UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE ENERGIAS ALTERNATIVAS E RENOVÁVEIS
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Desenvolvimento de um processador de ciclo único

PROFISSIONAL II

José Maurício Ramos de Souza Neto

MARIA PAULA MEDEIROS GOMES MIGUEL - 20180064191
PEDRO GERMANO AGRIPINO CRUZ - 20190033395
RAFAEL OLIVEIRA DOS SANTOS - 20200019402

João Pessoa – PB

Setembro, 2025

SUMÁRIO

1. INTRODUÇÃO

O relatório apresentado corresponde à primeira atividade experimental da disciplina Profissional II, do curso de Engenharia Elétrica da Universidade Federal da Paraíba, focando no desenvolvimento e na implementação de um MMASM e sua unidade central de processamento (CPU). O design e a construção de uma CPU envolvem a integração de diversos componentes lógicos, como unidades aritméticas e lógicas (ALUs), registradores e unidades de controle, que devem operar de maneira coordenada para garantir a execução eficiente dos programas.

A teoria por trás do design de CPUs e MMASMs é fundamentada em conceitos de arquitetura de computadores, como descrito por Patterson e Hennessy (2017). Eles discutem a importância de uma estrutura clara para a execução de instruções e a integração de unidades funcionais, enfatizando a necessidade de uma lógica de controle eficiente e a correta formatação de instruções assembly [1]. Adicionalmente, o entendimento das tabelas da verdade e diagramas de tempo é crucial para a análise e validação do funcionamento dos blocos lógicos, conforme abordado por Mano e Ciletti (2018) [2].

O objetivo deste relatório é fornecer uma visão detalhada da solução desenvolvida, incluindo os seguintes aspectos:

- Exploração dos algoritmos utilizados e da lógica de implementação do MMASM, destacando os aspectos técnicos e operacionais envolvidos na construção do processador.
- Descrição dos procedimentos seguidos para a construção da unidade central de processamento, acompanhada de diagramas lógicos para cada bloco funcional e suas respectivas tabelas da verdade ou diagramas de tempo.
- Investigação dos formatos e padrões das instruções assembly presentes no arquivo de entrada .asm que o MMASM reconhece e processa.
- Demonstração da capacidade do MMASM para interpretar e executar quatro exemplos de programas assembly, ilustrando a funcionalidade e a eficácia do processador.
- Inclusão de diagramas lógicos detalhados para cada bloco da CPU, além de tabelas da verdade ou diagramas de tempo que ilustram o funcionamento dos blocos.
- Discussão sobre as decisões técnicas e de design feitas durante o desenvolvimento do projeto e as razões que fundamentam essas escolhas.

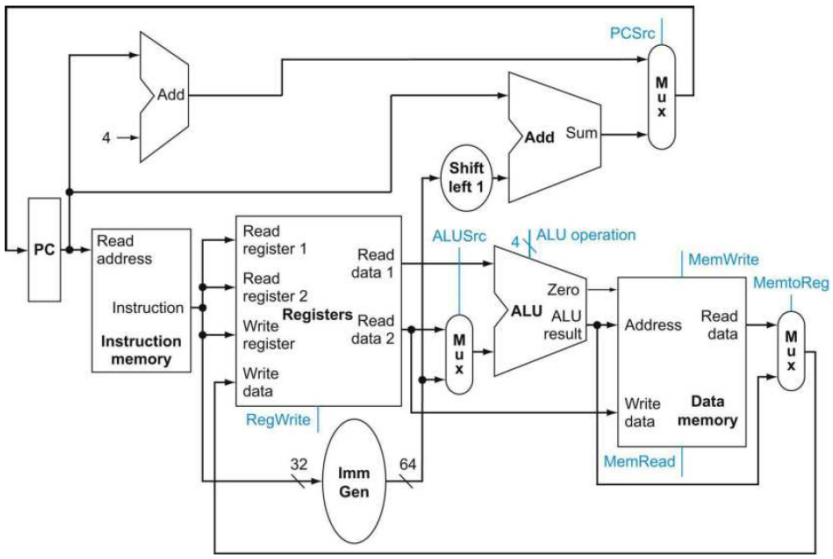
2. PROCESSADOR DE CICLO ÚNICO

O design e a implementação de processadores são fundamentais no estudo da arquitetura de computadores, e um dos modelos básicos de processadores é o processador de ciclo único. Este tipo de processador é caracterizado pela capacidade de executar cada instrução em

um único ciclo de clock, o que simplifica o seu design e operação. Para o desenvolvimento deste trabalho, utilizamos como referência principal o livro "*Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition*"de David A. Patterson e John L. Hennessy [1].

Um processador de ciclo único é composto por várias unidades funcionais essenciais que colaboram para a execução das instruções:

Figura 1 – A estrutura de dados simples para o núcleo da arquitetura RISC-V combina os elementos necessários para diferentes classes de instruções



Fonte: [1].

2.1. Unidade de Controle

A Unidade de Controle (*Control Unit*) é responsável por gerar os sinais de controle que coordenam as operações do processador. Para cada instrução, a Unidade de Controle decodifica o *opcode* (código de operação) e gera os sinais apropriados para os outros componentes do processador, assegurando que a instrução seja executada corretamente.

2.2. Unidade Lógica e Aritmética (ULA)

A Unidade Lógica e Aritmética (ULA ou ALU, do inglês *Arithmetic Logic Unit*) executa operações aritméticas e lógicas sobre os dados. A ULA recebe entradas de registros ou da memória e realiza operações como adição, subtração, AND, OR, e comparações, conforme especificado pelos sinais de controle.

2.3. Registros

O conjunto de registros (*Register File*) é um pequeno armazenamento interno de alta velocidade, onde o processador armazena dados temporários que são frequentemente acessados

durante a execução das instruções. Cada registro pode ser diretamente acessado e utilizado pela ULA para operações, o que torna a execução mais eficiente.

2.4. Memória de Dados

A Memória de Dados (*Data Memory*) armazena os dados que são necessários durante a execução do programa. Em um processador de ciclo único, tanto as operações de leitura quanto as de escrita na memória são realizadas em um único ciclo de clock, sincronizadas pelos sinais de controle.

2.5. Contador de Programa (PC)

O Contador de Programa (PC, do inglês *Program Counter*) mantém o endereço da próxima instrução a ser executada. Após cada instrução ser executada, o PC é atualizado para apontar para a próxima instrução na sequência, permitindo a execução contínua do programa.

2.6. Multiplexadores

Os Multiplexadores (MUXes) são usados para selecionar entre várias entradas possíveis para fornecer a saída correta para a ULA ou outros componentes do processador. Eles são controlados pelos sinais gerados pela Unidade de Controle, assegurando que o dado ou sinal apropriado seja usado em cada etapa da execução.

2.7. Memória de Instruções

A Memória de Instruções (*Instruction Memory*) armazena o conjunto de instruções que o processador deve executar. Durante o ciclo de busca de instrução, o endereço atual do PC é usado para ler a instrução correspondente da memória. A memória de instruções é geralmente implementada como uma memória de leitura apenas (ROM) em processadores simples.

