, or, sltu, sll, sra, sub.

Além do tipo R, existe o tipo I que é ilustrado na Tabela 2

Tabela 2 – Formato das Instruções do Tipo I

Campo	OpCode	RS	RT	Im
Tamanho (bits)	6	5	5	16

Fonte: O Autor

São instruções do tipo I as seguintes: addi, addiu, andi, beq, blez, bne, lbu, lhu, lui, lw, ori, sb, sh, slti, sltiu, sw.

Por último, existe o tipo J que é ilustrado na Tabela 3

São instruções do tipo J as seguintes: j, jal.

Tabela 3 – Formato das Instruções do Tipo J

Campo	OpCode	Address
Tamanho (bits)	6	26

Fonte: O Autor

3.3.3 Modos de endereçamento

Descritas mais profundamente no (1) e, brevemente em (2), o MIPS tem vários modos de endereçamento e são utilizados em diferentes instruções.
Abaixo está descrito os modos de endereçamento do MIPS, qual seu uso e como funciona.

- Endereçamento por registrador: É utilizado esse modo de endereçamento quando os operandos estão em registradores, p.e. add;
- Endereçamento por imediato: É utilizado quando deve-se somar o valor de um registrador com um número definido como imediato, p.e. addi;
- Endereçamento de base-deslocamento: É utilizado quando o operando é soma de um imediato com um registrador, p.e. lw;
- Endereçamento relativo ao PC: É utilizado quando o dado é a soma do imediato com a posição atual do PC, p.e. beq;
- Endereçamento absoluto: É utilizado quando o dado é passado diretamente pela instrução, p.e. j;

4 Desenvolvimento

Essa seção mostra os pontos principais da implementação do processador. O processador aqui aprensetado é monociclo, sem *pipeline* e RISC. Contém 28 instruções (discutidas nas próximas seções) e 5 tipos de formatos de instruções.
Além disso, ele possui 32 registradores de 32 bits, mas apenas alguns de próposito geral.

4.1 Formato das Instruções

Antes da implementação do processador, é necessário definir os formatos de instruções que será utilizado. A Tabela 4 contém todos os 5 formatos, divindo os bits e os campos de cada formato.

Tabela 4 – Formato das Instruções do CoreBassier

Tipo	**Campos**
1	[OpCode / RegDest / RegOrigem / RegAlvo / |] [31:27] [26:22] [21:17] [16:12] [11:0]
2	[OpCode / RegDest / RegOrigem / Imediato (ou |)] [31:27] [26:22] [21:17] [16:0]
3	[OpCode / RegDest / RegOrigem (ou |) / Imediato (ou |)] [31:27] [26:22] [21:17] [16:0]
4	[OpCode / RegDest / Imediato] [31:27] [26:22] [21:0]
5	[OpCode / |] [31:27] [26:0]

Fonte: O Autor

O tipo 1 destina-se a operações aritméticas com três registradores e sem imediato. Já o tipo 2, destina-se a operações aritméticas com dois registradores e um imediato. O tipo 3, destina-se aos desvios de fluxos condicionais e incondicionais. O tipo 4, destina-se as instruções de entrada e saída, comunicação que será desenvolvida mais adiante (mais sobre isso nos próximos tópicos). Já o tipo 5, destina-se as instruções de controle do processador.
É fácil perceber que o formato das instruções aqui ilustrados para o CoreBassier é muito semelhante aos formatos do MIPS.
A maior diferença entre os dois processadores é que o MIPS utiliza do Shamt e Funct para as instruções e o outro não, foram criados mais formatos de instruções para deixar mais modular e mais simples de se entender.

4.2 Conjunto de Instruções

A Tabela 5 contém todo o conjunto de instruções que o processador terá, os tipos das instruções e seu opCode.

Tabela 5 – Conjunto de Instruções do CoreBassier

OpCode	**Instrução**	**Tipo**
00000	Add	1
00001	Addi	2
00010	Sub	1
00011	Subi	2
00100	Mult	1
00101	Multi	2
00110	Div	1
00111	Divi	2
01000	Mod	1
01001	Slt	1
01010	Slti	2
01011	And	1
01100	Andi	2
01101	Or	1
01110	Ori	2
01111	Not	2
10000	ShR	2
10001	ShL	2
10010	Load	4
10011	Loadi	4
10100	Store	4
10101	Jump	3
10110	Beq	3
10111	Bne	3
11000	Nop	5
11001	Halt	5
11010	In	4
11011	Out	4
11100	Mov	2

Fonte: O Autor

A diferença entre o CoreBassier e o MIPS em relação ao seus conjuntos de instruções é o fato de o CoreBassier possuir menos instruções, não lidar com números *unsigned* como o MIPS faz. Além disso, enquanto o MIPS possui uma única instrução para carregar uma palavra da memória de dados, o CoreBassier já possui duas, separando o caso especial quando o imediato é zero.

4.3 Registradores

Como no MIPS, o processador também utilizará de 32 registradores e alguns deles serão limitados ao processador. A Tabela 6 descreve todos os registradores e suas funções.

Não há diferença entre os registradores do MIPS e do CoreBassier.

