Leonardo Courbassier Martins

# Laboratório de Arquitetura e Organização de Computadores: CoreBassier

São José dos Campos - Brasil

Julho 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

Julho 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.

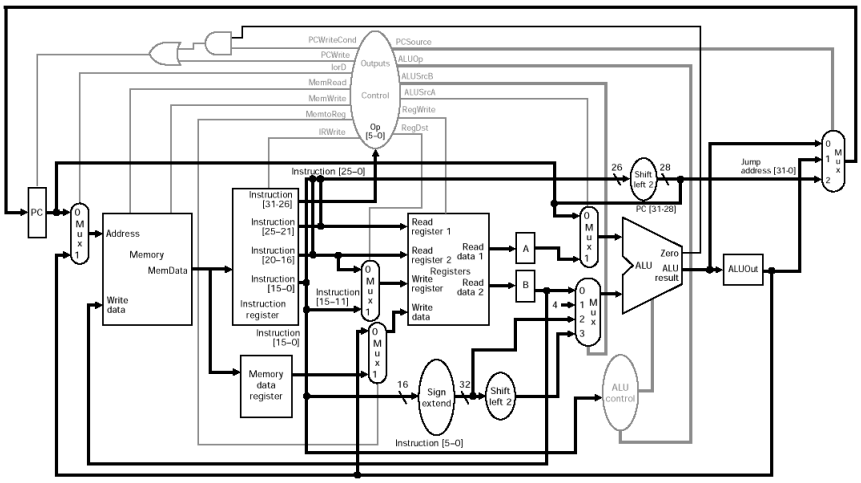
## 3.3 MIPS

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 / \| ]<br>[31:26] [25:21] [20:16] [15:11] [10:0] |
| 2 | [ OpCode / RegDest / RegOrigem / Imediato (ou \|) ]<br>[31:26] [25:21] [20:16] [15:0] |
| 3 | [ OpCode / RegDest / RegOrigem (ou \|) / Imediato (ou \|) ]<br>[31:26] [25:21] [20:16] [15:0] |
| 4 | [ OpCode / RegDest / Imediato ]<br>[31:26] [25:21] [20:0] |
| 5 | [ OpCode / RegDest (ou \|) / EndereçoAlvo (ou \|) ]<br>[31:26] [25:21] [20:16] |

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** |
|---|---|---|
| 000000 | Add | 1 |
| 000001 | Addi | 2 |
| 000010 | Sub | 1 |
| 000011 | Subi | 2 |
| 000100 | Mult | 1 |
| 000101 | Multi | 2 |
| 000110 | Div | 1 |
| 000111 | Divi | 2 |
| 001000 | Mod | 1 |
| 001001 | Slt | 1 |
| 001010 | Slti | 2 |
| 001011 | And | 1 |
| 001100 | Andi | 2 |
| 001101 | Or | 1 |
| 001110 | Ori | 2 |
| 001111 | Not | 2 |
| 010000 | ShR | 2 |
| 010001 | ShL | 2 |
| 010010 | Sgt | 1 |
| 010011 | Sgti | 2 |
| 010100 | Load | 4 |
| 010101 | Store | 4 |
| 010110 | Jump | 3 |
| 010111 | Beq | 3 |
| 011000 | Bne | 3 |
| 011001 | Nop | 5 |
| 011010 | Halt | 5 |
| 011011 | In | 4 |
| 011100 | Out | 4 |
| 011101 | 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

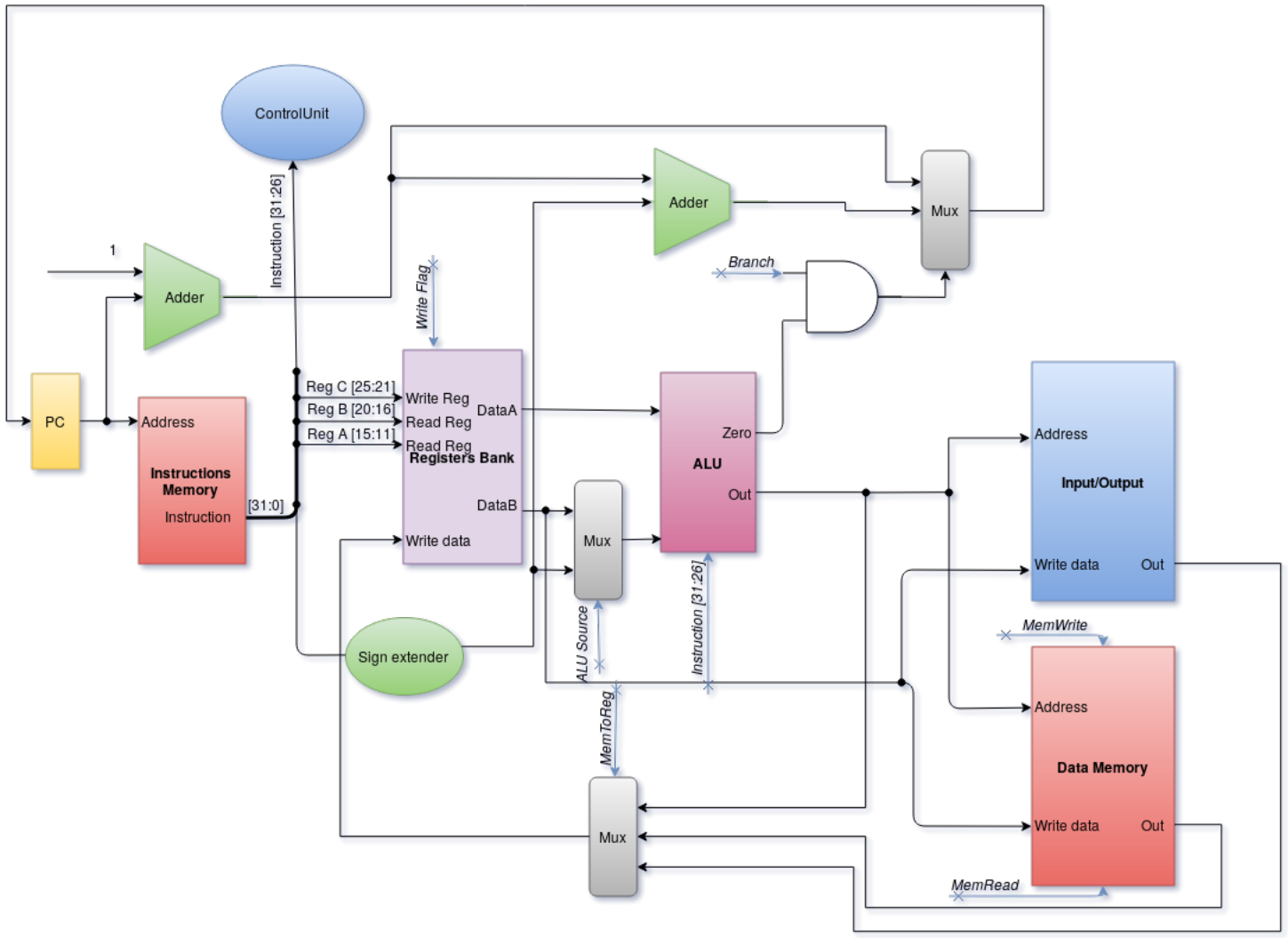
Os modos de endereçamento do CoreBassier são exatamente os mesmos que o do MIPS.