3. PROCEDIMENTO EXPERIMENTAL

A partir do referencial teórico da Seção 2, foi possível desenvolver o processador no *software* Quartus e implementação em FPGA. Além disso, foi desenvolvido um assembler em *python* capaz de transformar um código Assembly em um código de máquina “entendível"pelo processador desenvolvido.

A seguir, será descrito o procedimento experimental adotado para implementar e testar um processador de ciclo único, incluindo a construção e a simulação de seus principais módulos utilizando a linguagem Verilog. Essa abordagem prática visa ilustrar a funcionalidade de cada componente e como eles interagem para executar as instruções de um programa.

3.1. Processador

Este módulo implementa a unidade de controle do processador. Ele usa o campo Opcode da instrução para determinar os sinais de controle necessários. Dependendo do Opcode, ele ajusta sinais como `ALUSrc`, `MemtoReg`, `RegWrite`, entre outros, que controlam a operação da ALU, a leitura e escrita na memória, e o fluxo de controle.

3.1.1. Unidade de Controle

```
module Control_Unit ( Opcode,Jump,MemRead,MemtoReg,MemWrite,ALUSrc,
RegWrite, ALUOp,reset, EN);

    input [6:0] Opcode;
    output reg Jump,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,reset, EN;
    output reg [1:0] ALUOp;

    always @(*)
    begin
        case (Opcode)
            7'b0110011: // R-type instruction
            begin
                ALUSrc <= 0;
                MemtoReg <= 0;
                RegWrite <= 1;
                MemRead <= 0;
                MemWrite <= 0;
                Jump <= 0;
                ALUOp <= 2'b10;
                reset <= 0;
                EN <= 0;
            end

            7'b0000011: // Load instruction
            begin
                ALUSrc <= 1;
                MemtoReg <= 1;
                RegWrite <= 1;
                MemRead <= 1;
                MemWrite <= 0;
                Jump <= 0;
                ALUOp <= 2'b00;
                reset <= 0;
                EN <= 0;
            end

            7'b0100011: // Store instruction
```

```
38              begin
39                  ALUSrc <= 1;
40                  MemtoReg <= 0;
41                  RegWrite <= 0;
42                  MemRead <= 0;
43                  MemWrite <= 1;
44                  Jump <= 0;
45                  ALUOp <= 2'b00;
46                  reset <= 0;
47                  EN <= 0;
48              end
49
50              7'b1101111: // Jump instruction (J-type)
51              begin
52                  ALUSrc <= 0;
53                  MemtoReg <= 0;
54                  RegWrite <= 0;
55                  MemRead <= 0;
56                  MemWrite <= 0;
57                  Jump <= 1;
58                  ALUOp <= 2'b00;
59                  reset <= 0;
60                  EN <= 0;
61              end
62
63              7'b0010011: // ADDi (soma com imediato)
64              begin
65                  ALUSrc <= 1;
66                  MemtoReg <= 0;
67                  RegWrite <= 1;
68                  MemRead <= 0;
69                  MemWrite <= 0;
70                  Jump <= 0;
71                  ALUOp <= 2'b10;
72                  reset <= 0;
73                  EN <= 0;
74              end
75
76              7'b1010101: // reset
77              begin
78                  ALUSrc <= 0;
79                  MemtoReg <= 0;
80                  RegWrite <= 0;
81                  MemRead <= 0;
82                  MemWrite <= 0;
83                  Jump <= 0;
84                  ALUOp <= 2'b00;
```

```
85                  reset <= 1;
86                  EN <= 0;
87              end
88
89              7'b1110001: // DS7
90              begin
91                  ALUSrc <= 1;
92                  MemtoReg <= 0;
93                  RegWrite <= 0;
94                  MemRead <= 1;
95                  MemWrite <= 0;
96                  Jump <= 0;
97                  ALUOp <= 2'b10;
98                  reset <= 0;
99                  EN <= 1;
100             end
101
102             default: // same as R-type
103             begin
104                 ALUSrc <= 0;
105                 MemtoReg <= 0;
106                 RegWrite <= 1;
107                 MemRead <= 0;
108                 MemWrite <= 0;
109                 Jump <= 0;
110                 ALUOp <= 2'b10;
111                 reset <= 0;
112                 EN <= 0;
113             end
114         endcase
115     end
116
117 endmodule
```

Listing 1 – Código Verilog da Unidade de Controle

3.1.2. Unidade Lógica e Aritmética (ULA)

Este módulo implementa a unidade aritmética e lógica (ALU) que realiza operações baseadas no controle de entrada `ALUcontrol_In`. Dependendo do valor de `ALUcontrol_In`, a ALU pode executar operações como AND, OR, soma, subtração, e operações de deslocamento (que permitem multiplicação e divisão por dois).

```
1 module ALU( A, B, ALUcontrol_In, ALUResult, zero);
2
3 input [31:0] A,B;
```

```
input [3:0] ALUcontrol_In;
output reg zero;
output reg [31:0] ALUResult;

always @ (ALUcontrol_In or A or B)
begin
    case (ALUcontrol_In)
        4'b0000: begin zero<=0; ALUResult<=A&B; end
        4'b0001: begin zero<=0; ALUResult<=A|B; end
        4'b0010: begin zero<=0; ALUResult<=A+B; end
        4'b0110: begin if(A==B) zero<=1; else zero<=0; ALUResult<=A-B; end
        4'b1000: begin zero<=0; ALUResult<=A << 1; end  // Multiplica por 2
        4'b1001: begin zero<=0; ALUResult<=A >> 1; end  // Divide por 2
        default: begin zero<=0; ALUResult<=A; end
    endcase
end
endmodule
```

Listing 2 – Código Verilog da ALU

O módulo `ALU_Control` gera sinais de controle para a ALU com base nas entradas `ALUOp`, `funct7` e `funct3`. Esses sinais determinam a operação específica que a ALU deve realizar, como adição, subtração, e operações lógicas.

```
module ALU_Control(ALUOp, funct7, funct3, ALUControl_out);

    input [1:0] ALUOp;
    input funct7;
    input [2:0] funct3;
    output reg [3:0] ALUControl_out;

    always @(*) begin
        case ({ALUOp, funct7, funct3})
            6'b00_0_000 : ALUControl_out <= 4'b0010; // ADD
            6'b01_0_000 : ALUControl_out <= 4'b0110; // SUB
            6'b10_0_000 : ALUControl_out <= 4'b0010; // ADD aritmetica
            6'b10_1_000 : ALUControl_out <= 4'b0110; // SUB aritmetica
            6'b10_0_111 : ALUControl_out <= 4'b0000; // AND aritmetica
            6'b10_0_110 : ALUControl_out <= 4'b0001; // OR  aritmetica
            6'b10_0_001 : ALUControl_out <= 4'b1000; // Multiplica por 2
            6'b10_0_101 : ALUControl_out <= 4'b1001; // Divide por 2
            default: ALUControl_out <= 4'b0000;      // Handle invalid inputs
        endcase
    end

endmodule
```