Tabela 6 – Registradores do CoreBassier

Nome	Número	Descrição
$0	00	$zero: Constante zero
$1	01	$at: Reservado
$2	02	$v0: Resultado de uma função
$3	03	$v1: Resultado de uma função
$4	04	$a0: Argumento para uma função
$5	05	$a1: Argumento para uma função
$6	06	$a2: Argumento para uma função
$7	07	$a3: Argumento para uma função
$8	08	$t0: Uso geral
$9	09	$t1: Uso geral
$10	0A	$t2: Uso geral
$11	0B	$t3: Uso geral
$12	0C	$t4: Uso geral
$13	0D	$t5: Uso geral
$14	0E	$t6: Uso geral
$15	0F	$t7: Uso geral
$16	10	$s0: Uso geral (salvo em chamadas de função)
$17	11	$s1: Uso geral (salvo em chamadas de função)
$18	12	$s2: Uso geral (salvo em chamadas de função)
$19	13	$s3: Uso geral (salvo em chamadas de função)
$20	14	$s4: Uso geral (salvo em chamadas de função)
$21	15	$s5: Uso geral (salvo em chamadas de função)
$22	16	$s6: Uso geral (salvo em chamadas de função)
$23	17	$s7: Uso geral (salvo em chamadas de função)
$24	18	$t8: Uso geral
$25	19	$t9: Uso geral
$26	1A	$k0: Reservado
$27	1B	$k1: Reservado
$28	1C	$gp: Ponteiro global
$29	1D	$sp: Ponteiro da pilha
$30	1E	$fp: Ponteiro do frame
$31	1F	$ra: Endereço de retorno

Fonte: O Autor

4.4 Modos de endereçamento

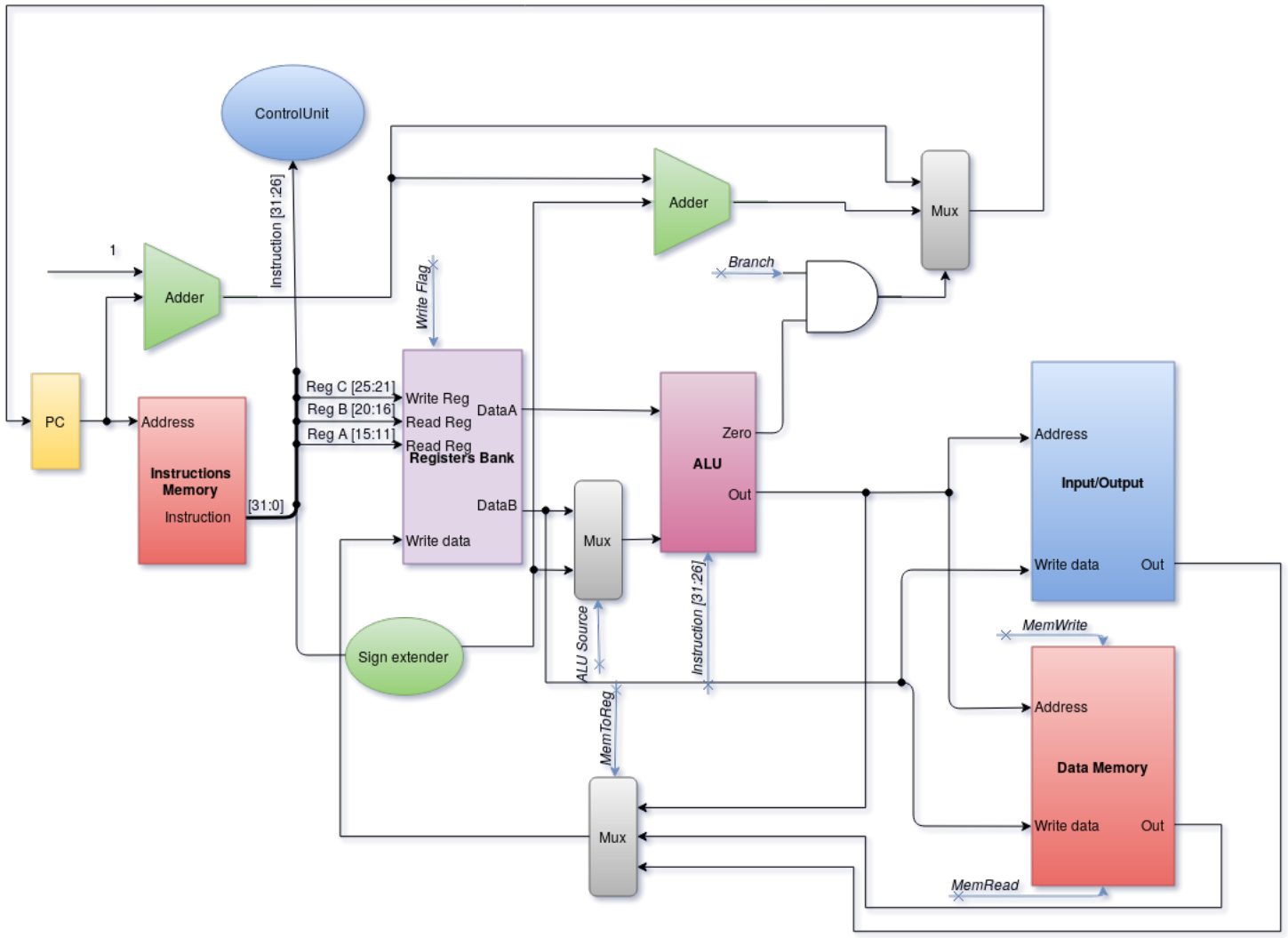
Os modos de endereçamento do CoreBassier são exatamente os mesmos que o do MIPS.

