



Universidade Federal de Campina Grande – UFCG
Centro de Engenharia Elétrica e Informática – CEEI
Departamento de Engenharia Elétrica – DEE

Título:

Relatório da Sprint 10: Projeto RISC-V.

Disciplina:

Laboratório de Arquitetura de Sistemas Digitais

Professor(a):

Gutemberg Gonçalves Dos Santos Júnior

Aluna:

Heloysa Veloso Fernandes

Matrícula: 120110951

heloysa.fernandes@ee.ufcg.edu.br

Turma: 01

Outubro / 2024

ÍNDICE

1	Introdução	7
2	Objetivos Gerais	8
3	Fundamentação Teórica	9
3.1	Fórmula de Fibonacci	9
3.2	Números de Fibonacci	10
3.3	Sequência de Fibonacci e o retângulo de ouro	11
3.4	A sequência de Fibonacci na natureza	12
3.5	Aplicações da sequência	14
4	Módulos do Projeto	15
4.1	Registrador PC (Program Counter)	15
4.2	Instruction Memory (Inst. Mem.)	15
4.3	Control Unit	16
4.4	Register File	17
4.5	ULA	18
4.6	Data Memory	19
4.7	ParallelOUT	20
4.8	ParallelIN	21
5	Metodologia	22
5.1	Instrução BNE	22
5.2	Instrução ORI	23
5.3	Instrução JAL	24



5.4	Código Assembly	25
6	Desenvolvimento do Projeto	26
6.1	Código Mod_Teste	30
7	Conclusão	35
8	Referências	36

ÍNDICE DE FIGURAS

Figura 1: Retângulo de Ouro.	12
Figura 2: Espiral de Fibonacci.	12
Figura 3: Espiral de Fibonacci – Concha em espiral.	13
Figura 4: Espiral de Fibonacci – Galho de Planta.	13
Figura 5: Espiral de Fibonacci – Furacão.	14
Figura 6: Módulo Pc.	15
Figura 7: Módulo Instr. Memory.	15
Figura 8: Módulo Register File.	18
Figura 9: Módulo ULA.	19
Figura 10: Módulo Data Memory.	20
Figura 11: Módulo Parallel Out.	21
Figura 12: Módulo Parallel In.	22
Figura 13: Tipo – B.	23
Figura 14: Tipo – I.	24
Figura 15: Tipo – J.	24
Figura 16: Código em Assembly RISC-V utilizado para calcular a sequência de Fibonacci.	25
Figura 17: Código do Módulo Div_Freq.	26
Figura 18: Código do Módulo Inst_Memory.	26
Figura 19: Código do Módulo ControlUnit.	29

Figura 20: Mod_Teste – Parte 1.	30
Figura 21: Mod_Teste – Parte 2.	31
Figura 22: Mod_Teste – Parte 3.	32
Figura 23: Mod_Teste – Parte 4.	32
Figura 24: Mod_Teste – Parte 5.	33
Figura 25: Mod_Teste – Parte 6.	33
Figura 26: Mod_Teste – Parte 7.	34



ÍNDICE DE TABELAS

Tabela 1: 30 Primeiros Números de Fibonacci.....	10
--	----

1 INTRODUÇÃO

Este projeto, desenvolvido no âmbito da disciplina de Laboratório de Arquitetura de Sistemas Digitais, tem como foco a implementação de um processador RISC-V de ciclo único capaz de calcular a sequência de Fibonacci, uma série numérica em que cada termo é a soma dos dois anteriores, iniciando pelos valores 0 e 1. A sequência de Fibonacci é amplamente utilizada em diversas áreas da ciência e da engenharia, devido às suas características matemáticas singulares e sua presença em fenômenos naturais, o que a torna um exemplo ideal para o estudo de arquiteturas de hardware.

A escolha deste algoritmo não é apenas pela sua simplicidade, mas também pelo seu valor pedagógico. Ele oferece uma oportunidade clara de explorar o fluxo de controle e o uso eficiente de recursos computacionais em processadores de arquitetura RISC-V. A implementação do cálculo da sequência de Fibonacci requer o uso de instruções fundamentais de controle de fluxo e operações aritméticas, permitindo que conceitos essenciais da arquitetura de computadores sejam colocados em prática de maneira clara e objetiva.