- 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     , immediate,
11     , branchSignal,
12     , aluBranchSignal,
13     , dataA, dataB,
14     , dataToWrite,
15         , aluSource1, aluSource2,
16     , aluError,
17     , aluOut,
18     , error, overflow,
19           , jump
20     , memoryOut,
21     inputOut);
22
23     // Inputs
24     input clock;
25     input sw0, sw1, sw2, sw3, sw4, sw5, sw6, sw7, sw8, sw9, sw10, sw11, sw12, sw13, sw14,
           sw15, sw16, sw17;
26     input key0, key1, key2, key3;
27
28     // Temporary
29     output wire halt;
30     output wire [31:0] _programCounter;
31          output wire [31:0] instructionOut;
32          output wire [5:0] aluCode;
33     output wire targetRegister, aluSource;
34     output wire writeRegister, memoryWrite, memoryRead;
35     output wire [2:0] memoryToRegister;
36     output wire branch, jump;
37     output reg[31:0] immediate;
38     output reg branchSignal;
```

```
39      output wire aluBranchSignal;
40      output wire [31:0] dataA, dataB;
41      output reg [31:0] dataToWrite;
42           output reg [31:0] aluSource1, aluSource2;
43      output wire aluError;
44      output wire [31:0] aluOut;
45      output wire error, overflow;
46      output wire [31:0] memoryOut;
47      output wire [31:0] inputOut;
48
49      // Outputs
50      output [6:0] hex0, hex1, hex2, hex3, hex4, hex5, hex6, hex7;
51
52      // Instances
53      PC programCounter(.clock(clock), .halt(halt), .immediate(immediate), .branchSignal(
            branchSignal), .programCounter(_programCounter), .jump(jump));
54      InstructionsMemory instructionsMemory(.read_addr(_programCounter), .clk(clock), .q(
            instructionOut));
55      ControlUnit controlUnit(.clock(clock), .opcode(instructionOut[31:26]), .aluCode(
            aluCode), .targetRegister(targetRegister), .aluSource(aluSource),
56          .writeRegister(writeRegister), .memoryWrite(memoryWrite), .memoryRead(memoryRead)
                , .memoryToRegister(memoryToRegister), .branch(branch), .halt(halt), .jump(
                jump));
57      RegistersBank registersBank(.regC(instructionOut[25:21]),
58      .regB( instructionOut[31:26] == 6'b010111 || instructionOut[31:26] == 6'b011000 ?
            instructionOut[25:21] : instructionOut[20:16]),
59      .regA( instructionOut[31:26] == 6'b010111 || instructionOut[31:26] == 6'b011000 ?
            instructionOut[20:16] : instructionOut[15:11]),
60      .writeFlag(writeRegister), .clock(clock), .dataA(dataA), .dataB(dataB), .dataWrite(
            dataToWrite));
61      ALU alu(.dataA(aluSource1), .dataB(aluSource2), .opCode(instructionOut[31:26]), .
            branchSignal(aluBranchSignal), .overflow(overflow), .error(error), .dataC(aluOut)
            , .clock(clock));
62      DataMemory dataMemory(.data(dataB), .read_addr(aluOut), .write_addr(aluOut), .we(
            memoryWrite), .clk(clock), .q(memoryOut));
63      IO io(.opcode(instructionOut[31:26]), .address(instructionOut[5:0]), .sw0(sw0), .sw1(
            sw1), .sw2(sw2),
64       .sw3(sw3), .sw4(sw4), .sw5(sw5), .sw6(sw6), .sw7(sw7), .sw8(sw8), .sw9(sw9),
65        .sw10(sw10), .sw11(sw11), .sw12(sw12), .sw13(sw13), .sw14(sw14), .sw15(sw15), .sw16
              (sw16),
66         .sw17(sw17), .key0(key0), .key1(key1), .key2(key2), .outL(inputOut), .hex0(hex0),
67          .hex1(hex1),  .hex2(hex2),  .hex3(hex3),  .hex4(hex4),  .hex5(hex5),  .hex6(hex6)
                ,
68           .hex7(hex7), .data(dataB));
69
70      always @ ( instructionOut or aluSource or branch or aluBranchSignal or
            memoryToRegister )
71      begin
72                  immediate[31:0] = { {16{instructionOut[15]}}, instructionOut[15:0] };
73          branchSignal = aluBranchSignal & branch;
74
75          aluSource1 = dataB;
76
77          /* Mux for ALU input */
78          case (aluSource)
79              1'b0: aluSource2 = dataA;
80              1'b1: aluSource2 = immediate;
81          endcase
82
83          /* Mux for writing data to register */
```

