

# Trabalho Prático 02 - Caminho de Dados do RISC-V

Ítallo Ferreira - 5101<sup>1</sup>, Manuel Ferreira - 5091<sup>2</sup>

<sup>1</sup>Universidade Federal de Viçosa - Campus Florestal (UFV-CAF)  
35.690-000 – Florestal – MG – Brasil

itallo.cardoso@ufv.br, manuel.simoes@ufv.br

**Abstract.** *This work presents a simplified implementation of the RISC-V datapath, implemented in a hardware description language, Verilog. The project follows a sequential datapath architecture with stages for instruction fetch, decoding, operand fetch, execution, and result storage. Registers are used to store operands and intermediate results, while control signals coordinate the operations in each stage. This implementation aims to provide a clear and accessible model for understanding the precise execution of instructions in the RISC-V architecture.*

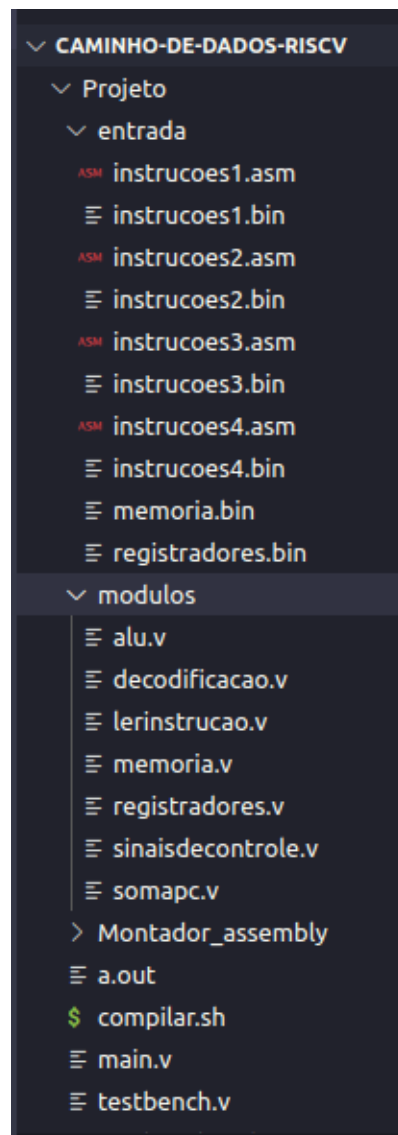
**Resumo.** *Este trabalho apresenta uma implementação simplificada do caminho de dados do RISC-V, implementado em uma linguagem de descrição de hardware, verilog. O projeto segue uma arquitetura de caminho de dados sequenciais, com estágios de busca, decodificação, busca de operandos, execução e armazenamento do resultado. Registradores são usados para armazenar os operandos e resultados intermediários, enquanto sinais de controle coordenam as operações em cada estágio. Essa implementação busca fornecer um modelo claro e acessível para entender a execução precisa de instruções na arquitetura RISC-V.*

## 1. Introdução

Neste trabalho, apresentamos a implementação do caminho de dados do RISC-V (RISC - Reduced Instruction Set Computer), disponível [aqui](#), utilizando a linguagem de descrição de hardware, Verilog. O caminho de dados, — em inglês, data path — é responsável pela execução das instruções do processador, envolvendo a busca, decodificação e execução de operações. Antes de tudo, é importante compreender o funcionamento de um processador, que é a unidade central de processamento de um computador (CPU) e desempenha o papel de cérebro do sistema, interagindo e estabelecendo as conexões necessárias para executar as instruções do computador.

Nesse contexto, o caminho de dados desempenha uma função crucial, pois ele é como o coração, onde as instruções são processadas e os dados são manipulados. Ele é composto por uma série de componentes interconectados, tais como registradores, unidades lógicas e aritméticas (ALUs), multiplexadores e memória, os quais trabalham em conjunto para realizar as operações necessárias. Com base nas informações apresentadas e aplicando os conceitos discutidos em sala de aula, foi possível realizar este trabalho prático.

## 2. Organização



**Figura 1. Organização**

Na figura 1 pode se ter uma visão geral da organização do trabalho. Na pasta Projeto/ estão todos os arquivos usados na criação do trabalho. A pasta "Ajuda/" contém o material de apoio que utilizamos durante o processo. A pasta "entrada/" contém os arquivos ".asm" que são a versão em assembly dos ".bin", e também os arquivos ".bin" que serão utilizados no caminho de dados. Já na pasta "modulos/" é onde estão os módulos em Verilog. O Diretório "Montador-assembly/" é o trabalho prático 1, em que é responsável pela leitura dos arquivos ".asm" e a tradução para linguagem de máquina. Por fim, temos a "main" onde acontece o gerenciamento do projeto, "compilar.sh" um script shell, ou seja, um arquivo de script executável, o "testbench.v" a simulação que cria o arquivo de onda "testbench.vsd" e imprimir no terminal do computador a memória de todos os 32 registradores.

### 3. Materiais e métodos

Antes de compreender os resultados obtidos, primeiro, é fundamental entender o funcionamento das ferramentas utilizadas, que desempenharam um papel crucial na viabilização deste trabalho, já que sem elas essa implementação não poderia ser feita.



**Figura 2. VSCode**

A IDE (Integrated Development Environment) VSCode (Visual Studio Code) é um editor de código-fonte desenvolvido pela Microsoft é um ambiente de desenvolvimento de código-fonte, gratuito, nós utilizamos esse editor para desenvolver o projeto, uma vez que ele tem um ótimo ambiente de desenvolvimento e de fácil utilização.



**Figura 3. GitHub**

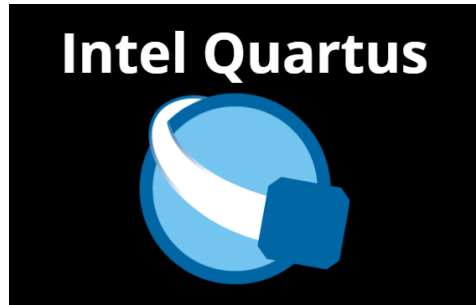
GitHub é uma plataforma de hospedagem de código-fonte e arquivos em nuvem, com controle de versão usando o Git. Utilizamos o GitHub para versionamento do código pensando em praticidade, pois essa plataforma permite que os membros da equipe podem contribuir para o mesmo projeto, enviando modificações, você pode facilmente rastrear as alterações feitas em cada arquivo, isso agiliza muito o processo já que a cada alteração feita, não há a necessidade de compartilhar um .zip com todos os membros da equipe.



**Figura 4. GTKWave**

O software GtkWave é uma ferramenta de visualização e análise de formas de onda (waveforms) geradas por simulações de circuitos digitais. Ele permite visualizar

e inspecionar os sinais elétricos e lógicos em um nível de detalhe granular, ajudando a depurar e verificar o comportamento do circuito. O GtkWave é amplamente utilizado em projetos de design de hardware para análise e depuração de circuitos digitais complexos.