- Endereçamento por registrador: Formato dos tipos 1 e 4;
- Endereçamento por imediato: Formato do tipo 2;
- Endereçamento de base-deslocamento: Formato dos tipos 1 e 2;
- Endereçamento relativo ao PC: Formato do tipo 3;
- Endereçamento absoluto: Formato do tipo 3;

4.5 Esquemático

A Figura 2 mostra o esquemático da organização do CoreBassier.
No esquemático abaixo, cada módulo está descrito (brevemente) com suas respectivas entradas e saídas. Assim como é no MIPS, o datapath do CoreBassier é o mesmo.
O PC é o Contador de Programa e armazena o endereço da instrução atual, essa instrução atual é buscada na Memória de Instruções e então decodificada. A Unidade de Controle recebe a instrução e envia todos os sinais – descritos em azul no esquemático – para controlar todos os módulos.
A diferença para o MIPS está no fato do CoreBassier possuir um módulo de Entrada e Saída que faz a comunicação da placa FPGA (seus periféricos) e o processador.

Figura 2 – Esquemático do CoreBassier



Fonte: O Autor

Todos os módulos foram implementados no Quartus II (3), utilizando o Verilog.

4.6 Módulos em Verilog

Abaixo temos o código fonte do *datapath* do processador e outros módulos que serão importantes para mostrar os resultados.
Esse módulo é responsável por instanciar todos os outros módulos e fazer as ligações necessárias entre eles, além disso ele substitui alguns muxes que estão presentes no esquemático para a simplificação do processador.

Observação: Nem todos os sinais são saídas nesse módulo, isso foi feito para a visualização no *Waveform* do Quartus II.

```
1  module main(
2                                clock, sw0, sw1, sw2, sw3, sw4, sw5, sw6, sw7, sw8, sw9,
                                     sw10, sw11, sw12, sw13, sw14, sw15, sw16, sw17,
3                                key0, key1, key2, key3,
4                                hex0, hex1, hex2, hex3, hex4, hex5, hex6, hex7, halt
5      , _programCounter,
6      , instructionOut,
7          , aluCode,
8      , targetRegister, aluSource,
9      , writeRegister, memoryWrite, memoryRead, memoryToRegister, branch,
10     , intermediate,
11     , branchSignal,
12     , aluBranchSignal,
13     , dataA, dataB,
14     , dataToWrite,
15         , aluSource1, aluSource2,
16     , aluError,
17     , aluOut,
18     , error, overflow,
19     , memoryOut);
20
21     // Inputs
22     input clock;
23     input sw0, sw1, sw2, sw3, sw4, sw5, sw6, sw7, sw8, sw9, sw10, sw11, sw12, sw13, sw14,
           sw15, sw16, sw17;
24     input key0, key1, key2, key3;
25
26     // Temporary
27     output wire halt;
28     output wire [9:0] _programCounter;
29         output wire [31:0] instructionOut;
30         output wire [5:0] aluCode;
31     output wire targetRegister, aluSource;
32     output wire writeRegister, memoryWrite, memoryRead, memoryToRegister, branch;
33     output reg[31:0] intermediate;
34     output reg branchSignal;
35     output wire aluBranchSignal;
36     output wire [31:0] dataA, dataB;
37     output reg [31:0] dataToWrite;
38         output reg [31:0] aluSource1, aluSource2;
```

```
39      output wire aluError;
40      output wire [31:0] aluOut;
41      output wire error, overflow;
42      output wire [31:0] memoryOut;
43
44      // Outputs
45      output [6:0] hex0, hex1, hex2, hex3, hex4, hex5, hex6, hex7;
46
47      // Instances
48      PC programCounter(.clock(clock), .halt(halt), .intermediate(intermediate), .
            branchSignal(branchSignal), .programCounter(_programCounter));
49      InstructionsMemory instructionsMemory(.read_addr(_programCounter), .clk(clock), .q(
            instructionOut));
50      ControlUnit controlUnit(.clock(clock), .opcode(instructionOut[31:26]), .aluCode(
            aluCode), .targetRegister(targetRegister), .aluSource(aluSource),
51          .writeRegister(writeRegister), .memoryWrite(memoryWrite), .memoryRead(memoryRead)
                , .memoryToRegister(memoryToRegister), .branch(branch), .halt(halt));
52      RegistersBank registersBank(.regC(instructionOut[25:21]), .regB(instructionOut
            [20:16]), .regA(instructionOut[15:11]), .writeFlag(writeRegister), .clock(clock),
             .dataA(dataA), .dataB(dataB), .dataWrite(dataToWrite));
53      ALU alu(.dataA(aluSource1), .dataB(aluSource2), .opCode(instructionOut[31:26]), .
            branchSignal(aluBranchSignal), .overflow(overflow), .error(error), .dataC(aluOut)
            , .clock(clock));
54      DataMemory dataMemory(.data(dataB), .read_addr(aluOut), .write_addr(aluOut), .we(
            memoryWrite), .clk(clock), .q(memoryOut));
55      IO io(.opcode(instructionOut[31:26]), .address(aluOut), .sw0(sw0), .sw1(sw1), .sw2(
            sw2),
56       .sw3(sw3), .sw4(sw4), .sw5(sw5), .sw6(sw6), .sw7(sw7), .sw8(sw8), .sw9(sw9),
57        .sw10(sw10), .sw11(sw11), .sw12(sw12), .sw13(sw13), .sw14(sw14), .sw15(sw15), .sw16
              (sw16),
58          .sw17(sw17), .key0(key0), .key1(key1), .key2(key2), .outL(inputOut), .hex0(hex0),
59          .hex1(hex1),  .hex2(hex2),  .hex3(hex3),  .hex4(hex4),  .hex5(hex5),  .hex6(hex6)
                ,
60           .hex7(hex7), .data(dataB));
61
62      always @ ( * )
63      begin
64                  intermediate[31:0] = { {16{instructionOut[15]}}, instructionOut[15:0] };
65          branchSignal = aluBranchSignal & branch;
66
67          aluSource1 = dataB;
68
69          /* Mux for ALU input */
70          case (aluSource)
71              1'b0: aluSource2 = dataA;
72              1'b1: aluSource2 = intermediate;
73          endcase
74
75          /* Mux for writing data to register */
76          case (memoryToRegister)
77              3'b000: dataToWrite = aluOut;
78              3'b001: dataToWrite = memoryOut;
79              3'b010: dataToWrite = inputOut;
80          endcase
81      end
82
83      // Out: convert the register data to needed algarisms
84  endmodule
```