```
84          case (memoryToRegister)
85              3'b000: dataToWrite = aluOut;
86              3'b001: dataToWrite = memoryOut;
87              3'b010: dataToWrite = inputOut;
88          endcase
89      end
90
91      // Out: convert the register data to needed algarisms
92           // Branch syntax: opcode_dataA_dataB_immediate;
93       // Mov syntax: opcode_to_from_000000000000...
94  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. Ela funciona e foi implementada do mesmo jeito que a Memória de Dados, utilizando o template que o Quartus possui para memórias, isso agiliza o tempo de compilação.

```
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
13          begin
14                  $readmemb("ins.txt", ram);
15                  /*ram[0] = 32'b00000100001000000000000000000101;
16                  ram[1] = 32'b00000100001000010000000000000101;
17                  ram[2] = 32'b01011000000000000000000000001010;
18                  ram[3] = 32'b01101000000000000000000000000000;
19                  ram[5] = 32'b01101000000000000000000000000000;
20                  ram[6] = 32'b01101000000000000000000000000000;
21                  ram[7] = 32'b01101000000000000000000000000000;
22                  ram[8] = 32'b01101000000000000000000000000000;
23                  ram[9] = 32'b01101000000000000000000000000000;
24                  ram[10] = 32'b01101000000000000000000000000000;
25                  ram[11] = 32'b01101000000000000000000000000000;*/
26          end
27
28          always @ (posedge clk)
29          begin
30                  q <= ram[read_addr];
31          end
32  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. Além disso, o que acontece aqui é que para cada tipo de instrução, a Unidade de Controle processa as *flags* corretamente. Essas *flags* são as mostradas na Figura 2.

```
module ControlUnit(clock,
                   opcode,
                   aluCode,
                   targetRegister,
                   aluSource,
                   writeRegister,
                   memoryWrite,
                   memoryRead,
                   memoryToRegister,
                   branch,
                   halt,
                                              jump);

    // Inputs
    input clock;
    input [5:0] opcode;
    // Wait for in | out

    // Temporary
    reg _halt;

    // Outputs
    output reg[5:0] aluCode;
    output reg targetRegister, aluSource, writeRegister, memoryWrite, memoryRead;
    output reg[2:0] memoryToRegister;
    output reg branch, jump;
         output halt;

    always @ ( opcode )
    begin
        //if (!_halt)
        //begin
            case (opcode)
                // Add
                6'b000000,
                // Sub
                6'b000010,
                // Mult
                6'b000100,
                // Div
                6'b000110,
                // Mod
                6'b001000,
                // SetLesserThan
                6'b001001,
                // And
                6'b001011,
                // Or
                6'b001101,
                // Not
                6'b001111,
                // Shift Right
                6'b010000,
                // Shift Left
                6'b010001,
                // SetGreaterThan
                6'b010010:
                begin
                    aluCode = opcode;
                    targetRegister = 1;
```

```
61                      aluSource = 0;
62                      writeRegister = 1;
63                      memoryWrite = 0;
64                      memoryRead = 0;
65                      memoryToRegister = 0;
66                      branch = 0;
67                      _halt = 0;
68                                              jump = 0;
69                  end
70
71                  // Addi
72                  6'b000001,
73                  // Subi
74                  6'b000011,
75                  // Multi
76                  6'b000101,
77                  // Divi
78                  6'b000111,
79                  // SetLesserThanImediate
80                  6'b001010,
81                  // Andi
82                  6'b001100,
83                  // Ori
84                  6'b001110,
85                  // SetGreaterThanImediate
86                  6'b010011:
87                  begin
88                      aluCode = opcode;
89                      targetRegister = 1;
90                      aluSource = 1;
91                      writeRegister = 1;
92                      memoryWrite = 0;
93                      memoryRead = 0;
94                      memoryToRegister = 0;
95                      branch = 0;
96                      _halt = 0;
97                                              jump = 0;
98                  end
99
100                 // Load, Store
101                 6'b010100, 6'b010101:
102                 begin
103                     aluCode = 6'b000000;
104                     targetRegister = 0;
105                     aluSource = 1;
106                     writeRegister = opcode == 6'b010100 ? 1 : 0;
107                     memoryWrite = opcode == 6'b010100 ? 0 : 1;
108                     memoryRead = opcode == 6'b010100 ? 1 : 0;
109                     memoryToRegister = 1;
110                     branch = 0;
111                     _halt = 0;
112                                             jump = 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 = 0;
126                                                 jump = 1;
127                     _halt = 0;
128                 end
129
130                 // Branch on (NOT) Equal
131                 6'b010111, 6'b011000:
132                 begin
133                     aluCode = opcode;
134                     targetRegister = 0;
135                     aluSource = 0;
136                     writeRegister = 0;
137                     memoryWrite = 0;
138                     memoryRead = 0;
139                     memoryToRegister = 0;
140                     branch = 1;
141                     _halt = 0;
142                                                 jump = 0;
143                 end
144
145                 // Nop
146                 6'b011001:
147                 begin
148                     aluCode = opcode;
149                     targetRegister = 0;
150                     aluSource = 0;
151                     writeRegister = 0;
152                     memoryWrite = 0;
153                     memoryRead = 0;
154                     memoryToRegister = 0;
155                     _halt = 0;
156                     branch = 0;
157                                                 jump = 0;
158                 end
159
160                 // Halt
161                 6'b011010:
162                 begin
163                     aluCode = opcode;
164                     targetRegister = 0;
165                     aluSource = 0;
166                     writeRegister = 0;
167                     memoryWrite = 0;
168                     memoryRead = 0;
169                     memoryToRegister = 0;
170                     _halt = 1;
171                     branch = 0;
172                                                 jump = 0;
173                 end
174
175                 // In
176                 6'b011011:
177                 begin
178                     aluCode = opcode;
179                     targetRegister = 0;
180                     aluSource = 0;
```