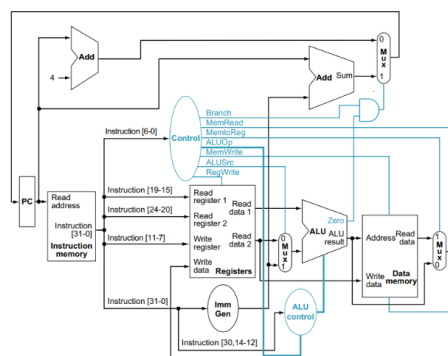
**Figura 5. Intel Quartus Prime**

O Intel Quartus Prime é um conjunto de ferramentas de design de hardware desenvolvido pela Intel. Ele é amplamente empregado no projeto, implementação e programação de dispositivos de lógica programável, como as FPGAs (Field-Programmable Gate Arrays). Além disso, tem suporte a linguagens como Verilog, VHDL e System Verilog, o Quartus Prime é uma ferramenta essencial para criar e gerenciar projetos. Durante a execução deste projeto, o Quartus Prime desempenhou um papel fundamental no desenvolvimento e sucesso do trabalho.

## **4. Desenvolvimento**

### **4.1. Funcionamento Geral**

Para implementação do caminho de dados foi dado como referência essa imagem.



**Figura 6. Caminho de dados**

Com ela foi possível a melhor compreensão do caminho de dados para sua implementação. Nossa implementação contém uma máquina de estado que definirá em cada estado uma bloco do caminho de dados é executado, contém também alguns estados para gerar um atrasado para os dados chegarem corretamente ao destino.

```

1 //parametros do estado
2     parameter IF = 4'b0000, //posição instrução
3             ID = 4'b0001, //leitura
4             EX = 4'b0010, //execução
5             AUX1 = 4'b0101, //auxiliar para atraso
6             AUX2 = 4'b1111, //auxiliar para atraso
7             MEM = 4'b0011, //leitura memoria
8             WB = 4'b0100, //escrita
9             AUX3 = 4'b0110, //auxiliar para atraso
10            AUX4 = 4'b0111, //auxiliar para atraso
11            SUMPc = 4'b1000, //soma pc
12            FIM = 4'b1001; //finish

```

**Figura 7. Parâmetros**

Estados:

IF: a instrução é busca na memória das instruções.

ID: aqui é feito a decodificação de cada campo da instrução.

MEM: é feito a leitura ou escrita na memória.

WB: onde é feita a escrita no banco de registradores.

SUMPc: onde é feito o cálculo do endereço da próxima instrução

FIM: após ler todas as instruções

todos estados com nome AUX é utilizado para gerar um atraso para os dados serem atualizados corretamente.

## 4.2. Módulo lerinstrucao.v

```

1 module lerinstrucao (instrucao, PC, clk, estado);
2     input wire [31:0] PC;
3     input wire clk;
4     input wire [3:0] estado;
5     output reg [31:0] instrucao;
6     // alterar conforme a quantidade de linhas do arquivo
7     reg [31:0] instrucoes [0:10]; // Memória de instruções
8
9     // Lendo instruções
10    // Especifique qual instrucoes.bin esta lendo
11    initial begin
12        $readmemb("entrada/instrucoes4.bin", instrucoes); // Lendo instruções em formato binário
13        instrucao <= instrucoes[PC];
14    end
15
16    // Atualizando instrução a cada ciclo de clock
17    always @(posedge clk) begin
18
19        if(estado == 4'b0000)begin
20            instrucao <= instrucoes[PC];
21        end
22    end
23
24 endmodule

```

**Figura 8. Leitura das instruções**

O módulo "lerinstrucao" apresentado é responsável por ler instruções de um arquivo de memória de instruções e fornecer a instrução correspondente ao contador de programa atual. Ele utiliza uma memória para armazenar as instruções e atualiza a instrução a cada ciclo de clock. Esse módulo é essencial para o funcionamento do processador, pois fornece as instruções que serão executadas. Um cuidado com esse módulo é o tamanho do array instruções que deve conter a quantidade de instruções - 1.

### 4.3. Módulo decodificacao.v

```

1 module decodificacao (instrucao, opcode, rd, rs1, rs2, funct3, funct7, immediate, tipo, clk, estado, negativo);
2   input wire [31:0] instrucao;
3   input wire clk;
4   input wire [3:0] estado;
5   output reg [6:0] opcode;
6   output reg [4:0] rd; // registrador de escrita
7   output reg [4:0] rs1; // registrador de leitura 1
8   output reg [4:0] rs2; // registrador de leitura 2
9   output reg [2:0] funct3;
10  output reg [6:0] funct7;
11  output reg [11:0] immediate;
12  output reg [2:0] tipo; // determinar de qual formato é a instrução
13  output reg negativo; // sinal de controle para saber se o imediato é negativo ou não
14
15  always @(posedge clk) begin
16    // estados de a decodificação é usada
17    if(estado == 4'b0001)begin
18      // recebendo os valores conforme o opcode de cada função
19      case (instrucao[6:4])
20        3'b000: begin //formato r
21          rd <= instrucao[11:7];
22          rs1 <= instrucao[19:15];
23          funct3 <= instrucao[14:12];
24          immediate <= instrucao[31:20];
25          negativo <= 1'b0;
26          tipo <= 3'b000;
27        end
28        3'b001: begin //formato I
29          rd <= instrucao[11:7];
30          rs1 <= instrucao[19:15];
31          funct3 <= instrucao[14:12];
32          if(instrucao[31] == 1'b1)begin // se o imediato for negativo
33            immediate <= (~instrucao[31:20]) + 1'b1;
34            negativo <= 1'b1;
35          end
36        end
37      endcase
38    end
39  end

```

Figura 9. decodificação

O código apresentado é um exemplo de um módulo de decodificação de instruções em Verilog. Ele recebe uma instrução de processador e extrai os campos relevantes, como opcode, registradores, funções e imediatos. O módulo utiliza um sinal de clock para executar a decodificação e possui diferentes casos para lidar com diferentes formatos de instrução. O objetivo do módulo é direcionar os campos extraídos para as saídas apropriadas, facilitando a execução correta das instruções pelo processador.

```

1 3'b110: begin //formato sb
2   if(instrucao[31] == 1'b1)begin // se o imediato for negativo
3     immediate <= (~instrucao[31], instrucao[7], instrucao[30:25], instrucao[11:8]) + 1;
4     negativo <= 1'b1;
5   end
6   else begin
7     immediate <= {instrucao[31], instrucao[7], instrucao[30:25], instrucao[11:8]} << 1;
8     negativo <= 1'b0;
9   end
10 end

```

Figura 10. Imediato Negativo

Na Figura 10 é mostrado o tratamento de imediato negativo, é verificado o bit mais significativo (bit 31) da instrução para determinar se o imediato é negativo. Se for negativo, é feita a conversão de complemento de dois do imediato negativo, adicionando 1 ao complemento de dois do valor absoluto do imediato e multiplicando o resultado por 2. O resultado é armazenado na variável "immediate". Além disso, a variável "negativo" é definida como 1'b1 para indicar que o imediato é negativo. Se o bit mais significativo não

for 1, significa que o imediato é positivo ou zero. Nesse caso, o imediato é deslocado para a esquerda em 1 bit, e o resultado é armazenado na variável "immediate". A variável "negativo" é definida como 1'b0 para indicar que o imediato não é negativo. Essa variável negativo irá para outros módulos indicando que é para subtrair o imediato caso alguma operação envolvendo ele seja feita.