O módulo abaixo é a Memória de Instruções, ela está carregada com seis instruções que são: addi, addi, addi, addi, out e halt. Isso será importante para a discussão dos resultados.

```
1  module InstructionsMemory
2  #(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=10)
3  (
4          input [(ADDR_WIDTH-1):0] read_addr,
5          input clk,
6          output reg [(DATA_WIDTH-1):0] q
7  );
8
9          // Declare the RAM variable
10         reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
11
12         initial begin
13                 // Addi reg[0x1], reg[0x0] + 0x5
14                 ram[0] = 32'b00000100001000000000000000000101;
15                 // Addi reg[0x1], reg[0x1] + 0x1
16                 ram[1] = 32'b00000100001000010000000000000001;
17                 // Addi reg[0x1], reg[0x1] + 0x1
18                 ram[2] = 32'b00000100001000010000000000000001;
19                 // Addi reg[0x1], reg[0x1] + 0x1
20                 ram[3] = 32'b00000100001000010000000000000001;
21                 // Out hex[0x0] <= reg[0x1]
22                 ram[4] = 32'b01110000000000000000000000000001;
23                 // Halt
24                 ram[5] = 32'b01101000000000000000000000000000;
25         end
26
27         always @ (posedge clk)
28         begin
29                 q <= ram[read_addr];
30         end
31 endmodule
```

O módulo abaixo é a Unidade de Controle. O que importa nela é apenas o detalhe da linha 32, onde é utilizado o *negedge clock*. Isso será explicado mais adiante.

```
1  module ControlUnit(clock,
2                     opcode,
3                     aluCode,
4                     targetRegister,
5                     aluSource,
6                     writeRegister,
7                     memoryWrite,
8                     memoryRead,
9                     memoryToRegister,
10                    branch,
11                    halt);
12
13     // Inputs
14     input clock;
15     input [5:0] opcode;
16     // Wait for in | out
17
18     // Temporary
19     reg _halt;
20
```

```
21      // Outputs
22      output reg[5:0] aluCode;
23      output reg targetRegister, aluSource, writeRegister, memoryWrite, memoryRead;
24      output reg[2:0] memoryToRegister;
25      output reg branch;
26           output halt;
27
28           initial begin
29                  _halt = 0;
30           end
31
32      always @ (negedge clock)
33      begin
34          if (!_halt)
35          begin
36              case (opcode)
37                  // Add
38                  6'b000000,
39                  // Sub
40                  6'b000010,
41                  // Mult
42                  6'b000100,
43                  // Div
44                  6'b000110,
45                  // Mod
46                  6'b001000,
47                  // SetLesserThan
48                  6'b001001,
49                  // And
50                  6'b001011,
51                  // Or
52                  6'b001101,
53                  // Not
54                  6'b001111,
55                  // Shift Right
56                  6'b010000,
57                  // Shift Left
58                  6'b010001,
59                  // SetGreaterThan
60                  6'b010010:
61                  begin
62                      aluCode = opcode;
63                      targetRegister = 1;
64                      aluSource = 0;
65                      writeRegister = 1;
66                      memoryWrite = 0;
67                      memoryRead = 0;
68                      memoryToRegister = 0;
69                      branch = 0;
70                      _halt = 0;
71                  end
72
73                  // Addi
74                  6'b000001,
75                  // Subi
76                  6'b000011,
77                  // Multi
78                  6'b000101,
79                  // Divi
80                  6'b000111,
```

```
81                  // SetLesserThanImediate
82                  6'b001010,
83                  // Andi
84                  6'b001100,
85                  // Ori
86                  6'b001110,
87                  // SetGreaterThanImediate
88                  6'b010011:
89                  begin
90                      aluCode = opcode;
91                      targetRegister = 1;
92                      aluSource = 1;
93                      writeRegister = 1;
94                      memoryWrite = 0;
95                      memoryRead = 0;
96                      memoryToRegister = 0;
97                      branch = 0;
98                      _halt = 0;
99                  end
100
101                 // Load, Store
102                 6'b010100, 6'b010101:
103                 begin
104                     aluCode = 6'b000000;
105                     targetRegister = 0;
106                     aluSource = 1;
107                     writeRegister = opcode == 6'b010100 ? 1 : 0;
108                     memoryWrite = opcode == 6'b010100 ? 0 : 1;
109                     memoryRead = opcode == 6'b010100 ? 1 : 0;
110                     memoryToRegister = 1;
111                     branch = 0;
112                     _halt = 0;
113                 end
114
115                 // Jump
116                 6'b010110:
117                 begin
118                     aluCode = opcode;
119                     targetRegister = 0;
120                     aluSource = 0;
121                     writeRegister = 0;
122                     memoryWrite = 0;
123                     memoryRead = 0;
124                     memoryToRegister = 0;
125                     branch = 1;
126                     _halt = 0;
127                 end
128
129                 // Branch on (NOT) Equal
130                 6'b010111, 6'b011000:
131                 begin
132                     aluCode = opcode;
133                     targetRegister = 0;
134                     aluSource = 0;
135                     writeRegister = 0;
136                     memoryWrite = 0;
137                     memoryRead = 0;
138                     memoryToRegister = 0;
139                     branch = 1;
140                     _halt = 0;
```

```
141                 end
142
143                 // Nop
144                 6'b011001:
145                 begin
146                     aluCode = opcode;
147                     targetRegister = 0;
148                     aluSource = 0;
149                     writeRegister = 0;
150                     memoryWrite = 0;
151                     memoryRead = 0;
152                     memoryToRegister = 0;
153                     _halt = 0;
154                     branch = 0;
155                 end
156
157                 // _halt
158                 6'b011010:
159                 begin
160                     aluCode = opcode;
161                     targetRegister = 0;
162                     aluSource = 0;
163                     writeRegister = 0;
164                     memoryWrite = 0;
165                     memoryRead = 0;
166                     memoryToRegister = 0;
167                     _halt = 1;
168                     branch = 0;
169                 end
170
171                 // In
172                 6'b011011:
173                 begin
174                     aluCode = opcode;
175                     targetRegister = 0;
176                     aluSource = 0;
177                     writeRegister = 1;
178                     memoryWrite = 0;
179                     memoryRead = 0;
180                     memoryToRegister = 2'b10;
181                     _halt = 0;
182                     branch = 0;
183                 end
184
185                 // Out
186                 6'b011100:
187                 begin
188                     aluCode = opcode;
189                     targetRegister = 0;
190                     aluSource = 0;
191                     writeRegister = 0;
192                     memoryWrite = 0;
193                     memoryRead = 0;
194                     memoryToRegister = 0;
195                     _halt = 0;
196                     branch = 0;
197                 end
198
199                 // Mov
200                 6'b011101:
```