Listing 3 – Código Verilog da Unidade de Controle da ALU

3.1.3. Registros

Este módulo representa o banco de registradores. Ele armazena 32 registradores de 32 bits e permite leitura e escrita de dados. Na borda positiva do clock, se o sinal de `reset` estiver ativado, todos os registradores são inicializados a zero. Caso contrário, se `RegWrite` estiver ativado, os dados são escritos no registrador especificado.

```
module Register_File(clk, reset, RegWrite, Rs1, Rs2, Rd,
Write_data, Read_data1, Read_data2);

input clk, reset, RegWrite;
input [4:0] Rs1, Rs2, Rd;
input [31:0] Write_data;
output [31:0] Read_data1, Read_data2;

reg [31:0] Registers [31:0];
integer k;

always @(posedge clk) begin
    if (reset == 1'b1) begin
        for (k = 0; k < 32; k = k + 1) begin
            Registers[k] = 32'h0;
        end
    end
    else if (RegWrite == 1'b1) begin
        Registers[Rd] = Write_data;
    end
end

assign Read_data1 = Registers[Rs1];
assign Read_data2 = Registers[Rs2];

endmodule
```

Listing 4 – Código Verilog do Arquivo de Registradores

3.1.4. Memória de Dados

O módulo de memória de dados é responsável por ler e escrever dados na memória. A memória tem 64 endereços de 32 bits. Na borda positiva do clock, se `MemWrite` estiver ativado, os dados são escritos na memória no endereço especificado. Se `MemRead` estiver ativado, os dados são lidos da memória e atribuídos à saída `Read_data`.

```
module Data_Memory(clk, reset, MemWrite, MemRead,
address, Writedata, Data_out);

    input clk, reset, MemWrite, MemRead;
```

```
    input [31:0] address, Writedata;
    output [31:0] Data_out;

    reg [31:0] DataMemory [63:0];
    assign Data_out = (MemRead) ? DataMemory[address] : 32'b0;

    integer k;
    always @(posedge clk)
    begin
        if (reset == 1'b1) begin
            for (k = 0; k < 64; k = k + 1)
                DataMemory[k] = 32'b0;
        end
        else if (MemWrite) begin
            DataMemory[address] = Writedata;
        end
    end

endmodule
```

Listing 5 – Código Verilog da Memória de Dados

3.1.5. Contador de Programa (PC)

O módulo `Program_Counter` é responsável por armazenar e atualizar o valor do contador de programa (PC). Na borda positiva do clock, se o sinal `reset` estiver ativado, o PC é zerado. Caso contrário, o PC é atualizado com o valor da entrada `PC_in`.

```
module Program_Counter (clk, reset, PC_in, PC_out);

input clk, reset;
input [31:0] PC_in;
output reg [31:0] PC_out;

always @ (posedge clk)
begin
    if(reset==1'b1)
        PC_out <= 32'h0;
    else
        PC_out <= PC_in;
end

endmodule
```

Listing 6 – Código Verilog do Contador de Programa

3.1.6. Multiplexadores

Os módulos `Mux1`, `Mux2` e `Mux3` são multiplexadores de 2 entradas e 1 saída. O sinal `Sel` controla a seleção da entrada que é passada para a saída `Mux1_out`, `Mux2_out` e `Mux3_out`. Se `Sel` for 0, a saída é igual à entrada `A`; caso contrário, é igual à entrada `B`.

```
module Mux1(Sel, A1, B1, Mux1_out);

    input Sel;
    input [31:0] A1, B1;
    output [31:0] Mux1_out;

    assign Mux1_out = (Sel == 1'b0) ? A1 : B1;

endmodule
```

Listing 7 – Código Verilog do Módulo Mux1

```
module Mux2(Sel, A2, B2, Mux2_out);

    input Sel;
    input [31:0] A2, B2;
    output [31:0] Mux2_out;

    assign Mux2_out = (Sel == 1'b0) ? A2 : B2;

endmodule
```

Listing 8 – Código Verilog do Módulo Mux2

```
module Mux3(Sel, A3, B3, Mux3_out);

    input Sel;
    input [31:0] A3, B3;
    output [31:0] Mux3_out;

    assign Mux3_out = (Sel == 1'b0) ? A3 : B3;

endmodule
```

Listing 9 – Código Verilog do Módulo Mux3

3.1.7. Memória de Instruções

O módulo `Instruction_Memory` é responsável por armazenar e fornecer instruções a partir de um arquivo de memória. A memória tem 64 endereços de 32 bits. As instruções são lidas da memória e fornecidas na saída `Instructions_out` com base no endereço `read_address`.

```
module Instruction_Memory(clk, read_address, Instructions_out);

    input clk;
    input [31:0] read_address;
    output [31:0] Instructions_out;
    reg [31:0] Imemory [63:0];
    integer k;

    initial begin
        $readmemb("teste.txt", Imemory);
    end

    assign Instructions_out = Imemory[read_address];

endmodule
```

Listing 10 – Código Verilog da Memória de Instruções

3.1.8. Gerador Imediato

O módulo `immediate_Generator` gera o valor imediato a partir da instrução com base no opcode fornecido. Dependendo do opcode, o valor imediato é extraído e estendido para 32 bits.

```
module immediate_Generator (
    input [6:0] Opcode,
    input [31:0] instruction,
    output reg [31:0] ImmExt
);

always @* begin
    case (Opcode)
        7'b0010011: ImmExt = {{22{instruction[29]}}, instruction[29:20]};
// I instruction
        7'b0100011: ImmExt = {{20{1'b0}}, instruction[11:7]};  // S instruction
        7'b0000011: ImmExt = {{20{1'b0}}, instruction[24:20]};  // L instruction
        7'b1110001: ImmExt = {{20{1'b0}}, instruction[11:7]};  // OUT instruction
        7'b1101111: ImmExt = {{26{instruction[31]}}, instruction[31:26]}; //Jump
        default: ImmExt = {{22{instruction[29]}}, instruction[29:20]};  // Default
    endcase
end
endmodule
```

Listing 11 – Código Verilog do Módulo immediate_Generator

O módulo `Demux_Imm` distribui o valor imediato `imm_in` entre duas saídas com base no sinal `Jump`. Se `Jump` estiver ativado, o valor imediato é enviado para `imm_to_pcplus` e

`imm_to_mux` é zerado. Caso contrário, o valor imediato é enviado para `imm_to_mux` e `imm_to_pcplus` é zerado.

```
module Demux_Imm(
    input [31:0] imm_in,
    input Jump,
    output reg [31:0] imm_to_pcplus,
    output reg [31:0] imm_to_mux
);
    always @(*) begin
        if (Jump) begin
            imm_to_pcplus = imm_in;
            imm_to_mux = 32'b0;
        end else begin
            imm_to_pcplus = 32'b0;
            imm_to_mux = imm_in;
        end
    end
endmodule
```

Listing 12 – Código Verilog do Módulo Demux_Imm

3.1.9. Outros

O módulo `And` realiza uma operação lógica E entre os sinais `branch` e `zero`, produzindo a saída `andout`. O resultado é 1 somente quando ambos os sinais são 1.

```
module And(branch, zero, andout);

    input branch, zero;
    output andout;

    assign andout = branch & zero;
endmodule
```