#### 4.4. Módulo registradores.v



```
1 // Lendo os valores dos registradores usados na alu
2 assign readdata1R = bancoregistradores[rs1];
3 assign readdata2R = bancoregistradores[rs2];
4
5 // escrevendo no registrador
6 always @(posedge clk) begin
7     // estado onde o registrador é escrito
8     if ((estado == 4'b0110) || (estado == 4'b0111)) begin // Estado de execução
9         // regiwirte mostra se o registrador é escrito ou não
10        case (regiwrite)
11            1'b1: begin
12                // mentoreg mostra se o dado vem da memória ou da alu
13                case (mentoreg)
14                    1'b1: begin
15                        if (rd != 0) begin
16                            bancoregistradores[rd] <= readdata1;
17                        end
18                    end
19                    1'b0: begin
20                        if (rd != 0) begin
21                            bancoregistradores[rd] <= writedataR;
22                        end
23                    end
24                endcase
25            end
26        endcase
27    end
28 end
```

Figura 11. Parte do Módulo Registradores

O módulo "registradores" é responsável pela leitura e escrita dos registradores em um processador. Ele utiliza uma memória para armazenar os valores dos registradores e atualiza esses valores com base nos sinais de controle e dados fornecidos. Os valores dos registradores também são atribuídos a registradores específicos para permitir a visualização dos mesmos, ele é executado em dois estados diferentes para que os dados sejam atribuídos corretamente. Como os registradores são lidos do arquivo registradores.bin na pasta entrada é possível iniciar um registrador com algum valor. Esse módulo desempenha um papel fundamental no processador, garantindo a correta manipulação dos dados e controle dos registradores durante a execução das instruções.

## 4.5. Módulo sinaisdecontrole.v

```
1 module sinaisdecontrole (tipo, regwrite, memwrite, memread, alucontrol, funct3, clk, branch, memtoreg, alusrc, funct7, estado);
2   input wire [2:0] tipo; //tipo os 3 bits mais significativos da opcode para compensação
3   input wire [2:0] funct3;
4   input wire [6:0] funct7;
5   input wire clk;
6   input wire [3:0] estado;
7   output reg regwrite;
8   output reg [1:0] aluop;
9   output reg memwrite;
10  output reg memread;
11  output reg [3:0] alucontrol;
12  output reg branch;
13  output reg memtoreg;
14  output reg alusrc;
15
16  // gerando sinais de controle
17  always @(posedge clk) begin
18    // estado para gerar valores de controle para a alu realizar determinada operação
19    /*
20     inicialmente os sinais de controle de escrita e leitura são iniciados como don't care
21     para não ocorrer a escrita no registrador antes do resultado da alu estiver pronto
22     */
23    if(estado == 4'b0010) begin
24      case (tipo)
25        // sinais de controle para i
26        3'b000: begin //lw
27          regwrite <= 1'b1;
28          memwrite <= 1'b0;
29          memread <= 1'b1;
30          alucontrol <= 4'b0010;
31          branch <= 1'b0;
32          memtoreg <= 1'b1;
33          alusrc <= 1'b1;
34        end
```

Figura 12. Sinais de controle

O módulo "sinaisdecontrole" implementa a geração de sinais de controle para um processador. Esses sinais são cruciais para controlar as operações do processador, como operações aritméticas, acesso à memória e manipulação de registradores. O módulo recebe vários sinais de entrada, como o tipo de instrução, códigos de função e estado atual, e gera os sinais de controle apropriados com base nesses valores. Os sinais de controle são atualizados em cada borda de subida do sinal de clock. A implementação do módulo garante que os sinais de controle corretos sejam ativados para cada tipo de instrução e estado específico. Inicialmente os sinais de controle de escrita e leitura são tido como don't care, pois ao não atribuí-los assim, acontecia casos em que o resultado da Alu da instrução anterior era escrito duas vezes, uma na instrução anterior e outra no registrador destino da instrução atual, eles somente recebem o sinal de leitura e escrita quando o novo resultado da Alu está pronto. Esses sinais de controle são essenciais para o correto funcionamento do processador e garantem a execução adequada das instruções.



## 4.6. Módulo alu.v

```
1 always @(posedge clk) begin
2     //estados em a alu é usada
3     if ((estado == 4'b0101) || (estado == 4'b0110)) begin // Estado de execução
4         //alusrc define se a alu vai usar imediato ou não
5         case (alusrc)
6             1'b0: begin // operações para funções que não usam imediato
7                 //alucontrol define a operação que a alu vai fazer
8                 case (alucontrol)
9                     4'b0000: begin // and
10                        aluresult2 <= readdata1R & readdata2R;
11                        aluresult1 <= 1'b0;
12                    end
13                    4'b0001: begin // or
14                        aluresult2 <= readdata1R | readdata2R;
15                        aluresult1 <= 1'b0;
16                    end
17                    4'b0010: begin // soma
18                        aluresult2 <= readdata1R + readdata2R;
19                        aluresult1 <= 1'b0;
20                    end
21                    4'b0110: begin //subtração
22                        aluresult2 <= readdata1R - readdata2R;
23                        aluresult1 <= 1'b0;
24                    end
25                    4'b0100: begin // xor
26                        aluresult2 <= readdata1R ^ readdata2R;
27                        aluresult1 <= 1'b0;
28                    end
29                    4'b0101: begin // slr
30                        aluresult2 <= readdata1R >>> readdata2R;
31                        aluresult1 <= 1'b0;
32                    end
33                endcase
34            endcase
35        end
36    end
37 end
```

Figura 13. ALU

Nessa parte do código, é implementada a execução da ALU (Unidade Lógico-Aritmética). O sinal "alusrc" determina se o valor do imediato será utilizado ou não na realização da operação. Se "alusrc" for 1'b0, o imediato não será utilizado. O sinal "alucontrol" define a operação a ser realizada pela ALU, contendo as seguintes operações ADD, SUB, LW, SW, XOR, SLR, AND, OR, ADDI, BEQ, BNE. Os resultados das operações são armazenados em "aluresult2", enquanto "aluresult1" é definido após o resultado de uma instrução SB determinando se o desvio será tomado ou não.

```
1 4'b0011: begin // addi
2     if(negativo == 1'b1) begin
3         aluresult2 <= readdata1R - immediate; // subtraindo pois o imediato é negativo
4     end else begin
5         aluresult2 <= readdata1R + immediate; // somando pois o imediato é positivo
6     end
7     aluresult1 <= 1'b0;
8 end
9
```

Figura 14. Exemplo uso do negativo

Nessa parte do código é mostrado um local onde a registrador negativo é utilizado.