O presente relatório foi elaborado para atender às exigências da Sprint 10 da disciplina, que propõe o desenvolvimento de um projeto prático destinado à aplicação dos conhecimentos adquiridos ao longo do curso. Essa atividade permite consolidar os princípios teóricos discutidos em sala de aula e fomentar o desenvolvimento de habilidades técnicas, com ênfase na implementação e análise de sistemas digitais. Por meio deste projeto, os conceitos de arquitetura de processadores, pipeline e execução de instruções são explorados, proporcionando uma visão prática e integrada da área.

2 OBJETIVOS GERAIS

O objetivo principal deste projeto é desenvolver um processador RISC-V de ciclo único capaz de calcular a sequência de Fibonacci, utilizando instruções fundamentais da arquitetura. Este projeto visa consolidar os conhecimentos teóricos adquiridos na disciplina de Laboratório de Arquitetura de Sistemas Digitais e aplicá-los em um cenário prático, proporcionando uma melhor compreensão do funcionamento de um processador e suas instruções.

Especificamente, o projeto tem como objetivos:

- Implementar um processador RISC-V de ciclo único capaz de executar o algoritmo da sequência de Fibonacci.
- Explorar o uso de instruções aritméticas e de controle de fluxo, como bne, jal, andi e ori, dentro da arquitetura RISC-V.
- Simular e validar o funcionamento do processador utilizando o ambiente Quartus II, garantindo a correta execução do algoritmo.
- Consolidar o aprendizado de conceitos como fluxo de controle, pipeline de execução, e instruções personalizadas em arquiteturas RISC-V.
- Proporcionar uma experiência prática na implementação e análise de sistemas digitais, fortalecendo as habilidades necessárias para o desenvolvimento de arquiteturas de computadores.

3 FUNDAMENTAÇÃO TEÓRICA

A sequência de Fibonacci é uma sequência numérica em que cada termo a partir do terceiro é a soma dos dois antecessores. O primeiro termo da sequência de Fibonacci é o número 1 e o segundo termo também é o número 1. O terceiro termo é 2, pois $1+1=2$. Já o quarto termo é 3, pois $1+2=3$. E assim sucessivamente.

Estudos sugerem que essa sequência foi atribuída a Fibonacci por causa de uma situação/problema em um de seus livros sobre a reprodução de uma população de coelhos.

Sequência de Fibonacci = 1,1,2,3,5,8,13,21,34,55...

3.1 FÓRMULA DE FIBONACCI

Uma maneira de expressar cada um dos termos da sequência de Fibonacci é através de uma fórmula que atribui cada termo à soma dos dois anteriores, ou seja, o n -ésimo termo da sequência é igual à soma dos termos $n-1$ e $n-2$, para todo número n natural maior do que 2.

Logo, os termos da sequência de Fibonacci são dados de forma recursiva:

$$F_n = F_{n-1} + F_{n-2}$$

A fórmula da sequência de Fibonacci é a forma algébrica de dizer que um termo qualquer na sequência é a soma dos seus dois anteriores.

3.2 NÚMEROS DE FIBONACCI

A seguinte tabela apresenta os 30 primeiros Números de Fibonacci e entre eles os números primos de Fibonacci.

Tabela 1: 30 Primeiros Números de Fibonacci.

Ordem/Índice	Número de Fibonacci (decimal)	Número de Fibonacci (hexadecimal)	Número de Fibonacci (binário)
F1	0	0	0
F2	1	1	1
F3	1	1	1
F4	2	2	10
F5	3	3	11
F6	5	5	101
F7	8	8	1000
F8	13	D	1101
F9	21	15	10101
F10	34	22	100010
F11	55	37	110111
F12	89	59	1011001
F13	144	90	10010000
F14	233	E9	11101001
F15	377	179	101111001

F16	610	262	1001100010
F17	987	3DB	1111011011
F18	1597	63D	11000111101
F19	2584	A18	101000011000
F20	4181	1055	1000001010101
F21	6765	1A6D	1101001101101
F22	10946	2AC2	10101011000010
F23	17711	452F	100010100101111
F24	28657	6FF1	110111111110001
F25	46368	B520	1011010100100000
F26	75025	12511	10010010100010001
F27	121393	1DA31	11101101000110001
F28	196418	2FF42	101111111101000010
F29	317811	4D973	1001101100101110011
F30	514229	7D8B5	1111101100010110101

3.3 SEQUÊNCIA DE FIBONACCI E O RETÂNGULO DE OURO

A partir dessa sequência, pode ser construído um retângulo, chamado de Retângulo de Ouro, uma representação visual da sequência.