Listing 13 – Código Verilog do Módulo And

O módulo `PCplus` calcula o próximo valor do contador de programa (`NexttoPC`). Se o sinal `jump` estiver ativado, o próximo PC é a soma do PC atual (`fromPC`) com o imediato (`Imediato`) e um incremento de 1. Caso contrário, o próximo PC é apenas o PC atual incrementado por 1.

```
module PCplus(fromPC, NexttoPC, jump, Imediato);

  input  [31:0] fromPC, Imediato;
  output [31:0] NexttoPC;
  input jump;

```

```
  assign NexttoPC = (jump) ? (fromPC + Imediato + 32'd1) : (fromPC + 32'd1);

endmodule
```

Listing 14 – Código Verilog do Módulo PCplus

O módulo `PCplus4` calcula o próximo valor do contador de programa (`NexttoPC`) adicionando 1 ao valor atual do PC (`fromPC`).

```
module PCplus4(fromPC, NexttoPC);

  input  [31:0] fromPC;
  output [31:0] NexttoPC;

  // No livro,    somado + 4. Para o que esper vamos, precisa somar 1
  assign NexttoPC = fromPC + 32'h00000001;

endmodule
```

Listing 15 – Código Verilog do Módulo PCplus4

```
module seletor_bits (
    input [31:0] data_in,
    output [6:0] instruction_control,
    output [4:0] intruction_r_register1,
    output [4:0] intruction_r_register2,
    output [4:0] instruction_w_register
);

assign instruction_control = data_in[6:0];
assign intruction_r_register1 = data_in[19:15];
assign intruction_r_register2 = data_in[24:20];
assign instruction_w_register = data_in[11:7];

endmodule
```

Listing 16 – Código Verilog do Módulo seletor_bits

```
module seletor_bits_2 (
    input [31:0] data_in,
   output func7,
   output [2:0] func3
);

assign func7 = data_in[30];
assign func3 = data_in[14:12];
endmodule
```

Listing 17 – Código Verilog do Módulo seletor_bits$_2$

3.1.10. Display de Sete Segmentos

```
module d7s(
    input [31:0] x,
    input EN,
    output reg [6:0] y0, y1, y2, y3, y4, y5, y6, y7
);

    reg [3:0] unidade, dezena, centena, milhar, dezena_milhar, centena_milhar, uni
    reg [31:0] temp;

always @(*) begin
    if (EN == 0) begin
        y0 = 7'b1111111;
        y1 = 7'b1111111;
        y2 = 7'b1111111;
        y3 = 7'b1111111;
        y4 = 7'b1111111;
        y5 = 7'b1111111;
        y6 = 7'b1111111;
        y7 = 7'b1111111;
    end else begin
        temp = x;

        unidade = temp % 10;
        temp = temp / 10;

        dezena = temp % 10;
        temp = temp / 10;

        centena = temp % 10;
        temp = temp / 10;

        milhar = temp % 10;
        temp = temp / 10;

        dezena_milhar = temp % 10;
        temp = temp / 10;

        centena_milhar = temp % 10;
        temp = temp / 10;

        unidade_milhao = temp % 10;
        temp = temp / 10;

        dezena_milhao = temp % 10;

```

```
46          case(unidade)
47              4'b0000: y0 = 7'b1000000;
48              4'b0001: y0 = 7'b1111001;
49              4'b0010: y0 = 7'b0100100;
50              4'b0011: y0 = 7'b0110000;
51              4'b0100: y0 = 7'b0011001;
52              4'b0101: y0 = 7'b0010010;
53              4'b0110: y0 = 7'b0000010;
54              4'b0111: y0 = 7'b1111000;
55              4'b1000: y0 = 7'b0000000;
56              4'b1001: y0 = 7'b0010000;
57              default: y0 = 7'b1111111;
58          endcase
59
60          case(dezena)
61              4'b0000: y1 = 7'b1000000;
62              4'b0001: y1 = 7'b1111001;
63              4'b0010: y1 = 7'b0100100;
64              4'b0011: y1 = 7'b0110000;
65              4'b0100: y1 = 7'b0011001;
66              4'b0101: y1 = 7'b0010010;
67              4'b0110: y1 = 7'b0000010;
68              4'b0111: y1 = 7'b1111000;
69              4'b1000: y1 = 7'b0000000;
70              4'b1001: y1 = 7'b0010000;
71              default: y1 = 7'b1111111;
72          endcase
73
74          case(centena)
75              4'b0000: y2 = 7'b1000000;
76              4'b0001: y2 = 7'b1111001;
77              4'b0010: y2 = 7'b0100100;
78              4'b0011: y2 = 7'b0110000;
79              4'b0100: y2 = 7'b0011001;
80              4'b0101: y2 = 7'b0010010;
81              4'b0110: y2 = 7'b0000010;
82              4'b0111: y2 = 7'b1111000;
83              4'b1000: y2 = 7'b0000000;
84              4'b1001: y2 = 7'b0010000;
85              default: y2 = 7'b1111111;
86          endcase
87
88          case(milhar)
89              4'b0000: y3 = 7'b1000000;
90              4'b0001: y3 = 7'b1111001;
91              4'b0010: y3 = 7'b0100100;
92              4'b0011: y3 = 7'b0110000;
```