## 4.7. Módulo memoria.v

```
1 // atualizando a memoria a cada ciclo de clock
2 always @(posedge clk) begin
3     if((estado == 4'b0011) || (estado == 4'b0110) || (estado == 4'b0111)) begin
4         // mux para saber se vai ser escrito ou lido na memoria
5         if(memwrite == 1'b1) begin
6             memoria[aluresult2] <= readdata2R;
7         end
8         if(memread == 1'b1) begin
9             readdataM <= memoria[aluresult2];
10        end
11        // atualizando o valor que vai ser escrito no registrador
12        writedataR <= aluresult2;
```

Figura 15. Memória

Nessa parte do código, ocorre a atualização da memória e a manipulação dos dados para escrita e leitura. É verificado se ocorrerá uma operação de escrita ou leitura na memória. Se o sinal "memwrite" for 1, o valor presente em "readdata2R" é armazenado na posição de memória indicada por "aluresult2". Se o sinal "memread" for 1, o valor contido na posição de memória indicada por "aluresult2" é armazenado em "reddataM". Além disso, o valor que será escrito no registrador é atualizado com o resultado da operação realizada pela ALU, armazenado em "aluresult2". Como a memória é lida do arquivo memoria.bin na pasta entrada é possível inicializar alguma posição da memória.

## 4.8. Módulo somapc.v

```
1 // inicializa o PC
2 initial begin
3     PC <= 0;
4 end
5
6 // incrementa o PC
7 always @(posedge clk) begin
8     if(estado == 4'b1000) begin
9         // sinais de controle para o pc saber se vai ser incrementado com imediato ou não
10        case (pcsrc)
11            1'b0: begin
12                PC <= PC + 1; // proxima instrucao
13            end
14            1'b1: begin
15                // sinal de controle para saber se o imediato e negativo ou não
16                if(negativo == 1'b1) begin
17                    PC <= PC - (imediato/4); // caso haja desvio
18                end
19                else begin
20                    PC <= PC + (imediato/4); // caso haja desvio
21                end
22            end
23        endcase
24    end
25 end
```

Figura 16. Cálculo do PC

Nesse trecho do código, o contador de programa (PC) é inicializado com zero e posteriormente incrementado. O sinal "pcsrc" determina se o incremento será realizado com base no próximo endereço sequencial (quando pcsrc for 0) ou se haverá um desvio condicional com base no valor do imediato (quando pcsrc for 1). Se pcsrc for 0, o PC é incrementado em 1, avançando para a próxima instrução. Se pcsrc for 1, há um desvio condicional. O valor do imediato é verificado, e se for negativo (sinal "negativo" igual a 1), o PC é decrementado pelo valor absoluto do imediato dividido por 4. Caso contrário, se o imediato for positivo (sinal "negativo" igual a 0), o PC é incrementado pelo valor absoluto do imediato dividido por 4. Essas operações garantem a atualização adequada do PC, controlando o fluxo de execução do programa.

## 4.9. Módulo main.v

```
1 module main(clk, rst, reg0, reg1, reg2, reg3, reg4, reg5,
2 reg6, reg7, reg8, reg9, reg10, reg11, reg12, reg13, reg14,
3 reg15, reg16, reg17, reg18, reg19, reg20, reg21, reg22,
4 reg23, reg24, reg25, reg26, reg27, reg28, reg29, reg30,
5 reg31, mem0, mem1, mem2, mem3, mem4, mem5, mem6, mem7,
6 mem8, mem9, mem10, mem11, mem12, mem13, mem14, mem15,
7 mem16, mem17, mem18, mem19, mem20, mem21, mem22, mem23,
8 mem24, mem25, mem26, mem27, mem28, mem29, mem30, mem31);
9
10 input wire clk, rst;
11
12 //IF - para ler a instrução
13 wire [31:0] instrução;
14 wire [6:0] opcode;
15 wire [4:0] rd; // registrador de destino
16 wire [4:0] rs1; // registrador de leitura 1
17 wire [4:0] rs2; // registrador de leitura 2
18 wire [2:0] funct3;
19 wire [6:0] funct7;
20 wire [11:0] immediate;
21 wire [2:0] tipo; // tipo da instrução
22 wire [31:0] PC; // posição para ler a instrução
23 wire negativo; // usado para quando o immediate é negativo
24 //wire clk;
25
26 //ID - para ler os registradores
27 wire [31:0] readdata1; //R para indicar que pertence ao banco de registradores
28 wire [31:0] readdata2; //R para indicar que pertence ao banco de registradores
29 wire [31:0] writedata; //R para indicar que pertence ao banco de registradores
30 //registradores de 32 bits
31 output [31:0] reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7, reg8, reg9, reg10, reg11;
32 output [31:0] reg12, reg13, reg14, reg15, reg16, reg17, reg18, reg19, reg20, reg21, reg22, reg23;
33 output [31:0] reg24, reg25, reg26, reg27, reg28, reg29, reg30, reg31;
34
35 //EX - para executar a instrução
36 //resultado da soma
37 wire aluresult1;
38 wire [31:0] aluresult2;
```

Figura 17. Main

A module main é responsável pela execução do processador, ela coordena todas as, permitindo que as instruções sejam buscadas, decodificadas e executadas corretamente. Ela manipula os registradores, extrai informações das instruções e controla o fluxo de execução. É recebido como parâmetro da main os registradores e as posições da memória para poderem ser impressos no terminal ao final do testbench

## 4.10. Módulo testbench.v

```
1 initial begin
2     $dumpfile("testbench.vcd");
3     $dumpvars(0, caminho_tb);
4     $display("\nResultados Finais");
5     rst_tb = 0;
6     #1000;
7     rst_tb = 1;
8     #1
9     $display("-----Memoria-----");
10    $display("memoria [0] = %d", mem0);
11    $display("memoria [1] = %d", mem1);
12    $display("memoria [2] = %d", mem2);
13    $display("memoria [3] = %d", mem3);
14    $display("memoria [4] = %d", mem4);
15    $display("memoria [5] = %d", mem5);
16    $display("memoria [6] = %d", mem6);
17    $display("memoria [7] = %d", mem7);
```

Figura 18. Testbench

Essa parte do código é um testbench em para o módulo main. Ele simula o comportamento do módulo fornecendo sinais de clock e reset, e conectando as saídas do módulo a sinais de teste. O bloco initial begin configura a inicialização da simulação. Ele define os valores iniciais dos sinais, configura o arquivo de despejo para gerar um arquivo VCD e exibe os resultados finais da simulação. O sinal clk-tb é alternado entre 0 e 1 a cada unidade de tempo usando o bloco always, simulando o sinal de clock. A sequência de eventos na simulação é a seguinte: primeiro, o sinal de reset (rst-tb) é definido como 0.