```
181                     writeRegister = 1;
182                     memoryWrite = 0;
183                     memoryRead = 0;
184                     memoryToRegister = 2'b10;
185                     _halt = 0;
186                     branch = 0;
187                                                 jump = 0;
188                 end
189
190                 // Out
191                 6'b011100:
192                 begin
193                     aluCode = opcode;
194                     targetRegister = 0;
195                     aluSource = 0;
196                     writeRegister = 0;
197                     memoryWrite = 0;
198                     memoryRead = 0;
199                     memoryToRegister = 0;
200                     _halt = 0;
201                                                 jump = 0;
202                     branch = 0;
203                 end
204
205                 // Mov
206                 6'b011101:
207                 begin
208                     aluCode = opcode;
209                     targetRegister = 0;
210                     aluSource = 0;
211                     writeRegister = 1;
212                     memoryWrite = 0;
213                     memoryRead = 0;
214                     memoryToRegister = 0;
215                     _halt = 0;
216                     branch = 0;
217                                                 jump = 0;
218                 end
219
220                                         default:
221                                         begin
222                                                 aluCode = 6'b000000;
223                                                 targetRegister = 0;
224                                                 aluSource = 0;
225                                                 writeRegister = 0;
226                                                 memoryWrite = 0;
227                                                 memoryRead = 0;
228                                                 memoryToRegister = 0;
229                                                 branch = 0;
230                                                 _halt = 0;
231                                                 jump = 0;
232                                         end
233             endcase
234         end
235     //end
236
237     assign halt = _halt;
238     // Wrong case or beq
239 endmodule
```

O módulo abaixo é a Unidade Lógica e Aritmética. Ela funciona muito semelhante a Unidade de Controle no quesito de como foi implementada, ambas checam para ver qual é a instrução que está sendo executada (através do opcode) e fazem as operações de acordo. A única diferença da Unidade Lógica e Aritmética do CoreBassier para o MIPS é que o CoreBassier tem um temporário de 64 bits para verificar se houve *overflow* nas operações.

```
1  module ALU
2                  (opCode,
3                  dataA,
4                  dataB,
5                  branchSignal,
6                  overflow,
7                  error,
8                  dataC,
9                  clock);
10
11         // Inputs
12         input [31:0] dataA;
13         input [31:0] dataB;
14         input [5:0] opCode;
15         input clock;
16
17         // Temporary
18         reg [63:0] _temp;
19
20         // Outputs
21         output reg branchSignal, overflow;
22         output reg error;
23         output reg [31:0] dataC;
24
25         always @ ( opCode or dataA or dataB )
26         begin
27                 error = 0;
28                 branchSignal = 1;
29                 case(opCode)
30                         // Add, Addi
31                         6'b000000, 6'b000001:
32                         begin
33                                 _temp = {8'h00000000,dataA} + {8'h00000000,dataB};
34                                 overflow = (_temp < 8'hFFFFFFFF) ? 0 : 1;
35                                 dataC = _temp;
36                         end
37
38                         // Sub, Subi
39                         6'b000010, 6'b000011:
40                         begin
41                                 dataC = dataA - dataB;
42                         end
43
44                         // Mult, Multi
45                         6'b000100, 6'b000101:
46                         begin
47                                 _temp = {8'h00000000,dataA} * {8'h00000000,dataB};
48                                 overflow = (_temp < 8'hFFFFFFFF) ? 0 : 1;
49                                 dataC = _temp;
50                         end
51
52                         // Div, divi
```

```
53                          6'b000110, 6'b000111:
54                          begin
55                                  // Div by zero error
56                                  error = (dataB == 0) ? 1 : 0;
57                                  if (error == 0) dataC = dataA / dataB;
58                                  else dataC = 32'b0;
59                          end
60
61                          // Mod
62                          6'b001000: dataC = dataA % dataB;
63
64                          // SetLesserThan, SetLesserThanImediate
65                          6'b001001, 6'b001010: dataC = (dataA < dataB);
66
67                          // And, Andi
68                          6'b001011, 6'b001100: dataC = dataA & dataB;
69
70                          // Or, Ori
71                          6'b001101, 6'b001110: dataC = dataA | dataB;
72
73                          // Not
74                          6'b001111: dataC = ~dataA;
75
76                          // Shift Right
77                          6'b010000: dataC = dataA >> dataB;
78
79                          // Shift Left
80                          6'b010001: dataC = dataA << dataB;
81
82                          // SetGreaterThan, SetGreaterThanImediate
83                          6'b010010, 6'b010011: dataC = (dataA > dataB);
84
85                          // Branch on Equal
86                          6'b010111: branchSignal = (dataB == dataA);
87
88                          // Branch on Not Equal
89                          6'b011000: branchSignal = (dataB != dataA);
90
91                          // Out
92          6'b011100: dataC = dataA;
93
94                          // Mov
95                          6'b011101:
96                          begin
97                                  branchSignal = 1;
98                                  dataC = dataA;
99                          end
100
101                         default:
102                         begin
103                                 branchSignal = 1;
104                                 dataC = dataB;
105                         end
106                 endcase
107                 error = (overflow == 1) ? 1 : 0;
108         end
109 endmodule
```

O módulo abaixo é o Banco de Registradores. Não tem nada de especial, é apenas

um vetor de posições.

```
module RegistersBank(regA,
                regB,
                regC,
                dataWrite,
                writeFlag,
                clock,
                dataA,
                dataB);

  input [31:0] dataWrite;
  input [4:0] regC, regA, regB;
  input writeFlag, clock;
  output [31:0] dataA, dataB;

  reg [31:0] registers[31:0];

  initial begin
    registers[0] = 32'b0;
  end

  always @ ( posedge clock )
  begin
                if(writeFlag)
      registers[regC] = dataWrite;
  end

  assign dataA = registers[regA];
  assign dataB = registers[regB];
endmodule
```

O módulo abaixo é a Memória de Dados.