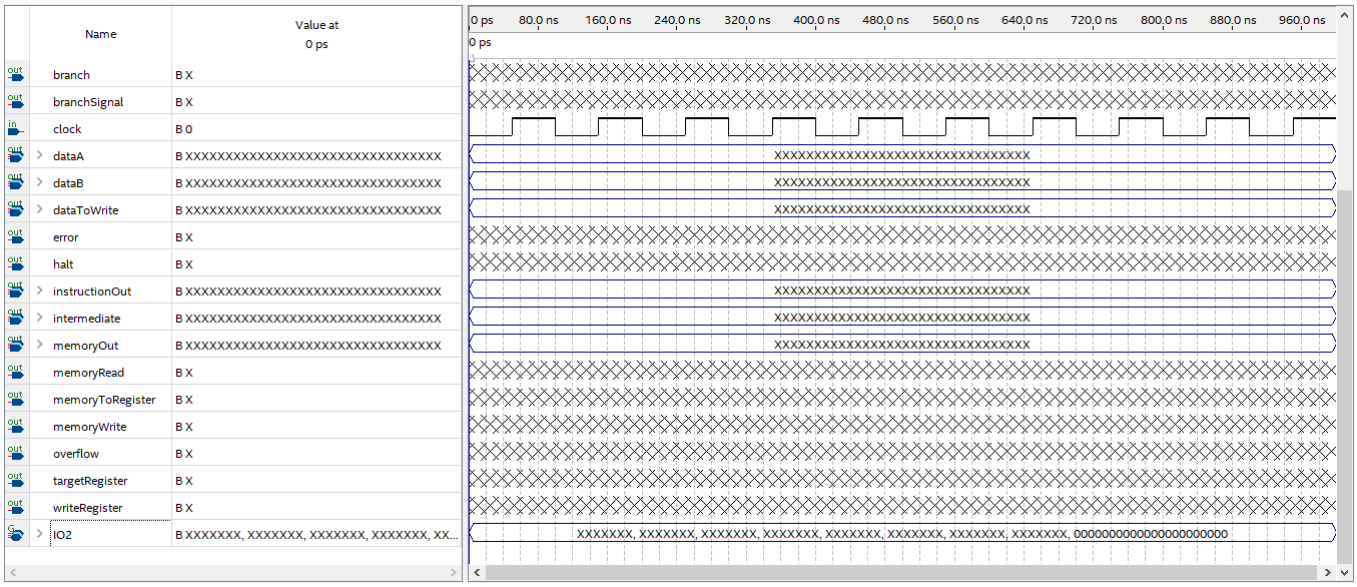
```
201                 begin
202                     aluCode = opcode;
203                     targetRegister = 0;
204                     aluSource = 0;
205                     writeRegister = 1;
206                     memoryWrite = 0;
207                     memoryRead = 0;
208                     memoryToRegister = 0;
209                     _halt = 0;
210                     branch = 0;
211                 end
212             endcase
213         end
214     end
215 
216     assign halt = _halt;
217     // Wrong case or beq
218 endmodule
```

5 Resultados Obtidos e Discussões

O CoreBassier ainda não está completo. Ele está em fase de desenvolvimento, principalmente o módulo de IO.
Após desenvolvido os módulos, chegou a hora de realizar os testes. Utilizando as instruções que foram colocadas na Memória de Instruções, foi realizado o teste utilizando o *Waveform* do Quartus II. A Figura 3 mostra o primeiro momento após rodado a simulação.