Após um atraso de 1000 unidades de tempo, o sinal de reset é alterado para 1, ativando o reset, assim a máquina de estados ir para o estado FIM. Depois disso, os resultados finais da simulação são exibidos. São mostrados os valores da memória e dos registradores. Por fim, a simulação é finalizada. O objetivo desse testbench é verificar o comportamento do módulo main sob condições específicas de entrada e avaliar os resultados esperados.

Alguns pontos importantes sobre nossa implementação do caminho de dados é que somente as instruções de desvio condicional (BEQ, BNE) e a instrução ADDI aceitam imediato negativo. Nos resultados do nosso trabalho os dados estão em formato de complemento de dois. A memória só contém apenas 32 posições e acessos para endereços maiores que 32 o valor não será computado, pois contamos com apenas 32 posições de memória.

#### 4.11. Implementação no FPGA

Neste projeto, também apresentamos a execução de instruções em uma FPGA (Field-Programmable Gate Array). Foram utilizados três arquivos de instruções, nomeadamente `instrucoes1.bin`, `instrucoes2.bin` e `instrucoes3.bin`. Os resultados obtidos a partir da execução dessas instruções foram registrados e apresentados em formato hexadecimal.

O objetivo deste projeto foi implementar e executar um conjunto de instruções em uma FPGA para observar o comportamento e desempenho do sistema no processamento das operações. Cada arquivo de instruções contém uma sequência de instruções binárias que foram lidas e interpretadas pela FPGA.

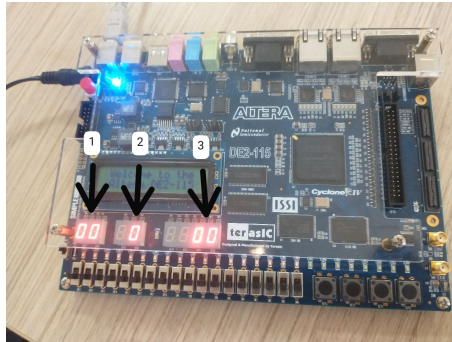
Para implementação na FPGA algumas alterações precisaram ser feitas, um sinal de reset foi adicionado que ao ser ativado ele irá inicializar todas as variáveis do caminho de dados, todos os registradores e memória como zero ou algum valor desejado, também as instruções são colocadas manualmente conforme as figuras.



```
1 always @(posedge clk) begin
2   if(reset == 1'b1) begin
3     instructions[0] <= 32'b000000000011000000000000100010011;
4     instructions[1] <= 32'b000000000010000000000000100010011;
5     instructions[2] <= 32'b0000000001000000000000001000001;
6     instructions[3] <= 32'b000000000000000000000000100010011;
7     instructions[4] <= 32'b000000000010000010000000010110011;
8     instructions[5] <= 32'b000000000010000010000000010110011;
9     instructions[6] <= 32'b010000000010000010000000010110011;
10    instructions[7] <= 32'b010000000010000010000000010110011;
11    instructions[8] <= 32'b0000000000100000100001100110011;
12    instructions[9] <= 32'b000000000001000010000000010110011;
13    instructions[10] <= 32'b00000000000100000010000000010011;
14    instructions[11] <= 32'b00000000001000001110000010110011;
15    instructions[12] <= 32'b00000000000000000000000010110011;
16    instructions[13] <= 32'b0000000000010000000000000000010011;
17    instructions[14] <= 32'b0000000000000000000000000000000;
18    instructions[15] <= 32'b0000000000000000000000000000000;
19  end
```

Figura 19. Exemplo do uso do reset

Aqui uma visão geral dos nossos diplays na FPGA



**Figura 20. displays usados na FPGA**

Na seta 1, está localizado dois display de sete segmentos que irá algum registrador escolhido no código, Na seta 2, está um display usado para sinalizar o final de todas as instruções, 0 para caminho de dados em execução, 1 para todas as instruções foram executadas, Por fim a seta 3 mostra displays representando o program counter. Foram usados dois displays para os registrados, dois displays para o program counter e um displays para o sinal de término das instruções. Os displays estão em formato hexadecimal, com os displays do PC e resgistradores podendo mostrar valores de 0 a 255.

```
1  always @(posedge clk) begin
2      //campo das unidades do pc
3      case (pc[3:0])
4          4'b0000: display1 <= 7'b1000000; // 0
5          4'b0001: display1 <= 7'b1111001; // 1
6          4'b0010: display1 <= 7'b0100100; // 2
7          4'b0011: display1 <= 7'b0110000; // 3
8          4'b0100: display1 <= 7'b0011001; // 4
9          4'b0101: display1 <= 7'b0010010; // 5
10         4'b0110: display1 <= 7'b0000010; // 6
11         4'b0111: display1 <= 7'b1111000; // 7
12         4'b1000: display1 <= 7'b0000000; // 8
13         4'b1001: display1 <= 7'b0010000; // 9
14         4'b1010: display1 <= 7'b0001000; // A
15         4'b1011: display1 <= 7'b0000011; // B
16         4'b1100: display1 <= 7'b1000110; // C
17         4'b1101: display1 <= 7'b0100001; // D
18         4'b1110: display1 <= 7'b0000110; // E
19         4'b1111: display1 <= 7'b0001110; // F
20     endcase
```

**Figura 21. Display program counter**

Na main.v você poderá escolher o registrador a ser mostrado no display na linha 116

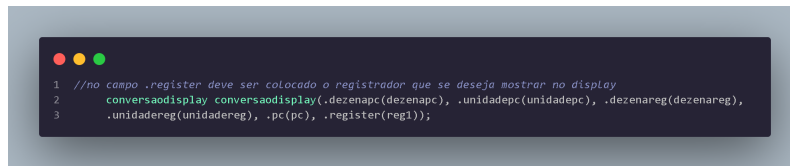


Figura 22. Mudar registrador a ser exibido no display

## 5. Resultados

Antes de apresentar os resultados obtidos, é importante mostrar como colocar a implementação em funcionamento, a fim de garantir uma compreensão completa do processo de execução. Ademais, será exibido os resultados, como arquivo de testbench(ou seja, simulação) mostrando no terminal, não só todos os 32 registradores, mas também os dados da memória, e além disso será disponibilizado o arquivo de onda gerado durante a simulação, permitindo uma análise aprofundada dos sinais e das interações dentro do sistema.