No desenvolvimento do retângulo de ouro, dois quadrados com lados de unidade 1 são postos lado a lado. Eles representam os dois primeiros termos da sucessão.

Como o terceiro termo é $1 + 1 = 2$, um terceiro quadrado com lado medindo 2 unidades é desenhado. Continuando, temos a representação do retângulo de ouro.

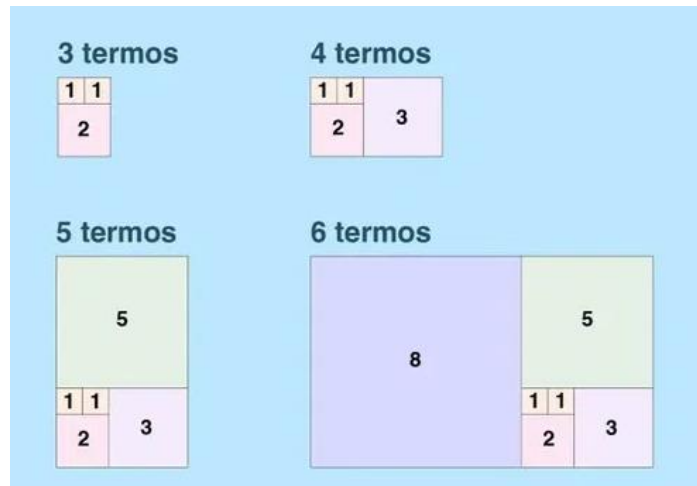


Figura 1: Retângulo de Ouro.

Ao desenhar um arco dentro desse retângulo, obtemos, por sua vez, a Espiral de Fibonacci.

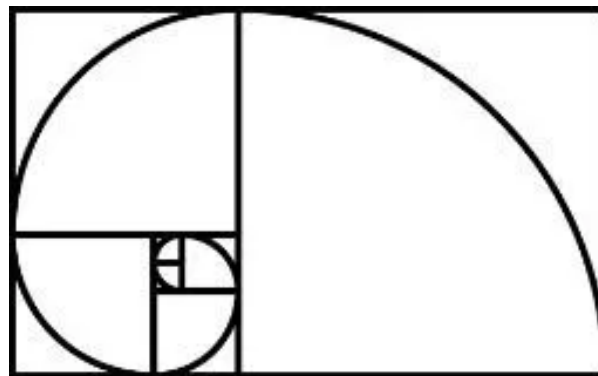


Figura 2: Espiral de Fibonacci.

3.4 A SEQUÊNCIA DE FIBONACCI NA NATUREZA

A verdade é que a sequência de Fibonacci pode ser percebida na natureza. São exemplos disso as folhas das árvores, as pétalas das rosas, os frutos como o abacaxi, as conchas espiraladas dos caracóis ou as galáxias.