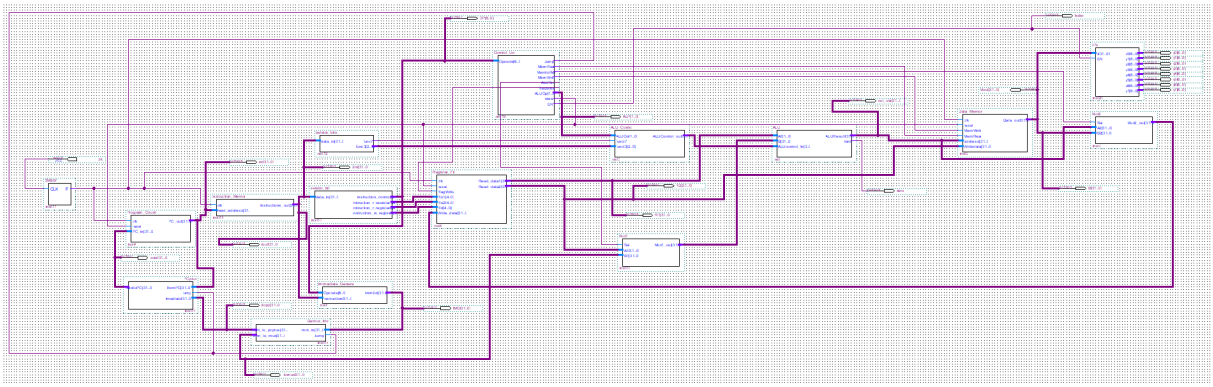
```
93              4'b0100: y3 = 7'b0011001;
94              4'b0101: y3 = 7'b0010010;
95              4'b0110: y3 = 7'b0000010;
96              4'b0111: y3 = 7'b1111000;
97              4'b1000: y3 = 7'b0000000;
98              4'b1001: y3 = 7'b0010000;
99              default: y3 = 7'b1111111;
100         endcase
101
102         case(dezena_milhar)
103             4'b0000: y4 = 7'b1000000;
104             4'b0001: y4 = 7'b1111001;
105             4'b0010: y4 = 7'b0100100;
106             4'b0011: y4 = 7'b0110000;
107             4'b0100: y4 = 7'b0011001;
108             4'b0101: y4 = 7'b0010010;
109             4'b0110: y4 = 7'b0000010;
110             4'b0111: y4 = 7'b1111000;
111             4'b1000: y4 = 7'b0000000;
112             4'b1001: y4 = 7'b0010000;
113             default: y4 = 7'b1111111;
114         endcase
115
116         case(centena_milhar)
117             4'b0000: y5 = 7'b1000000;
118             4'b0001: y5 = 7'b1111001;
119             4'b0010: y5 = 7'b0100100;
120             4'b0011: y5 = 7'b0110000;
121             4'b0100: y5 = 7'b0011001;
122             4'b0101: y5 = 7'b0010010;
123             4'b0110: y5 = 7'b0000010;
124             4'b0111: y5 = 7'b1111000;
125             4'b1000: y5 = 7'b0000000;
126             4'b1001: y5 = 7'b0010000;
127             default: y5 = 7'b1111111;
128         endcase
129
130         case(unidade_milhao)
131             4'b0000: y6 = 7'b1000000;
132             4'b0001: y6 = 7'b1111001;
133             4'b0010: y6 = 7'b0100100;
134             4'b0011: y6 = 7'b0110000;
135             4'b0100: y6 = 7'b0011001;
136             4'b0101: y6 = 7'b0010010;
137             4'b0110: y6 = 7'b0000010;
138             4'b0111: y6 = 7'b1111000;
139             4'b1000: y6 = 7'b0000000;
```

```
140             4'b1001: y6 = 7'b0010000;
141             default: y6 = 7'b1111111;
142         endcase
143
144         case(dezena_milhao)
145             4'b0000: y7 = 7'b1000000;
146             4'b0001: y7 = 7'b1111001;
147             4'b0010: y7 = 7'b0100100;
148             4'b0011: y7 = 7'b0110000;
149             4'b0100: y7 = 7'b0011001;
150             4'b0101: y7 = 7'b0010010;
151             4'b0110: y7 = 7'b0000010;
152             4'b0111: y7 = 7'b1111000;
153             4'b1000: y7 = 7'b0000000;
154             4'b1001: y7 = 7'b0010000;
155             default: y7 = 7'b1111111;
156         endcase
157     end
158 end
159 endmodule
```

Listing 18 – Código Verilog da Memória de Instruções

Ao final da programaçÃo Verilog de todos os módulos, foram gerados blocos de cada módulo de forma que foi possível fazer as conexões entre eles de forma mais visual. Portanto, a montagem completa do processador pode ser vista na Figura 2.

Figura 2 – Processador



Fonte: Autores.

3.2. Assembler

O assembler desenvolvido para este projeto, chamado MMASM, é responsável por converter as instruções assembly, que são legíveis por humanos, em código de máquina, que pode ser executado diretamente pelo processador de ciclo único que estamos desenvolvendo.

Este processo é fundamental para que o processador seja capaz de interpretar e executar as operações desejadas.

O MMASM trabalha analisando cada instrução assembly e mapeando-a para um conjunto específico de bits que representam a operação e seus operandos em código binário. Cada instrução assembly, como ADD, SUB, ADDi, STORE, JMP, entre outras, é traduzida em um formato binário de acordo com a sintaxe específica do processador. Por exemplo, a instrução ADD R1 R2 R0 será convertida para um código de máquina que inclui o opcode correspondente, registradores e outros bits de controle necessários para que o processador execute a operação.

A função assemble, dentro do assembler, é a responsável por essa conversão. Ela quebra a instrução em partes, identifica o opcode, os registradores envolvidos, e quaisquer valores imediatos ou deslocamentos, e então monta a instrução binária correspondente. Por exemplo, a instrução ADDi R1 0 R0 é traduzida em um binário que combina o opcode de uma operação de adição imediata, o valor imediato, e os registradores destino e origem.

Além da funcionalidade básica de conversão, foi desenvolvida uma interface gráfica que facilita a interação do usuário com o assembler. A interface permite que o usuário insira código assembly e, ao clicar no botão "Gerar Código", o código de máquina correspondente é exibido ao lado. O usuário também tem a opção de salvar o código de máquina gerado em um arquivo de texto (.txt), que pode ser posteriormente carregado para execução no hardware.

Essa integração eficiente entre o assembler e a interface gráfica torna o processo de desenvolvimento mais intuitivo, permitindo uma rápida verificação e validação das instruções que serão executadas pelo processador.

```
def assemble(instruction):

def to_twos_complement(value, bits):
    if value < 0:
        value = (1 << bits) + value
    return format(value, f'0{bits}b')

parts = instruction.split()
opcode = parts[0]

if opcode == 'ADD':
    func7 = '0000000'
    regDest = format(int(parts[1][1:]), '05b')
    reg2 = format(int(parts[3][1:]), '05b')
    reg1 = format(int(parts[2][1:]), '05b')
    func3 = '000'
    opcode_bin = '0110011'
    return f'{func7}{reg2}{reg1}{func3}{regDest}{opcode_bin}'
```

```
elif opcode == 'SUB':
    func7 = '0100000'
    regDest = format(int(parts[1][1:]), '05b')
    reg2 = format(int(parts[3][1:]), '05b')
    reg1 = format(int(parts[2][1:]), '05b')
    func3 = '000'
    opcode_bin = '0110011'
    return f'{func7}{reg2}{reg1}{func3}{regDest}{opcode_bin}'

elif opcode == 'ADDi':
    func7 = '00'
    imediato = format(int(parts[2]), '010b')
    reg = format(int(parts[3][1:]), '05b')
    func3 = '000'
    regDest = format(int(parts[1][1:]), '05b')
    opcode_bin = '0010011'
    return f'{func7}{imediato}{reg}{func3}{regDest}{opcode_bin}'

elif opcode == 'SUBi':
    func7 = '01'
    imediato = format(int(parts[2]), '010b')
    reg = format(int(parts[1][1:]), '05b')
    func3 = '000'
    regDest = format(int(parts[3][1:]), '05b')
    opcode_bin = '0010011'
    return f'{func7}{imediato}{reg}{func3}{regDest}{opcode_bin}'

elif opcode == 'STORE':
    reg1 = format(int(parts[2][1:]), '05b')
    regDest = format(int(parts[1][1:]), '05b')
    func3 = '000'
    opcode_bin = '0100011'
    endreg = '00000'
    zeros = '0000000'
    return f'{zeros}{reg1}{endreg}{func3}{regDest}{opcode_bin}'

elif opcode == 'JMP':
    offset = to_twos_complement(int(parts[1]), 6)  # 6-bit offset
```

```
        opcode_bin = '1101111'
        zeros = '0' * 19
        return f'{offset}{zeros}{opcode_bin}'

    elif opcode == 'LOAD':
        reg1 = format(int(parts[2][1:]), '05b')
        regDest = format(int(parts[1][1:]), '05b')
        func3 = '000'
        opcode_bin = '0000011'
        endreg = '00000'
        complemento = '0000000'
        return f'{complemento}{reg1}{endreg}{func3}{regDest}{opcode_bin}'

    elif opcode == 'DIV2':
        reg2 = format(int(parts[3][1:]), '05b')
        reg1 = format(int(parts[2][1:]), '05b')
        func3 = '001'
        opcode_bin = '0110011'
        endreg = format(int(parts[1][1:]), '05b')
        complemento = '0000000'
        return f'{complemento}{reg2}{reg1}{func3}{endreg}{opcode_bin}'

    elif opcode == 'MUL2':
        reg2 = format(int(parts[3][1:]), '05b')
        reg1 = format(int(parts[2][1:]), '05b')
        func3 = '101'
        opcode_bin = '0110011'
        endreg = format(int(parts[1][1:]), '05b')
        complemento = '0100000'
        return f'{complemento}{reg2}{reg1}{func3}{endreg}{opcode_bin}'

    elif opcode == 'OUT':
        opcode_bin = '1110001'
        endreg = format(int(parts[1][1:]), '05b')
        complemento = 20 * '0'
        return f'{complemento}{endreg}{opcode_bin}'

    elif opcode == 'RESET':
        zeros = 25*'0'
```

```
        opcode_bin = '1010101'
        return f'{zeros}{opcode_bin}'

    elif opcode == 'MOV':
        func7 = '0000000'
        regDest = format(int(parts[1][1:]), '05b')
        reg2 = '00000'
        reg1 = format(int(parts[2][1:]), '05b')
        func3 = '000'
        opcode_bin = '0110011'
        return f'{func7}{reg2}{reg1}{func3}{regDest}{opcode_bin}'
    else:
        raise ValueError("Instrução desconhecida!")
```