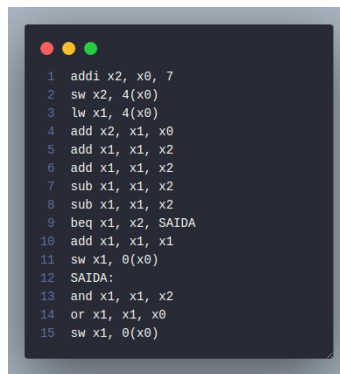
Figura 23. compile.sh

O arquivo "compile.sh" é um script shell executável, projetado para facilitar a compilação e execução do código. Ele contém uma sequência de comandos para automatizar o processo. Ao executar esse script, ele realizará os seguintes comandos: "iverilog testbench.v" ele é utilizado para compilar um testbench em Verilog. Após a compilação, o próximo passo é executar o arquivo de simulação gerado, "a.out", usando o comando "vvp a.out", isso iniciará a simulação do testbench, que executará as instruções definidas no código e gerará os resultados no terminal e o arquivo "testbench.vcd" para simulação.

Por fim, o comando "gtkwave testbench.vcd" é usado para abrir o arquivo de onda VCD (Value Change Dump) no software GTKWave para visualização das ondas, dentro da ferramenta GTKWave, você pode navegar pelos sinais, ampliar ou reduzir a escala temporal, ajustar a exibição dos sinais e realizar análises, como medição de tempos, valores de registradores, busca de padrões e muito mais.

### 5.1. Testbench

Para realizar a execução do projeto foi utilizado as seguintes instruções:



```

1 addi x2, x0, 7
2 sw x2, 4(x0)
3 lw x1, 4(x0)
4 add x2, x1, x0
5 add x1, x1, x2
6 add x1, x1, x2
7 sub x1, x1, x2
8 sub x1, x1, x2
9 beq x1, x2, SAIDA
10 add x1, x1, x1
11 sw x1, 0(x0)
12 SAIDA:
13 and x1, x1, x2
14 or x1, x1, x0
15 sw x1, 0(x0)

```

**Figura 24. instrucoes.asm**

Nesta figura, pode-se observar as instruções em assembly que serão transformadas em linguagem de máquina pelo montador assembly, construído no TP01, essas instruções podem ser encontradas também na especificação do trabalho prático, [aqui](#).



```

1 000000000111000000000000100010011
2 0000000001000000010001000100011
3 0000000001000000001000010000011
4 00000000000000000100000100110011
5 0000000001000001000000010110011
6 0000000001000001000000010110011
7 0100000001000001000000010110011
8 0100000001000001000000010110011
9 0000000001000001000011001100011
10 0000000001000001000000010110011
11 0000000000100000010000000100011
12 000000000010000011110000010110011
13 00000000000000001110000010110011
14 0000000000100000010000000100011

```

**Figura 25. instrucoes em linguagem de máquina**

Aqui podemos ver as instruções codificadas em linguagem de máquina prontas para serem processadas no caminho de dados do RISC-V. Como visto anteriormente, após a execução, o arquivo de simulação é gerado, "a.out", os resultados serão apresentados no terminal do computador a memória de todos os 32 registradores, juntamente com os dados da memória.

Nas imagens anteriores, é possível observar os valores de cada um dos registradores e da memória ao final da execução. Com o objetivo de verificar a precisão desses dados, utilizamos um interpretador chamado RISC-V Interpreter, desenvolvido pela Cornell University, para realizar uma comparação dos resultados obtidos. Essa breve descrição destaca que nas imagens anteriores foram apresentados os valores dos registradores e da memória ao final da execução. Além disso, menciona que foi realizado um processo de comparação utilizando um interpretador do RISC-V para validar a correteza dos dados.

Conforme observado na Figura 23 e na Figura 24, os resultados dos registradores "x1" e "x2" são exatamente iguais aos apresentados na Figura 22. Esses resultados são obtidos após o processamento no caminho de dados. Portanto, podemos afirmar com certeza que a implementação do projeto foi bem-sucedida e que os dados fornecidos são verídicos. Essa versão revisada ressalta que os resultados dos registradores são consistentes entre as figuras mencionadas, demonstrando a confiabilidade dos dados e a confirmação do su-

```

-----Memoria-----
memoria [0] = 7
memoria [1] = 7
memoria [2] = 0
memoria [3] = 0
memoria [4] = 0
memoria [5] = 0
memoria [6] = 0
memoria [7] = 0
memoria [8] = 0
memoria [9] = 0
memoria [10] = 0
memoria [11] = 0
memoria [12] = 0
memoria [13] = 0
memoria [14] = 0
memoria [15] = 0
memoria [16] = 0
memoria [17] = 0
memoria [18] = 0
memoria [19] = 0
memoria [20] = 0
memoria [21] = 0
memoria [22] = 0
memoria [23] = 0
memoria [24] = 0
memoria [25] = 0
memoria [26] = 0
memoria [27] = 0
memoria [28] = 0
memoria [29] = 0
memoria [30] = 0
memoria [31] = 0

```

**Figura 26. Dados da memória**

```

-----Registradores-----
Registrador [0] = 0
Registrador [1] = 7
Registrador [2] = 7
Registrador [3] = 0
Registrador [4] = 0
Registrador [5] = 0
Registrador [6] = 0
Registrador [7] = 0
Registrador [8] = 0
Registrador [9] = 0
Registrador [10] = 0
Registrador [11] = 0
Registrador [12] = 0
Registrador [13] = 0
Registrador [14] = 0
Registrador [15] = 0
Registrador [16] = 0
Registrador [17] = 0
Registrador [18] = 0
Registrador [19] = 0
Registrador [20] = 0
Registrador [21] = 0
Registrador [22] = 0
Registrador [23] = 0
Registrador [24] = 0
Registrador [25] = 0
Registrador [26] = 0
Registrador [27] = 0
Registrador [28] = 0
Registrador [29] = 0
Registrador [30] = 0
Registrador [31] = 0

```

**Figura 27. Dados dos registradores**

cesso na implementação do projeto.

## 5.2. Arquivo de onda

O comportamento dos sinais ao longo do tempo, conforme ilustrado na Figura 24, durante a execução dos dados da Figura 20, está corretamente refletido nos resultados apresentados. Os valores e transições dos sinais registrados no arquivo de ondas estão em conformidade com as expectativas e o comportamento esperado do sistema simulado.