```
module DataMemory
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=10)
(
        input [(DATA_WIDTH-1):0] data,
        input [(ADDR_WIDTH-1):0] read_addr, write_addr,
        input we, clk,
        output reg [(DATA_WIDTH-1):0] q
);

        // Declare the RAM variable
        reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

        always @ (negedge clk)
        begin
                // Write
                if (we)
                        ram[write_addr] <= data;

                q <= ram[read_addr];
        end

endmodule
```

O módulo abaixo é a Entrada e Saída.

O módulo de Entrada e Saída trata as duas instruções separadamente: ele lê o conjunto

de 17 chaves que o kit FPGA possui; e escreve nos displays de sete segmentos um dado que está no banco de registradores.

Quando utiliza-se a instrução de saída, deve-se notar que há como imprimir números separadamente (números de apenas um bit), ou imprimir um conjunto de números, tudo isso depende de qual é o endereço que foi o alvo da instrução.

```
1  /*
2   * A note to the reader: UNLESS you're using 'out' targeting the ldc display,
3   * you must know that targeting:
4   * - 0x0: prints a 4 bit (signed) integer, using the adjacents displays;
5   * - 0x4: prints a 2 (unsigned) integer;
6   * - 0x6: prints a 2 (unsigned) integer;
7   * BUT, if you target:
8   * - 0x1, 0x2, 0x3, 0x5, 0x6: prints an 1 bit (unsigned) integer;
9   */
10 module IO(
11         address,
12         data,
13                                 opcode,
14                                 sw0, sw1, sw2, sw3, sw4, sw5, sw6, sw7, sw8, sw9, sw10,
                                       sw11, sw12, sw13, sw14, sw15, sw16, sw17,
15                                 key0, key1, key2, key3,
16                                 lcdOut,
17                                 hex0, hex1, hex2, hex3, hex4, hex5, hex6, hex7,
18                                 outL
19          );
20     // Inputs
21     input [4:0] address;
22           input [31:0] data;
23     input [5:0] opcode;
24     input sw0, sw1, sw2, sw3, sw4, sw5, sw6, sw7, sw8, sw9, sw10, sw11, sw12, sw13, sw14,
           sw15, sw16, sw17;
25     input key0, key1, key2, key3;
26
27     // Temporary
28     reg [31:0] _temp;
29     reg negative;
30     reg [3:0] hundred, ten, unit;
31     reg [7:0] outHundreds, outTens, outUnits;
32     integer i;
33
34     // Outputs
35     output [31:0] lcdOut;
36     output reg [6:0] hex0, hex1, hex2, hex3, hex4, hex5, hex6, hex7;
37     output [31:0] outL;
38
39     always @ (address or opcode)
40     begin
41       case (opcode)
42           // In
43           6'b011011:
44           begin
45             case (address)
46               5'b00000,
47               5'b00001,
48               5'b00010,
49               5'b00011,
```

```
50                5'b00100,
51                5'b00101,
52                5'b00110,
53                5'b00111,
54                5'b01000,
55                5'b01001,
56                5'b01010,
57                5'b01011,
58                5'b01100,
59                5'b01101,
60                5'b01110,
61                5'b01111,
62                5'b10000,
63                5'b10001:
64                begin
65                  _temp[31:0] = { {15{sw17}}, sw16, sw15, sw14, sw13, sw12, sw11, sw10, sw9
                        , sw8, sw7, sw6, sw5, sw4, sw3, sw2, sw1, sw0 };
66                end
67                5'b10010: _temp = key0;
68                5'b10011: _temp = key1;
69                5'b10100: _temp = key2;
70                5'b10101: _temp = key3;
71              endcase
72            end
73            // Out
74            6'b011100:
75            begin
76                    i = 7;
77                    // Convert numbers to BCD before using
78                    /* Max 8 bits */
79                    if (data[7])
80                    begin
81                      negative = 1;
82                      _temp = ~data + 1;
83                    end
84                    else
85                    begin
86                      negative = 0;
87                      _temp = data;
88                    end
89
90                    for (i = 7; i >= 0; i = i - 1)
91                    begin
92                      if (hundred > 4)
93                        hundred = hundred + 3;
94                      if (ten > 4)
95                        ten = ten + 3;
96                      if (unit > 4)
97                        unit = unit + 3;
98
99                      hundred = hundred << 1;
100                     hundred[0] = ten[3];
101                     ten = ten << 1;
102                     ten[0] = unit[3];
103                     unit = unit << 1;
104                     unit[0] = _temp[i];
105                   end
106
107                   case (hundred)
108                       4'b0000: outHundreds = 7'b0000001;
```

```
109                     4'b0001: outHundreds = 7'b1001111;
110                     4'b0010: outHundreds = 7'b0010010;
111                     4'b0011: outHundreds = 7'b0000110;
112                     4'b0100: outHundreds = 7'b1001100;
113                     4'b0101: outHundreds = 7'b0100100;
114                     4'b0110: outHundreds = 7'b0100000;
115                     4'b0111: outHundreds = 7'b0001111;
116                     4'b1000: outHundreds = 7'b0000000;
117                     4'b1001: outHundreds = 7'b0001100;
118                     default: outHundreds = 7'b1111111;
119                 endcase
120
121                 case (ten)
122                     4'b0000: outTens = 7'b0000001;
123                     4'b0001: outTens = 7'b1001111;
124                     4'b0010: outTens = 7'b0010010;
125                     4'b0011: outTens = 7'b0000110;
126                     4'b0100: outTens = 7'b1001100;
127                     4'b0101: outTens = 7'b0100100;
128                     4'b0110: outTens = 7'b0100000;
129                     4'b0111: outTens = 7'b0001111;
130                     4'b1000: outTens = 7'b0000000;
131                     4'b1001: outTens = 7'b0001100;
132                     default: outTens = 7'b1111111;
133                 endcase
134
135                 case (unit)
136                     4'b0000: outUnits = 7'b0000001;
137                     4'b0001: outUnits = 7'b1001111;
138                     4'b0010: outUnits = 7'b0010010;
139                     4'b0011: outUnits = 7'b0000110;
140                     4'b0100: outUnits = 7'b1001100;
141                     4'b0101: outUnits = 7'b0100100;
142                     4'b0110: outUnits = 7'b0100000;
143                     4'b0111: outUnits = 7'b0001111;
144                     4'b1000: outUnits = 7'b0000000;
145                     4'b1001: outUnits = 7'b0001100;
146                     default: outUnits = 7'b1111111;
147                 endcase
148
149             case (address)
150               // Targeting seven segments display
151               5'b00000:
152               begin
153                 hex0 = outUnits;
154                 hex1 = outTens;
155                 hex2 = outHundreds;
156               end
157               5'b00001: hex1 = outUnits;
158               5'b00010: hex2 = outUnits;
159               5'b00011: hex3 = outUnits;
160               5'b00100:
161               begin
162                 hex4 = outUnits;
163                 hex5 = outTens;
164               end
165               5'b00101: hex5 = outUnits;
166               5'b00110:
167               begin
168                 hex6 = outUnits;
```