Figura 3 – Implementação do assembler

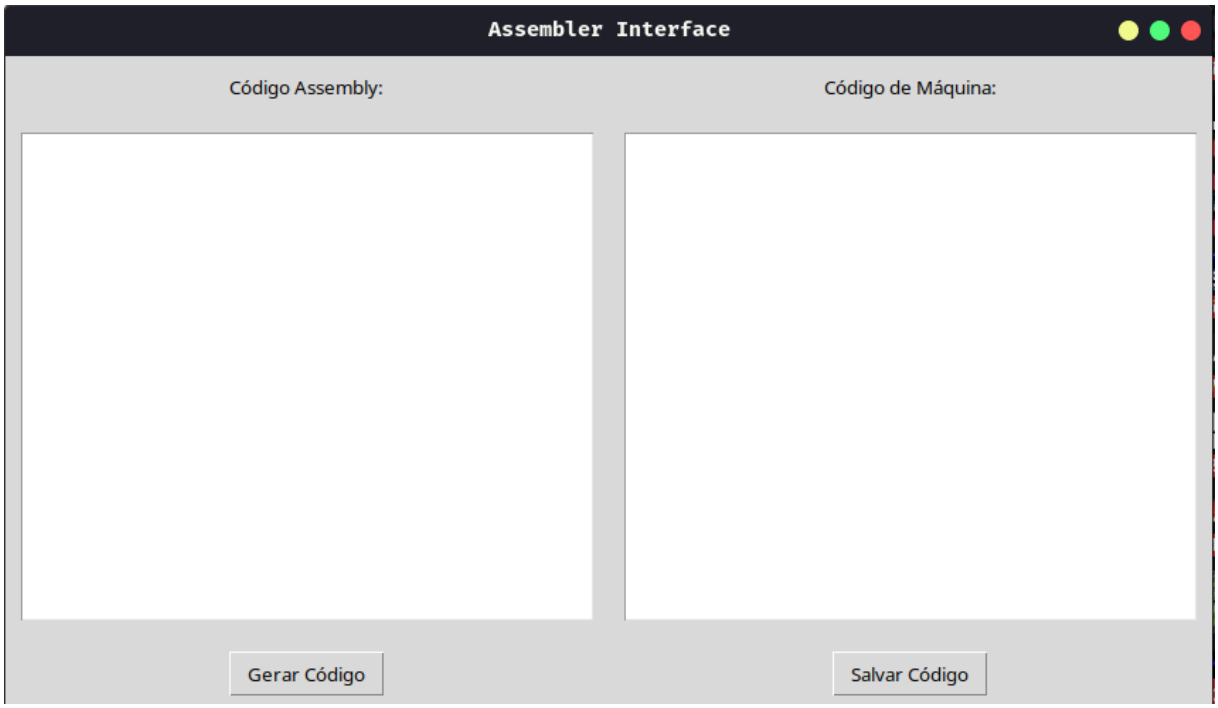


Figura 4 – Interface gráfica desenvolvida para o assembler.

4. RESULTADOS E DISCUSSÕES

Diante do exposto anteriormente, essa sessão tratará das conclusões obtidas a partir da implementação e testes do processador de ciclo único desenvolvido. Nesta parte, serão analisados os resultados da simulação do processador, verificando se os objetivos propostos foram atingidos.

4.1. Integração com o Assembler

Nesta seção, discutiremos o funcionamento do assembler desenvolvido, utilizando como exemplo o código assembly que gera a sequência de Fibonacci. O objetivo é demonstrar como as instruções em assembly são convertidas em código de máquina, que pode ser executado diretamente pelo processador.
O código utilizado para gerar a sequência de Fibonacci é composto pelas seguintes instruções:

```
ADDi R1 0 R0
OUT R1
ADDi R2 1 R0
STORE R1 R2
OUT R1
ADD R3 R1 R2
STORE R1 R3
OUT R1
MOV R1 R2
MOV R2 R3
JMP -6
```

Figura 5 – Código que gera a sequência de fibonacci.

Cada uma dessas instruções é processada pelo assembler, que converte o código assembly em uma sequência binária correspondente. Por exemplo, a instrução ADDi R2 1 R0 é interpretada como uma operação de adição imediata, onde o valor 1 é adicionado ao registrador R0, e o resultado é armazenado no registrador R2. Esta instrução é então convertida em um código binário específico que representa essa operação.
Ao longo da execução do código, a sequência de Fibonacci é gerada ao armazenar e somar os valores dos registradores, com os resultados sendo periodicamente exibidos por meio da instrução OUT. A instrução JMP -6 é utilizada para criar um loop, retornando o fluxo de execução ao início do código para continuar a geração da sequência.
A interface gráfica desenvolvida permite visualizar tanto o código assembly quanto o código de máquina gerado pelo assembler. 6, pode-se observar como o código assembly da sequência de Fibonacci é traduzido em seu equivalente binário, demonstrando a funcionalidade e a precisão do assembler.
Essa visualização não só facilita a validação do processo de conversão como também oferece uma ferramenta prática para entender como o processador interpreta e executa as instruções a partir do código de máquina gerado.