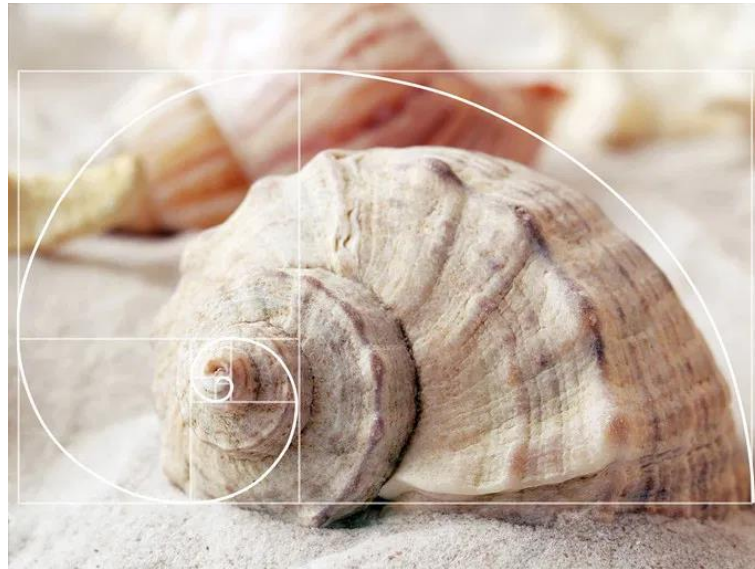


Figura 3: Espiral de Fibonacci – Concha em espiral.

Ao desenvolver o retângulo de ouro e a espiral de Fibonacci, é possível fazer uma comparação de sua forma com diversas outras na natureza.



Figura 4: Espiral de Fibonacci – Galho de Planta.

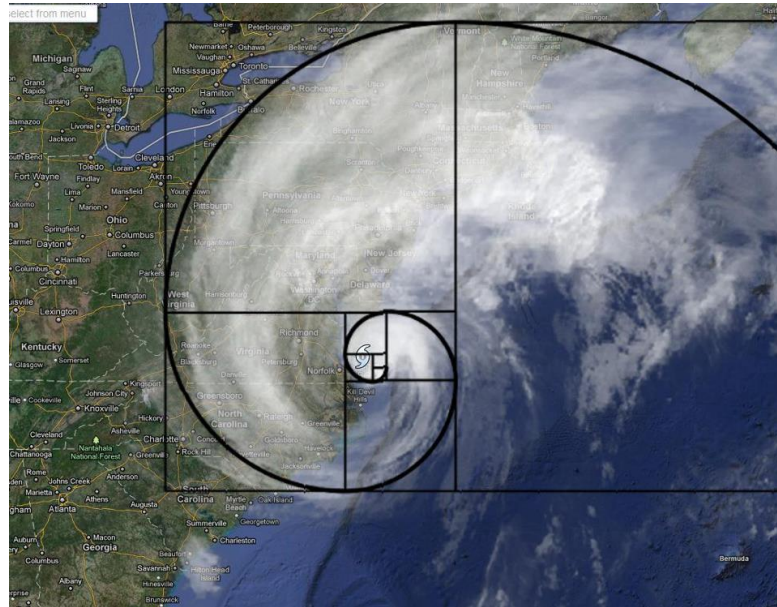


Figura 5: Espiral de Fibonacci – Furacão.

3.5 APLICAÇÕES DA SEQUÊNCIA

A sequência de Fibonacci tem uma ampla gama de aplicações em diversos campos, desde matemática pura até ciências aplicadas e até mesmo em aspectos estéticos e de design.

Muito interessante é que através do coeficiente de um número com o seu antecessor, obtém-se a constante com o valor aproximado de 1,618.

A sequência de Fibonacci é aplicada em análises financeiras e na informática, sendo utilizada por Da Vinci, que chamou a sequência de Divina Proporção, para fazer desenhos perfeitos.

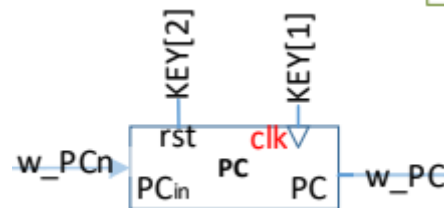
Na Biologia, a sequência descreve padrões de crescimento em plantas e animais. A arte, o design e a arquitetura se inspiram para criar desde obras de arte a construções. No campo da matemática e computação, ela contribui com aplicações na teoria dos números, estudos de sequências, algoritmos e ciência da computação.

4 MÓDULOS DO PROJETO

4.1 REGISTRADOR PC (PROGRAM COUNTER)

Este módulo controla o fluxo de execução das instruções.

Figura 6: Módulo Pc.

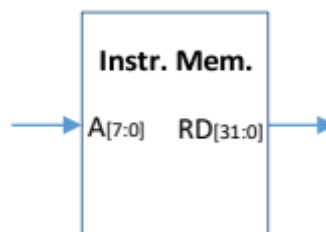


Tal componente é um registrador de 8 bits com uma entrada de clock (clk), uma entrada de reset (rst), uma entrada para carregamento paralelo (PCin) e uma saída paralela (PC).