```
169                     hex7 = outTens;
170                   end
171                   5'b00111: hex7 = outUnits;
172                 endcase
173             end
174         endcase
175           end
176   assign outL = _temp;
177
178 endmodule
```

# 5 Resultados Obtidos e Discussões

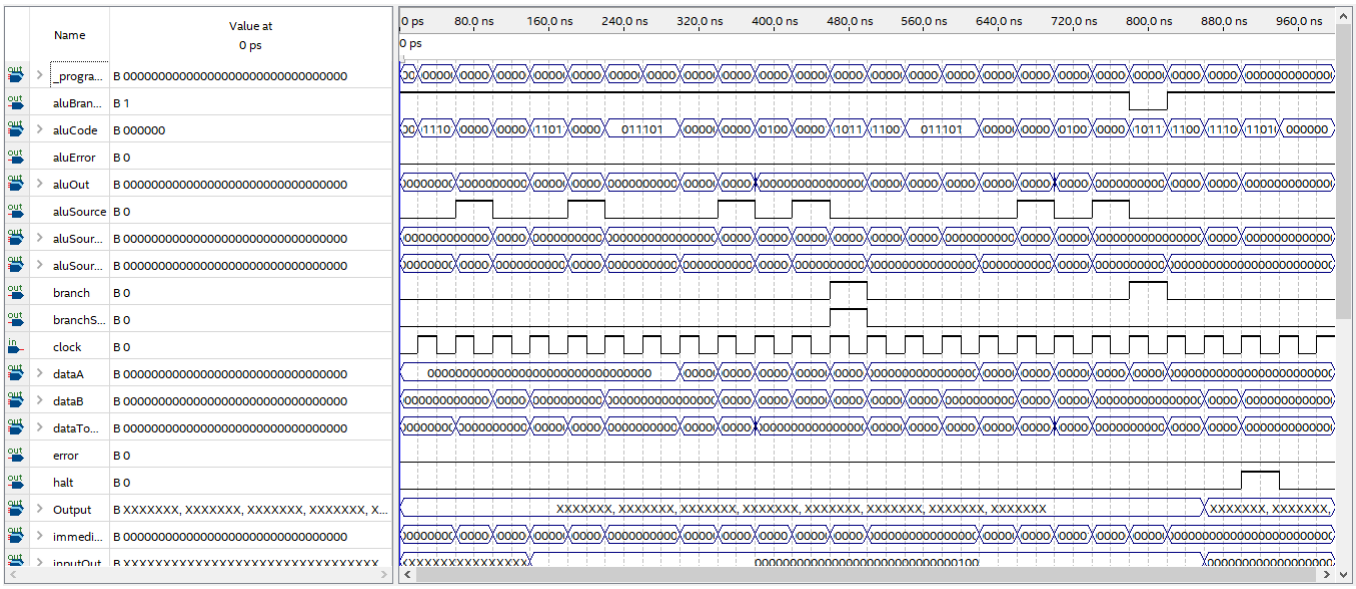
O CoreBassier está completo.
Após algumas trocas nas instruções, e no tipo de instruções, está finalizado. Utilizando as instruções que foram colocadas na Memória de Instruções, foi realizado o teste utilizando o *Waveform* do Quartus II. O código que o processador está rodando é o fibonacci, listado abaixo.