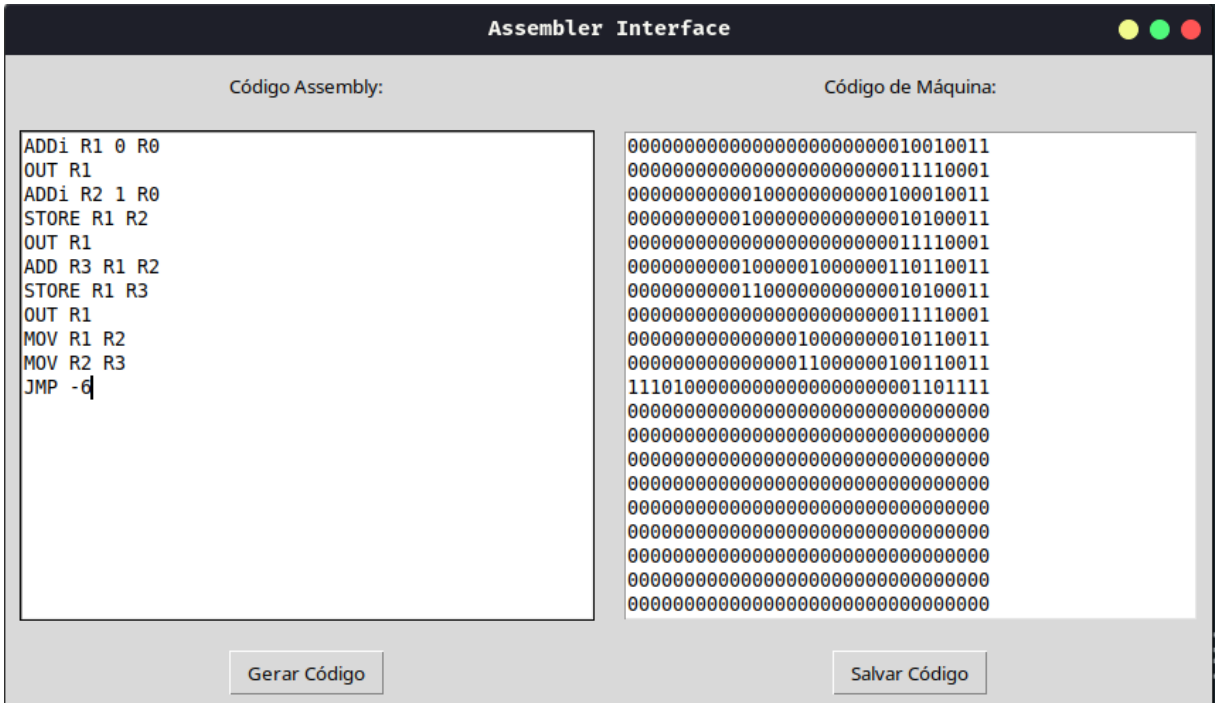


Figura 6 – Interface gráfica desenvolvida para o assembler.

4.2. Resultados e Implementação no Quartus e FPGA

Após a conversão do código assembly em código de máquina utilizando o assembler desenvolvido, a implementação foi realizada no Quartus, onde o código binário gerado será carregado, posteriormente, na FPGA para execução. O objetivo dessa etapa foi verificar se o processador, ao executar as instruções geradas, reproduzia corretamente a sequência de Fibonacci.

Para validar o funcionamento, utilizamos a ferramenta de simulação Waveform do Quartus, que nos permitiu observar as saídas geradas pelo processador em tempo real. Durante a simulação, pudemos monitorar os valores dos registradores e a saída final em cada ciclo de clock, garantindo que os valores gerados correspondiam às expectativas da sequência de Fibonacci.

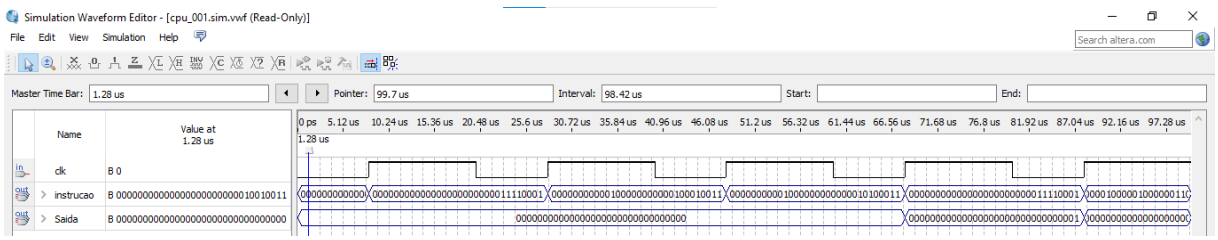


Figura 7 – Resultados obtidos na waveform.

Após a simulação anterior, o clock foi ajustado para incluir mais pulsos, permitindo a observação da sequência de Fibonacci. Em seguida, movemos o cursor para analisar os pulsos específicos em que os valores na saída podem ser observados, ou seja, os pulsos onde as instruções OUT são executadas.