4.2 INSTRUCTION MEMORY (INST. MEM.)

Tal componente é a memória de instruções com uma entrada Address Bus (A) e uma saída Read Data (RD).

Figura 7: Módulo Instr. Memory.



Esse módulo é responsável por fornecer e armazenar as instruções que o processador deve executar. O módulo atua como uma memória somente de leitura (ROM), que armazena o conjunto de instruções de um programa que o processador irá executar.

4.3 CONTROL UNIT

Esse módulo é responsável por gerar sinais de controle para cada uma das instruções suportadas pela CPU. Ele recebe informações da instrução atual e, com base nelas, gera sinais de controle para coordenar as operações do processador.

Entradas:

- **Op:** Campo de 7 bits do opcode da instrução, que indica qual tipo de operação deve ser realizada (como operações aritméticas, de memória, controle de fluxo, etc.).
- **Funct3:** Campo de 3 bits que especifica a função exata dentro de um grupo de operações determinado pelo opcode.
- **Funct7:** Campo de 7 bits que, junto com Op e Funct3, ajuda a determinar a operação específica a ser realizada, especialmente em operações mais complexas como multiplicação ou comparação.

Saídas:

- **Branch:** Indica se a instrução atual é uma instrução de desvio condicional (branch). É ativado para instruções que alteram o fluxo de execução do programa, como beq, bne, etc.
- **ResultSrc:** Seleciona a origem do resultado que será escrito no banco de registradores. Pode escolher entre o resultado da ALU, o valor lido da memória ou um imediato.
- **MemWrite:** Ativa a escrita na memória de dados. Se ativado, a operação atual é uma escrita na memória (sw, sh, sb).
- **ULAControl:** Define a operação que a Unidade Lógica e Aritmética (ULA) deve realizar, como adição, subtração, AND, OR, XOR, etc. Essa saída é derivada dos campos Funct3, Funct7 e Op.
- **ULASrc:** Controla a fonte do segundo operando para a ULA. Pode selecionar entre um registrador ou um valor imediato.

- ImmSrc: Determina o tipo de imediato a ser gerado a partir da instrução. Dependendo do tipo de instrução, o imediato pode ter diferentes formatos e tamanhos (ex.: instruções de carga, armazenagem, saltos, etc.).
- RegWrite: Habilita a escrita no banco de registradores. Se ativado, o resultado de uma operação será escrito em um registrador.
- Jump: Responsável por controlar saltos incondicionais no fluxo de execução do programa, como no caso da instrução JAL (Jump and Link)
- PCSrc: Controla a fonte do valor para o Program Counter (PC), ou seja, ela determina qual valor será usado para atualizar o PC após a execução de uma instrução.

4.4 REGISTER FILE

O módulo Register File gerencia os registradores e apresenta as seguintes entradas e saídas:

Entradas:

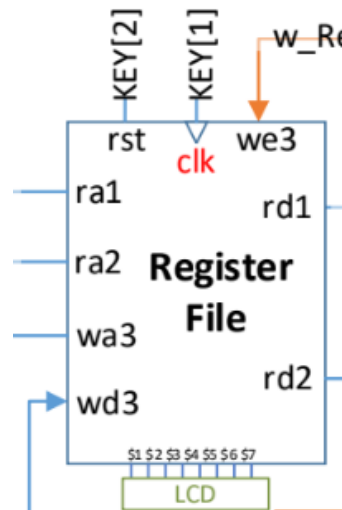
- Write Data (wd3): Entrada de dados;
- Write Address (wa3): Seleção do registrador que armazenará o dado proveniente de Write Data (wd3);
- Write Enable (we3): Habilita (1) ou desabilita (0) a gravação de dados nos registradores de \$1 a \$7;
- Clock (clk) (1 bit): Se o sinal Write Enable (we3) estiver ativo (1), na borda de subida do clock, o dado é gravado no registrador selecionado.
- Register Address 1 (ra1): Seleção de qual registrador será disponibilizado na saída rd1;

- Register Address 2 (ra2): Seleção de qual registrador será disponibilizado na saída rd2;
- Reset (rst): Reseta o valor dos registradores, em nível baixo (0);

Saídas:

- Register Data 1 (rd1) (8 bits) – Barramento de saída de 8 bits;
- Register Data 2 (rd2) (8 bits) – Barramento de saída de 8 bits;

Figura 8: Módulo Register File.