```
1  011101_00010_00000_00000000000000000 // mov $2, $0
2  000001_00010_00000_00000000000000001 // addi $2, $0, 1
3  000000_00011_00010_00001_00000000000 // add $3, $2, $1
4  000001_00110_00000_00000000000000100 // addi $6, $0, 4
5  011011_00110_00000_00000000000000000 // in $6
6  000001_00101_00000_00000000000000010 // addi $5, $0, 2
7  011101_00001_00010_00000000000000000 // L0: mov $1, $2
8  011101_00010_00011_00000000000000000 // mov $2, $3
9  000000_00011_00010_00001_00000000000 // add $3, $2, $1
10 000001_00101_00101_00000000000000001 // addi $5, $5, 1
11 001001_00111_00101_00110_00000000000 // slt $7, $5, $6
12 000001_01000_00000_00000000000000001 // addi $8, $0, 1
13 010111_00111_01000_00000000000000101 // beq $7, $8, L0
14 011001_00000000000000000000000000000 // nop
15 011100_00000_00011_00000000000000000 // output $3, 0x0
16 011010_00000000000000000000000000000 // halt
```

O código funciona da seguinte forma: Primeiro, é movido 0 para o registrador 2 (que fará o papel de fib[i - 2]) e então é movido 1 para o registrador 1 (que fará o papel de fib[i - 1]). Após isso é calculado o terceiro termo de fibonacci e iniciado a contagem em 2 (o registrador 5 contará quantos números foram gerados), é utilizado a instrução de *input* para obter o valor de parada (ou seja, quantos números de fibonacci o usuário deseja) esse valor é armazenado no registador 6. Agora começa o laço, com a label L0, onde move-se o conteudo do fib[i - 1] para fib[i - 2], depois o conteudo de fib[i] para fib[i - 1], atualizando os anteriores. Então é calculado o próximo valor do fibonacci, somando-se os dois. Com isso, soma-se 1 ao contador $5, e testa se já foi gerado o número desejado com o *Set on Less Than*. E é feito o branch para L0 caso preciso calcular outros valores, ou vai direto para mostrar no display e depois halt, caso não precise.