A figura detalha todos os sinais, os de controle: branch, branchSignal (controle de desvio), halt (controle de parada), error (erros), overflow (estouro nas operações aritméticas), targetRegister (*flag* para saber qual registrador deve-se escrever), writeRegister (se deve ou não escrever num registrador), memoryToRegister (se está a carregar um dado da Memória de Dados para o registrador), memoryWrite (se deve ou não escrever na Memória de Dados), memoryRead (se pode ler a Memória de Dados); e os intermediários. O grupo IO2 deve ser ignorado por enquanto.

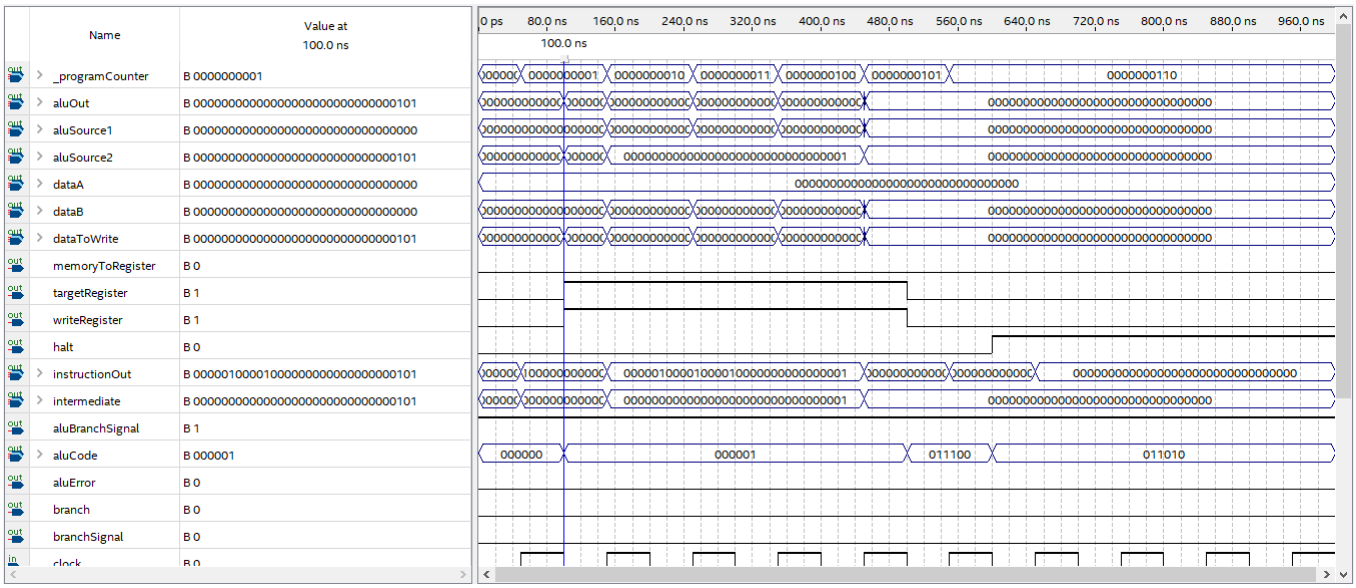
Figura 3 – Antes do primeiro *clock*



Fonte: O Autor

A Figura 4 mostra os sinais após o período de 1 *clock*.
A primeira instrução é o addi, então percebe-se que o imediato foi buscado da própria instrução (*InstructionOut*) e foi armazenado em um outro sinal. A Unidade de Controle colocou todos as *flags* para seus valores corretos para a soma ser realizada pela Unidade Lógica e Aritmética e depois ser armazenada no próprio registrador.