4.5 ULA

O módulo da ULA é responsável por operações aritméticas e lógicas. Apresenta as seguintes entradas e saídas:

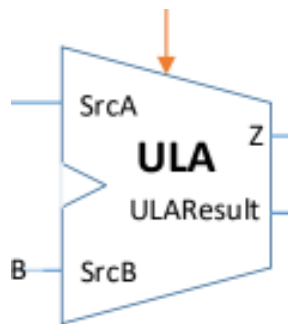
Entradas:

- SrcA (8bits): Entrada de dados do operando A;
- SrcB (8bits): Entrada de dados do operando B;
- ULAControl (3bits): Entrada para seleção da operação realizada.

Saídas:

- ULAResult (8bits): Saída do resultado da operação realizada;
- Flag Z (1bit): Bit de status que indica se a saída da operação realizada é zero (resultado igual a zero: Z=1; resultado diferente de zero: Z=0).

Figura 9: Módulo ULA.



4.6 DATA MEMORY

Esse módulo é usado para armazenar e recuperar dados durante a execução do programa. Apresenta as seguintes entradas e saídas:

Entradas:

- A (Address): É a entrada de endereço.
- WD (Write Data): É a entrada de dados de escrita. Esse sinal carrega o valor que será escrito na memória, caso a operação seja de escrita. O dado será armazenado na posição de memória indicada pelo endereço A.
- WE (Write Enable): Este é um sinal de controle que habilita a operação de escrita na memória. Quando WE está ativo, a memória executa a operação de escrita; caso contrário, apenas operações de leitura são permitidas.
- clk (Clock): O sinal de clock sincroniza as operações de escrita e leitura na memória.

- rst (Reset): Este sinal é usado para reinicializar a memória.

Saídas:

- RD (Read Data): É a saída de dados de leitura.

Figura 10: Módulo Data Memory.



4.7 PARALLELOUT

O módulo Parallel OUT é responsável por transferir dados para fora do sistema de forma controlada e sincronizada, com a capacidade de habilitar ou desabilitar essa operação conforme necessário.

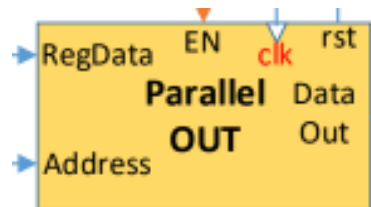
Entradas:

- RegData: Esse barramento de dados carrega as informações que serão enviadas para fora do módulo.
- Address: Indica o endereço ou a posição onde o valor de RegData deve ser enviado.
- EN (Enable): Este sinal habilita a operação de saída.
- clk (Clock): O sinal de clock é usado para sincronizar a operação de saída dos dados.
- rst (Reset): Reinicializa o módulo.

Saída:

- Data Out: É a saída de dados paralelos do módulo.

Figura 11: Módulo Parallel Out.



4.8 PARALLELIN

O módulo Parallel IN é responsável por receber dados de várias fontes e disponibilizá-los ao sistema.

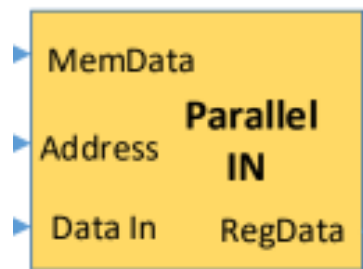
Entradas:

- MemData: Esse barramento carrega dados que vêm de uma memória, geralmente a memória de dados do processador.
- Address: O endereço que especifica a posição da memória ou do registrador de onde o dado deve ser lido ou armazenado. Ele é usado para selecionar a origem ou destino dos dados dentro do sistema.
- Data In: Esse barramento recebe dados que serão enviados para dentro do sistema, vindos de uma fonte externa, como um periférico ou uma interface de comunicação.