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.

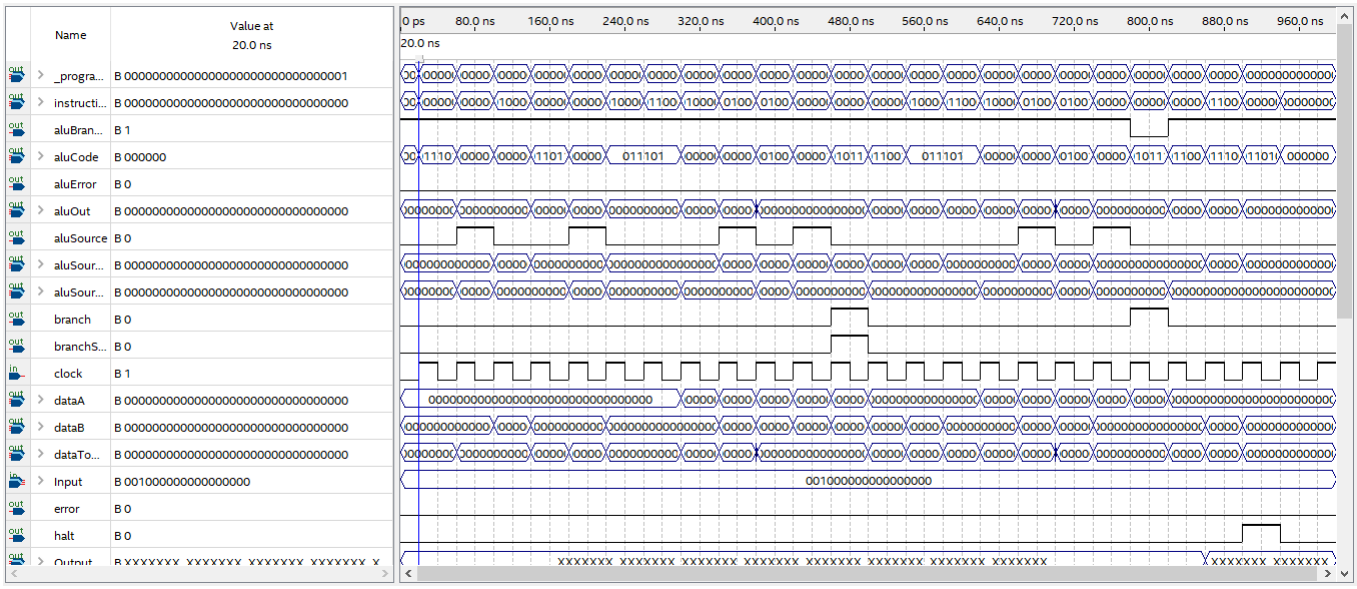
Figura 3 – Antes do primeiro *clock*




Fonte: O Autor

A Figura 4 mostra os sinais após o período de 1 *clock*.
A primeira instrução é o mov, então percebe-se que foi buscado o valor de um registrador para passar para o outro. A Unidade de Controle colocou todos as *flags* para seus valores corretos para ser realizada pela Unidade Lógica e Aritmética a passagem de um para o outro registrador.

Figura 4 – Resultado do primeiro *clock*

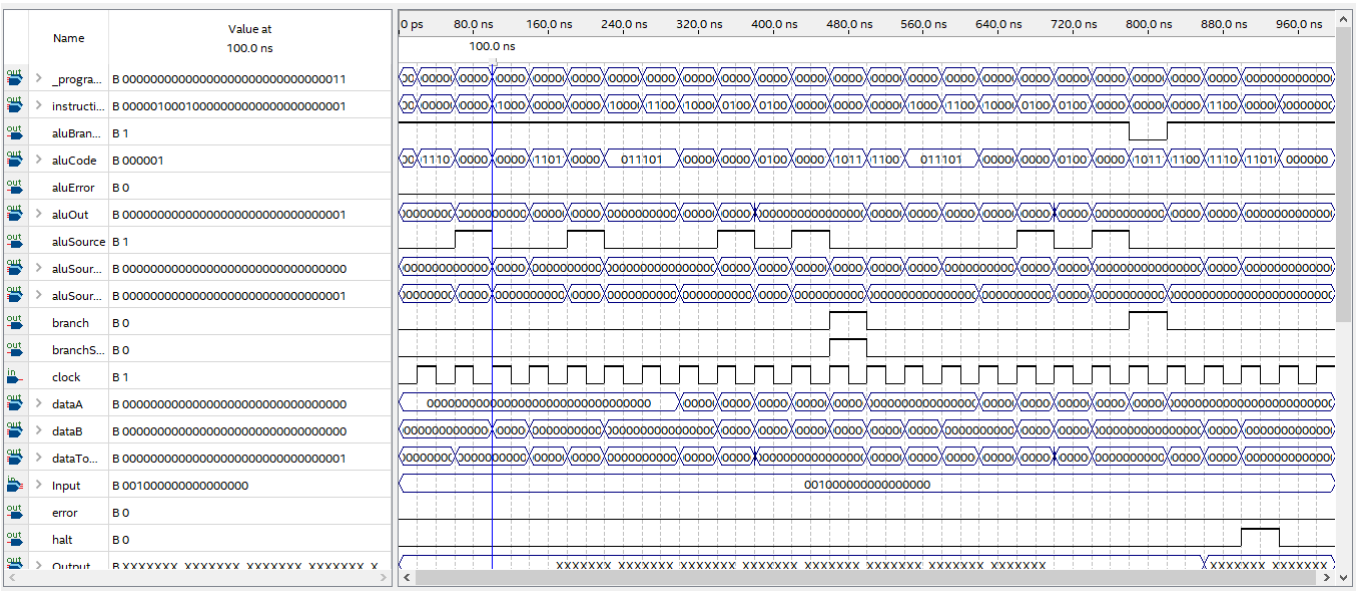



Fonte: O Autor

A Figura 5 mostra os sinais após o período de 3 *clock*.

A terceira instrução é o add, então percebe-se que foi buscado no banco de registradores. 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 registrador 3.

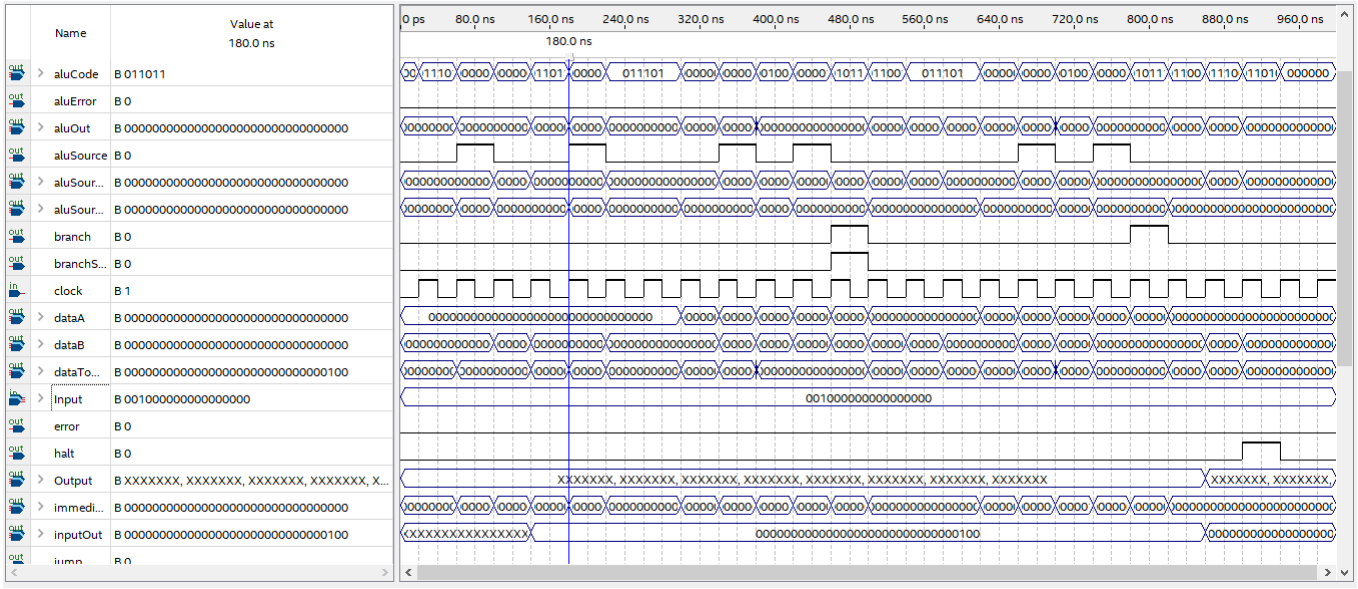
Figura 5 – Resultado do segundo *clock*




Fonte: O Autor

Isso será repetido mais algumas vezes até chegar na Figura 6.
Nesse momento, a instrução sendo executada é o in, e como pode ser visto, o valor $100_2$ é colocado.

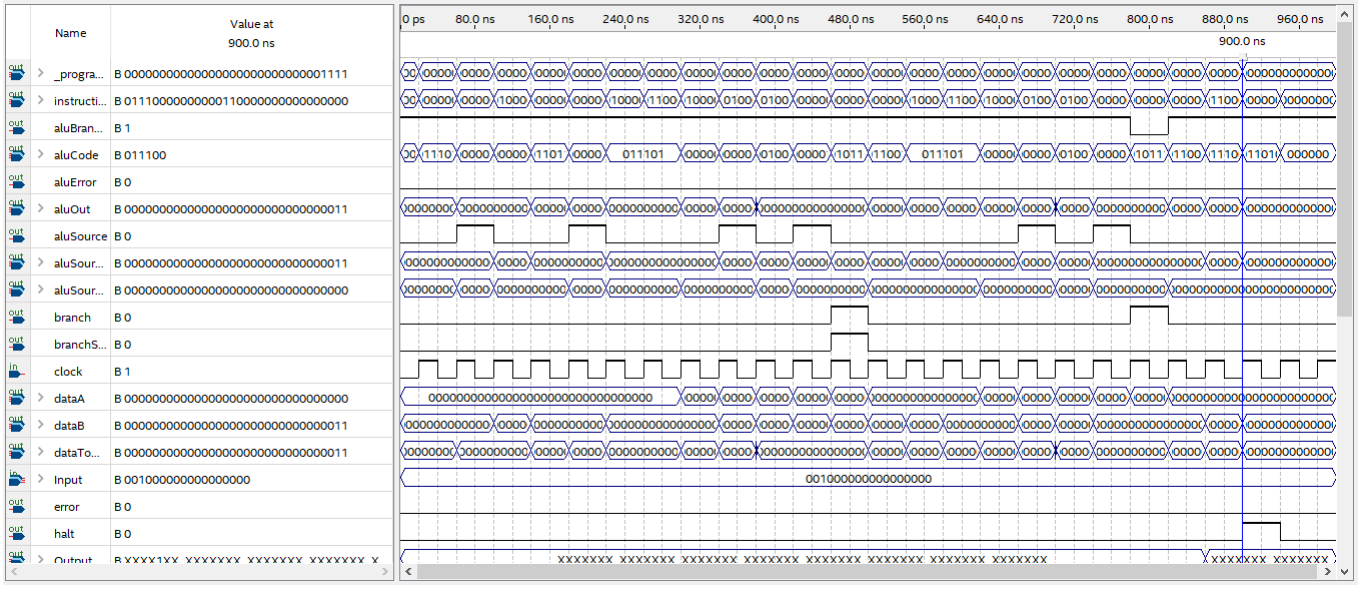
Figura 6 – Resultado do último addi




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. Além disso, pode-se ver que o resultado do programa foi $11_2$ que é o quinto termo: 0 1 1 2 3. (O motivo disso é que o zero é ignorado como primeiro no algoritmo, vide linha 6).

Figura 7 – Resultado do Halt




Fonte: O Autor

# 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.