Ao analisar o arquivo de ondas, podemos observar a sequência de instruções. Na instrução "addi", o valor 7 é adicionado ao registrador "x0" e o resultado é armazenado em "x2". Em seguida, na instrução "sw", o valor de "x2" é escrito na memória, no endereço



## RISC-V Interpreter

Input your RISC-V code here:

```

1  addi x2, x0, 7
2  sw x2, 4(x0)
3  lw x1, 4(x0)
4  add x2, x1, x0
5  add x1, x1, x2
6  add x1, x1, x2
7  sub x1, x1, x2
8  beq x1, x2, SAIDA
9  add x1, x1, x1
10 sw x1, 0(x0)
11 SAIDA:
12 and x1, x1, x2
13 or x1, x1, x0
14 sw x1, 0(x0)
15
16

```

Reset Stop CPU: 32 Hz

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
0	x1 (ra)	7	0x00000007	0b0000000000000000000000000000111
0	x2 (sp)	7	0x00000007	0b0000000000000000000000000000111
0	x3 (gp)	0	0x00000000	0b0000000000000000000000000000000
0	x4 (tp)	0	0x00000000	0b0000000000000000000000000000000
0	x5 (t0)	0	0x00000000	0b0000000000000000000000000000000
0	x6 (t1)	0	0x00000000	0b0000000000000000000000000000000
0	x7 (t2)	0	0x00000000	0b0000000000000000000000000000000
0	x8 (s0/fp)	0	0x00000000	0b0000000000000000000000000000000
0	x9 (s1)	0	0x00000000	0b0000000000000000000000000000000
0	x10 (a0)	0	0x00000000	0b0000000000000000000000000000000
0	x11 (a1)	0	0x00000000	0b0000000000000000000000000000000
0	x12 (a2)	0	0x00000000	0b0000000000000000000000000000000

Figura 28. RISC-V Interpreter Registradores

Memory Address	Decimal	Hex	Binary
0x00000000	7	0x00000007	0b0000000000000000000000000000111
0x00000004	7	0x00000007	0b0000000000000000000000000000111
0x00000008	0	0x00000000	0b0000000000000000000000000000000
0x0000000c	0	0x00000000	0b0000000000000000000000000000000
0x00000010	0	0x00000000	0b0000000000000000000000000000000
0x00000014	0	0x00000000	0b0000000000000000000000000000000
0x00000018	0	0x00000000	0b0000000000000000000000000000000
0x0000001c	0	0x00000000	0b0000000000000000000000000000000
0x00000020	0	0x00000000	0b0000000000000000000000000000000
0x00000024	0	0x00000000	0b0000000000000000000000000000000

Figura 29. RISC-V Interpreter Memória

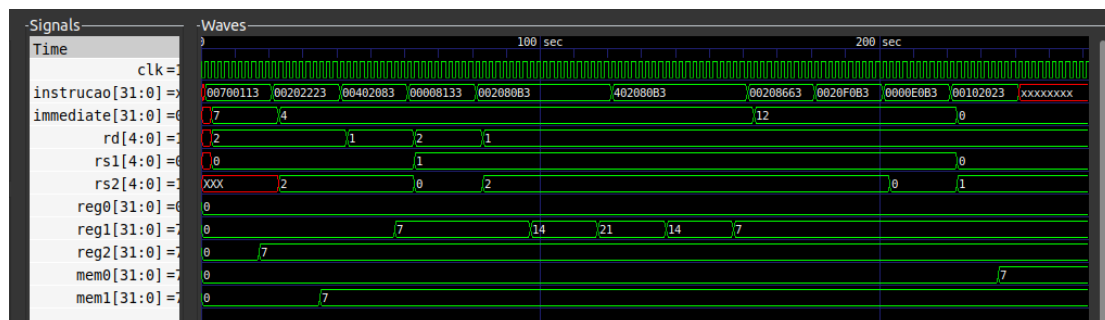


Figura 30. testbench.vcd

determinado pelo registrador "x0" mais o deslocamento 4. Esse valor é claramente visível no sinal do componente "mem01". Durante a execução, várias instruções aritméticas de adição e subtração são realizadas, depois, na instrução "beq" que permite desviar o fluxo de execução do programa para um determinado endereço se os valores dos registradores "x1" e "x2" forem iguais, como são a execução é direcionada para o rótulo "SAIDA", onde é executados duas operações lógicas "and" e "or" e a instrução de escrita na memória "sw", em que o valor de "x1" é escrito em "mem01".

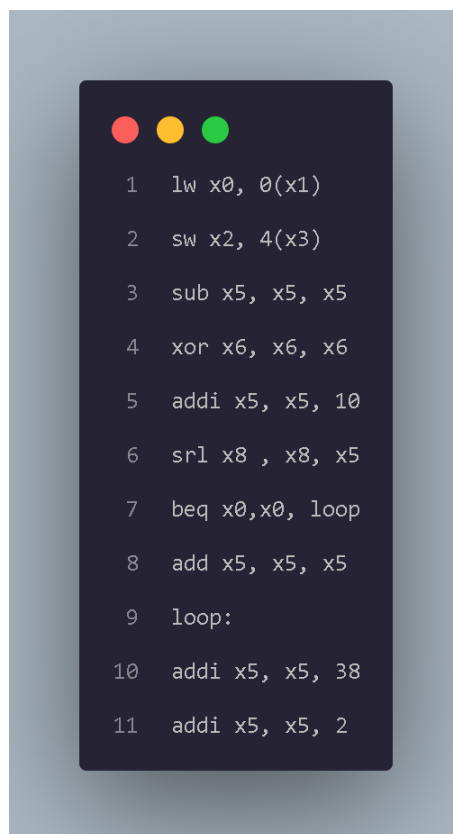
Portanto, ao considerar o arquivo de ondas em questão, podemos afirmar que os resultados apresentados nele são considerados corretos, fornecendo uma representação

precisa e confiável do comportamento dos sinais durante a simulação.

### 5.3. Resultados FPGA

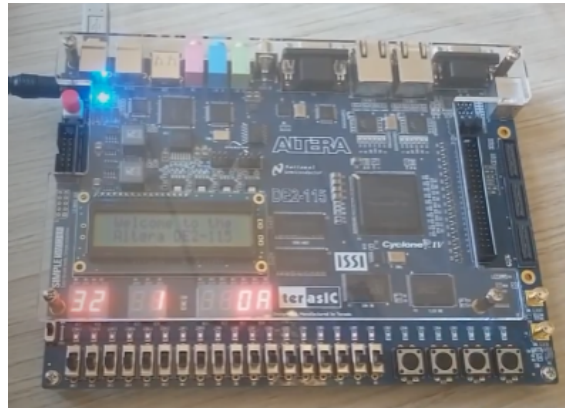
Aqui iremos apresentar algumas instruções executadas na FPGA e seus resultados implementados. As instruções contidas nos arquivos `instrucoes1.bin`, `instrucoes2.bin` e `instrucoes3.bin` foram cuidadosamente selecionadas para demonstrar diversas operações e fluxos de controle. As instruções incluíam operações aritméticas, operações lógicas, desvios condicionais e incondicionais, acesso à memória de dados, entre outras. E como dito anteriormente os resultados da execução das instruções foram registrados e apresentados em formato hexadecimal.