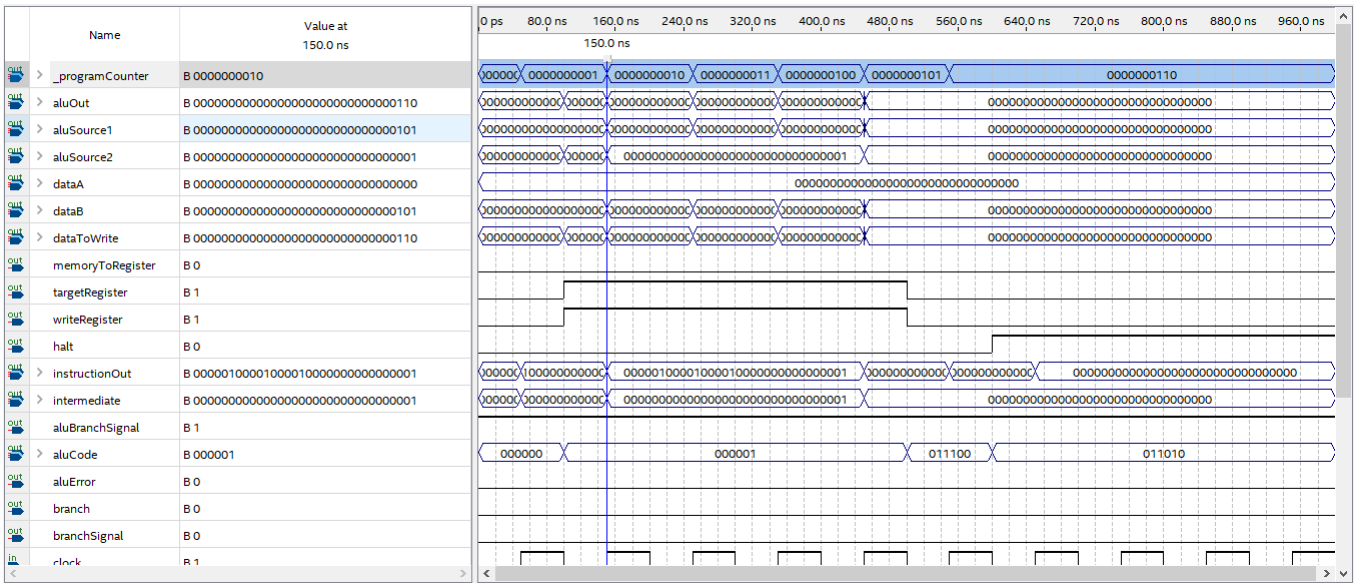
Figura 4 – Resultado do primeiro *clock*



Fonte: O Autor

A Figura 5 mostra os sinais após o período de 2 *clock*.
A segunda instrução é o addi, então percebe-se que o imediato foi buscado da própria instrução (*InstructionOut*) e foi armazenado em um outro sinal. A Unidade de Controle colocou todos as *flags* para seus valores corretos para a soma ser realizada pela Unidade Lógica e Aritmética e depois ser armazenada no próprio registrador.

Figura 5 – Resultado do segundo *clock*

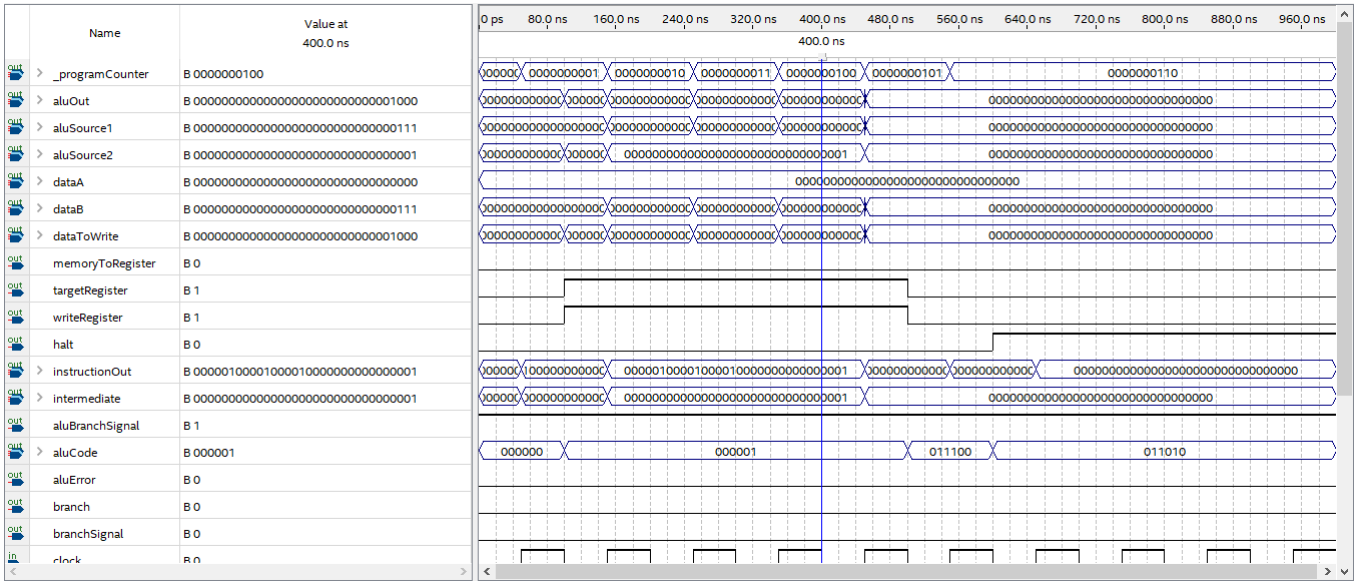


Fonte: O Autor

Isso será repetido mais algumas vezes até chegar na Figura 6.

Após repetir isso mais duas vezes, o resultado final é 1000_2 que exatamente o esperado. Após isso, o próximo comando é o out, porém ele não está funcionando corretamente.