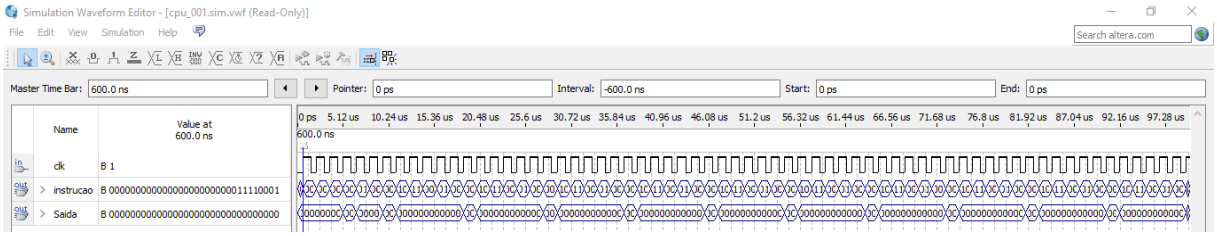


Figura 8 – Valor 1

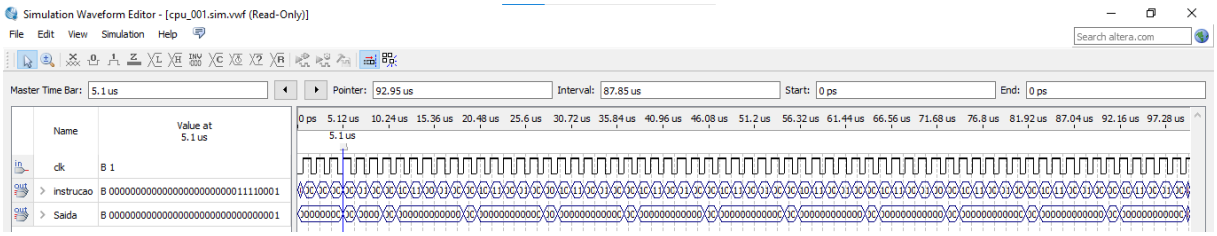


Figura 9 – Valor 2

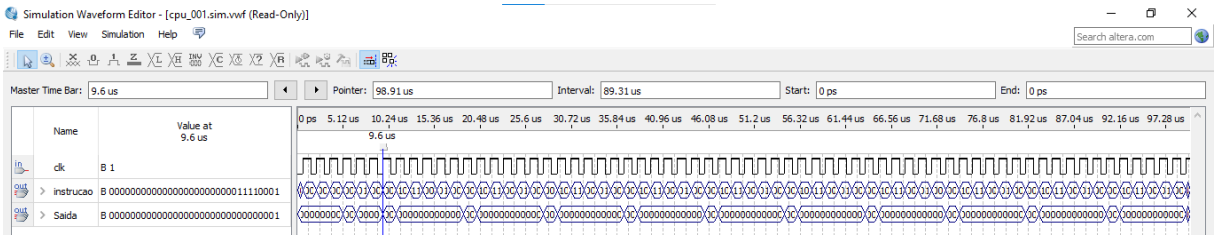


Figura 10 – Valor 3

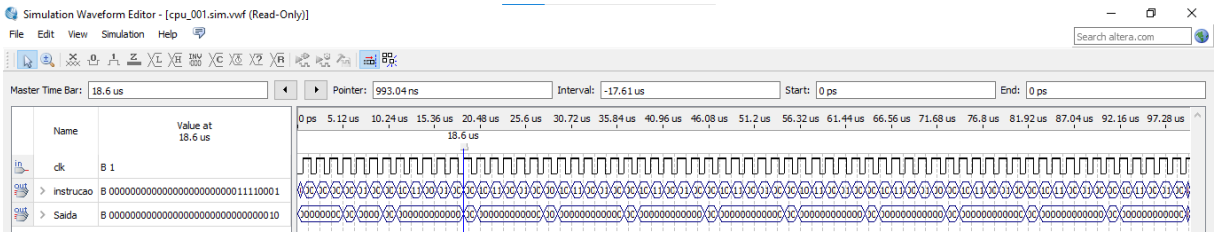


Figura 11 – Valor 4

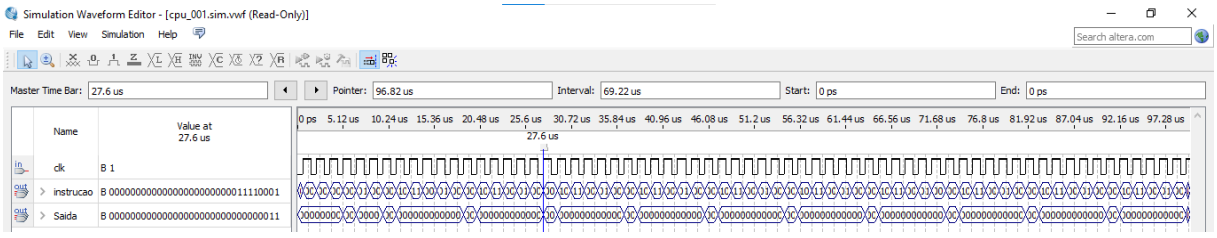


Figura 12 – Valor 5

Os resultados observados na Waveform confirmaram que o processador estava operando corretamente, com os valores exibidos coincidindo perfeitamente com os números da sequência de Fibonacci nas posições esperadas. Isso valida tanto a funcionalidade do assembler quanto a correta implementação e funcionamento do processador no Quartus.

Essa verificação por meio da simulação é crucial, pois demonstra que todas as etapas, desde a montagem do código até a execução no hardware, foram realizadas com precisão, garantindo a confiabilidade do sistema desenvolvido.
Por fim, podemos observar os resultados obtidos na FPGA, mostrando o primeiro e o quinto valor:

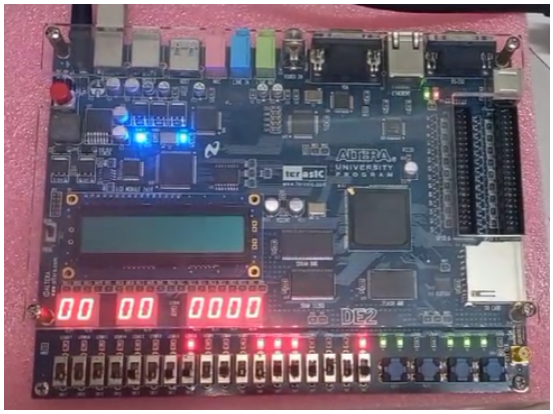


Figura 13 – Valor 1 visto na FPGA

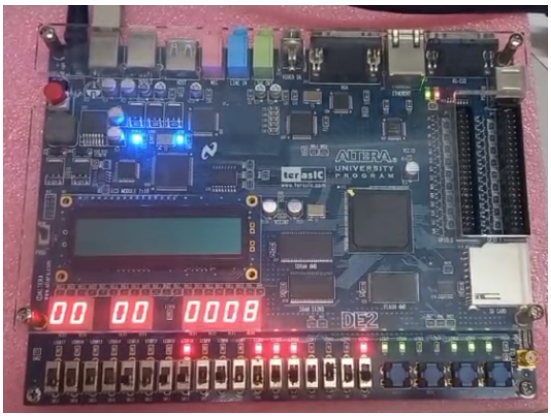


Figura 14 – Valor 5 visto na FPGA

5. CONCLUSÃO

Neste trabalho, apresentamos a implementação e validação de um processador de ciclo único, com foco no desenvolvimento do MMASM e na integração com a unidade central de processamento (CPU). O processo envolveu a construção de um assembler que converte o código assembly em código de máquina, a simulação do processador utilizando Quartus e a execução final na FPGA.
Os objetivos propostos foram alcançados com sucesso, demonstrando a eficácia tanto do assembler quanto do processador desenvolvido. A simulação da sequência de Fibonacci forneceu uma validação robusta das funcionalidades implementadas. A análise dos resultados mostrou que o processador executa corretamente o código assembly, gerando a sequência esperada e confirmando o funcionamento adequado através da visualização na ferramenta de simulação Waveform do Quartus e na FPGA.
Os diagramas lógicos detalhados e as tabelas da verdade fornecidas evidenciam o funcionamento interno dos blocos da CPU e garantem que a implementação atende aos requisitos especificados. A interface gráfica do assembler permitiu uma visão clara do processo de conversão, facilitando a validação e compreensão das operações realizadas pelo processador.
A verificação prática realizada na FPGA confirmou que o processador opera corretamente em um ambiente real, com os resultados obtidos correspondendo precisamente aos valores esperados. As figuras obtidas durante a simulação e a execução na FPGA reforçam a precisão e a confiabilidade do sistema desenvolvido.
Em resumo, o projeto demonstrou a capacidade de construir e validar um processador de ciclo único, integrando com sucesso as etapas de desenvolvimento do assembler, simulação e imple-

mentação em hardware. As escolhas de design e as técnicas empregadas mostraram-se eficazes, e os resultados obtidos corroboram a funcionalidade do processador e do MMASM.
Este trabalho não apenas consolidou o conhecimento teórico em arquitetura de computadores, mas também forneceu uma experiência prática valiosa na implementação de sistemas digitais. As lições aprendidas e os desafios superados contribuem significativamente para a compreensão e aplicação dos conceitos de design e operação de processadores.

Referências

[1] David Patterson and John Hennessy. *Computer Organization and Design: The Hardware/-Software Interface*. Morgan Kaufmann, 2017.

[2] M. Morris Mano and Charles R. Kime. *Digital Design and Computer Architecture*. Pearson, 2018.