Primeiramente executando as instruções do arquivo `instrucoe1.bin`, abaixo está o assembler da `instrucoes1.bin`



**Figura 31. Assembler do primeiro arquivo de instruções**

Apresentando resultados na FPGA do primeiro arquivo de instrução. No display destinado ao registrador está representando o registrador x5.



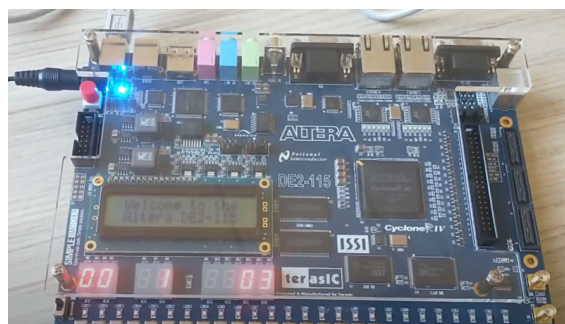
**Figura 32. Resultado do primeiro arquivo de instruções na FPGA**

Agora apresentando o assembler do arquivo instrucoe2.bin



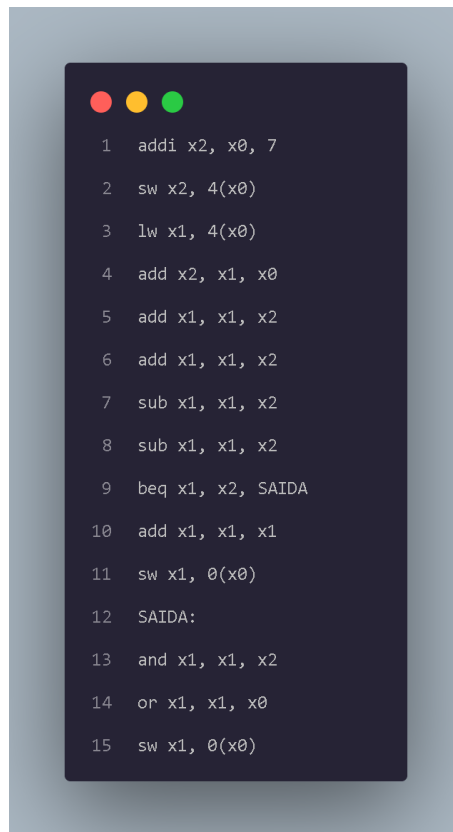
**Figura 33. Assembler do segundo arquivo de instruções**

Apresentando resultados na FPGA do segundo arquivo de instrução. No display destinado ao registrador está representando o registrador x7.



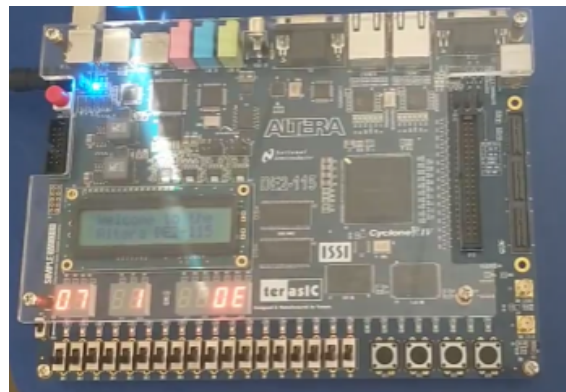
**Figura 34. Resultado do segundo arquivo de instruções na FPGA**

Por fim apresentando resultado do último arquivo implementado, abaixo está o assembler do arquivo instrucoes3.bin



**Figura 35. Assembler do terceiro arquivo de instruções**

Apresentando resultados na FPGA do segundo arquivo de instrução. No display destinado ao registrador está representando o registrador x1.



**Figura 36. Resultado do terceiro arquivo de instruções na FPGA**

A execução das instruções na FPGA foi bem-sucedida, demonstrando o correto funcionamento da arquitetura e dos módulos implementados. O projeto proporcionou uma compreensão aprofundada do processamento de instruções em um sistema digital, bem como dos desafios envolvidos na implementação de uma CPU em uma FPGA. O projeto possibilitou a aplicação prática dos conceitos teóricos estudados em arquitetura de computadores. Além disso, a execução das instruções e os resultados obtidos foram fundamentais para validar o funcionamento correto do sistema.

## 6. Considerações Finais

Durante o desenvolvimento deste trabalho, tivemos acesso a valiosos recursos fornecidos no curso, além de aplicarmos o conhecimento adquirido durante as aulas. Em conclusão, ao implementar o caminho de dados do RISC-V em Verilog, não apenas compreendemos a sequência de estágios necessários para a execução das instruções, mas também obtivemos resultados consistentes e significativos.

Além disso, foi um trabalho meticulosamente elaborado e incrivelmente interessante de se pensar e chegar à solução. A tarefa revelou-se desafiadora, exigindo empenho para percorrer todos os passos necessários e concluir o projeto. Ao longo desse processo, foi possível identificar uma série de erros, alguns dos quais provaram ser bastante desafiadores de corrigir.

## 7. References

[Patterson and Hennessy 2021] [Patterson and Waterman 2019] [Jesser 2021]  
[Vakil 2022] [University 2023] [chipverify 2015]

## Referências

- [chipverify 2015] chipverify (2015). Chipverif. <https://www.chipverify.com/verilog/>. [Online; acesso em 4-Junho-2023].
- [Jesser 2021] Jesser (2021). Datapath. <https://jesse-r-s-hines.github.io/RISC-V-Graphical-Datapath-Simulator/>. [Online; acesso em 1-Junho-2023].
- [Patterson and Hennessy 2021] Patterson, D. A. and Hennessy, J. L. (2021). *Computer Organization and Design RISC-v Edition*. Morgan-Kaufmann, 2th edition.
- [Patterson and Waterman 2019] Patterson, D. A. and Waterman, A. (2019). *Guia Prático RISC-V*. Morgan-Kaufmann, 1th edition.
- [University 2023] University, C. (2023). Risc-v interpreter. <https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/>. [Online; acesso em 3-Junho-2023].
- [Vakil 2022] Vakil, K. (2022). Venus. <https://venus.kvakil.me/>. [Online; acesso em 18-Abril-2023].