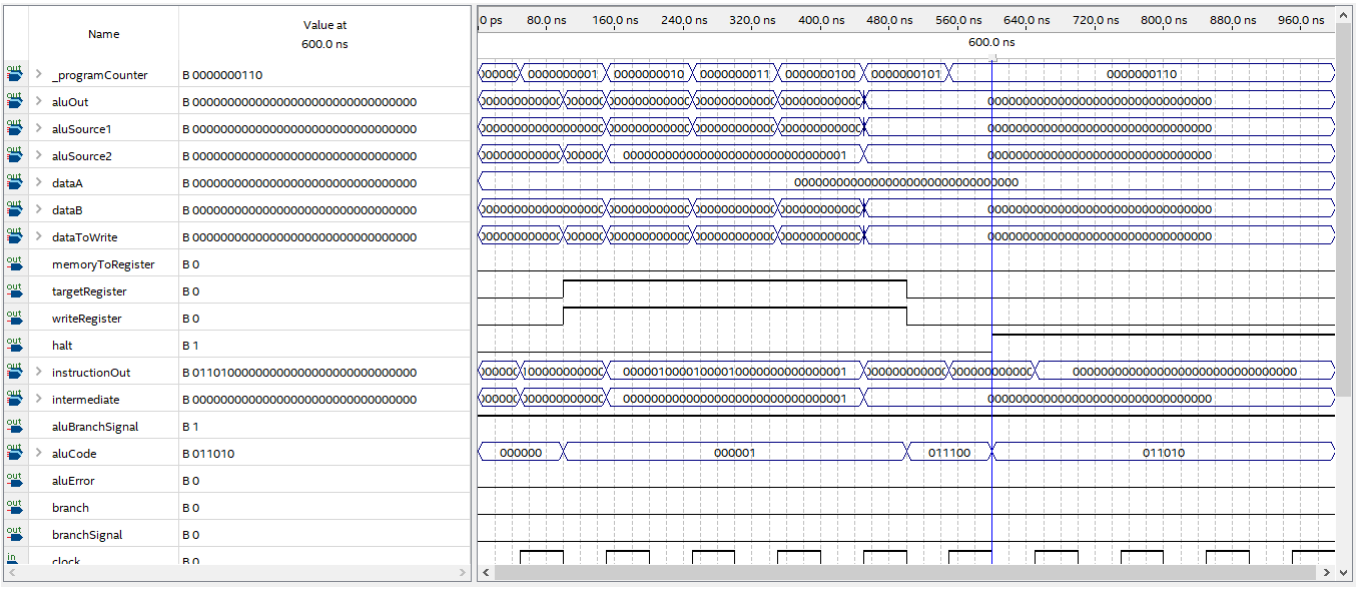
Figura 6 – Resultado do último addi



Fonte: O Autor

E por fim, a Figura 7 mostra a última instrução que é o halt. Após isso o processador não faz mais nada.

Figura 7 – Resultado do Halt



Fonte: O Autor

Por que foi importante mostrar a Unidade de Controle?

O que está acontecendo é que o Contador de Programa atualiza seu valor na subida do *clock*, e ao mesmo tempo, na subida do *clock*, a instrução é decodificada. Porém apenas na descida do *clock* as *flags* são configuradas corretamente, isso garante que a instrução correta já foi decodificada e que os operandos serão buscados na próxima subida.
Na próxima subida então, os operandos são buscados e a Unidade Lógica e Aritmética faz a operação que a instrução precisa que ela faça, e nessa mesma subida, o Contador de Programa é atualizado e a próxima instrução decodificada. Isso dá tempo suficiente para o resultado ser escrito de volta no Banco de Registradores ou na Memória de Dados antes que as próximas *flags* sejam configuradas pela Unidade de Controle.
Com isso, é garantido que o resultado correto é escrito.

6 Considerações Finais

A construção de um processador é um processo muito complexo devido as várias considerações de *design* que são necessárias. A maior das escolhas é entre seguir a arquitetura RISC ou CISC, ambas tem pontos positivos muito interessantes (como discutido nas primeiras seções) e que facilitam em algum momento da construção do processador. Seja no começo com a implementação física dele (no caso do RISC) ou depois, na utilização do processador (no caso do CISC).

A maior dificuldade encontrada para projetar esse processador foi conseguir passar toda a ideia para o Verilog. Ter que pensar nos tempos que cada módulo precisa para executar e garantir que o *datapath* esteja correto. Por isso foi tão importante os *negedge clock*. Isso é uma particularidade do *design* via Quartus e Verilog, pois numa implementação física deve-se pensar mais ainda nos tempos – pois no Verilog, uma multiplicação pode durar apenas um *clock*, enquanto quando é projetado fisicamente, demora muito mais.

O próximo passo é arrumar o módulo de IO e fazer o teste na placa FPGA depois que tudo estiver pronto. O módulo está implementado, mas ainda não está correto. Além disso, faltam outras combinações de instruções para serem testadas. A combinação testada primeiro era apenas somas e um halt, falta testar *branch* e *load* e *store*.

O resultado gerado atendeu as expectativas e objetivo do relatório, mostrando como é importante o estudo de arquitetura e organização de computadores.

Referências

1 PATTERSON, D. A.; HENNESY, J. L. *Computer Organization and Design*. 5th edition. ed. Waltham/MA, EUA: Morgan Kaufmann, 2007. Citado 2 vezes nas páginas 11 e 12.

2 BLAS, D. A. D. *MIPS addressing modes*. Disponível em: <https://classes.soe.ucsc.edu/cmpe110/Spring11/lectures/05_MIPS_addressing.pdf>. Citado na página 12.

3 ALTERA. *Quartus II*. Disponível em: <https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html>. Citado na página 18.