Saída:

- RegData: Dados vindos de registradores ou diretamente do processador.

Figura 12: Módulo Parallel In.



5 METODOLOGIA

Para cumprir com os requisitos mínimos, foi implementado no módulo Control Unit mais 3 instruções: bne, jal e ori.

5.1 INSTRUÇÃO BNE

A instrução BNE (Branch if Not Equal) em RISC-V é uma instrução de desvio condicional que faz o processador saltar para um novo endereço se dois registradores não forem iguais. Caso os valores dos dois registradores sejam diferentes, o PC (Program Counter) é atualizado com o endereço de destino do salto, que é calculado somando um deslocamento (offset) ao valor atual do PC.

Opcode: 1100011.

Funct3: 001.

A instrução BNE segue o formato Tipo - B em RISC-V:

bne rs1, rs2, label

rs1: Registro fonte 1.

rs2: Registro fonte 2.

Label: Indica a localização de uma instrução.

No formato Tipo – B, as instruções são divididas em vários campos, como mostrado abaixo:

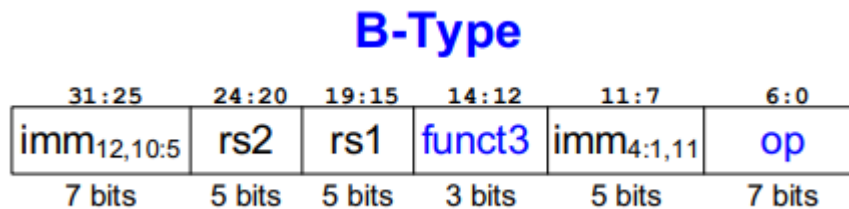


Figura 13: Tipo – B.

5.2 INSTRUÇÃO ORI

A instrução ORI (OR Immediate) em RISC-V é uma instrução do formato Tipo - I, que realiza uma operação lógica OR bit a bit entre o valor de um registrador e um valor imediato. O resultado é armazenado em um registrador de destino.

Opcode: 0010011.

Funct3: 110.

A instrução ORI tem a seguinte forma:

ori rd, rs1, imm

rd: Registrador de destino, onde o resultado será armazenado.

rs1: Registrador fonte 1, cujo valor será usado na operação.

imm: Valor imediato de 12 bits, que será aplicado ao rs1 usando a operação lógica OR.

No formato Tipo – I, as instruções são divididas em vários campos, como mostrado abaixo:

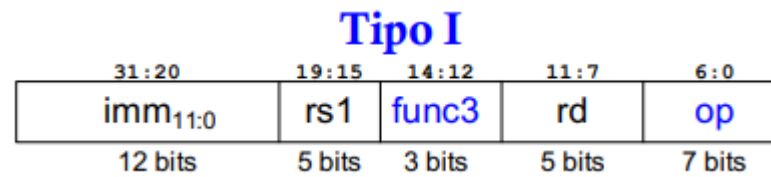


Figura 14: Tipo – I.

5.3 INSTRUÇÃO JAL

A instrução JAL (Jump and Link) em RISC-V é uma instrução usada para realizar um salto incondicional para um endereço especificado. Ao mesmo tempo, ela armazena o endereço de retorno (o endereço da instrução seguinte ao JAL) no registrador de destino. Isso permite que o programa retorne ao ponto de origem após a execução do salto, tornando a instrução útil para chamadas de função e sub-rotinas.

Opcode: 1101111.

A instrução JAL tem a seguinte forma:

jal rd, label

rd: Registrador de destino, onde o resultado será armazenado.

label: Indica a localização de uma instrução.

No formato Tipo – J, as instruções são divididas em vários campos, como mostrado abaixo:

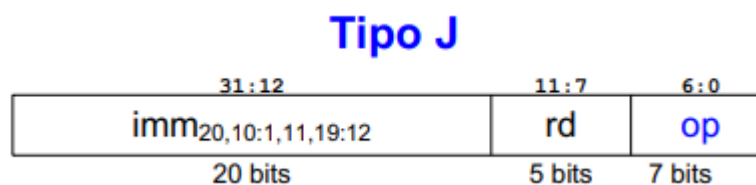
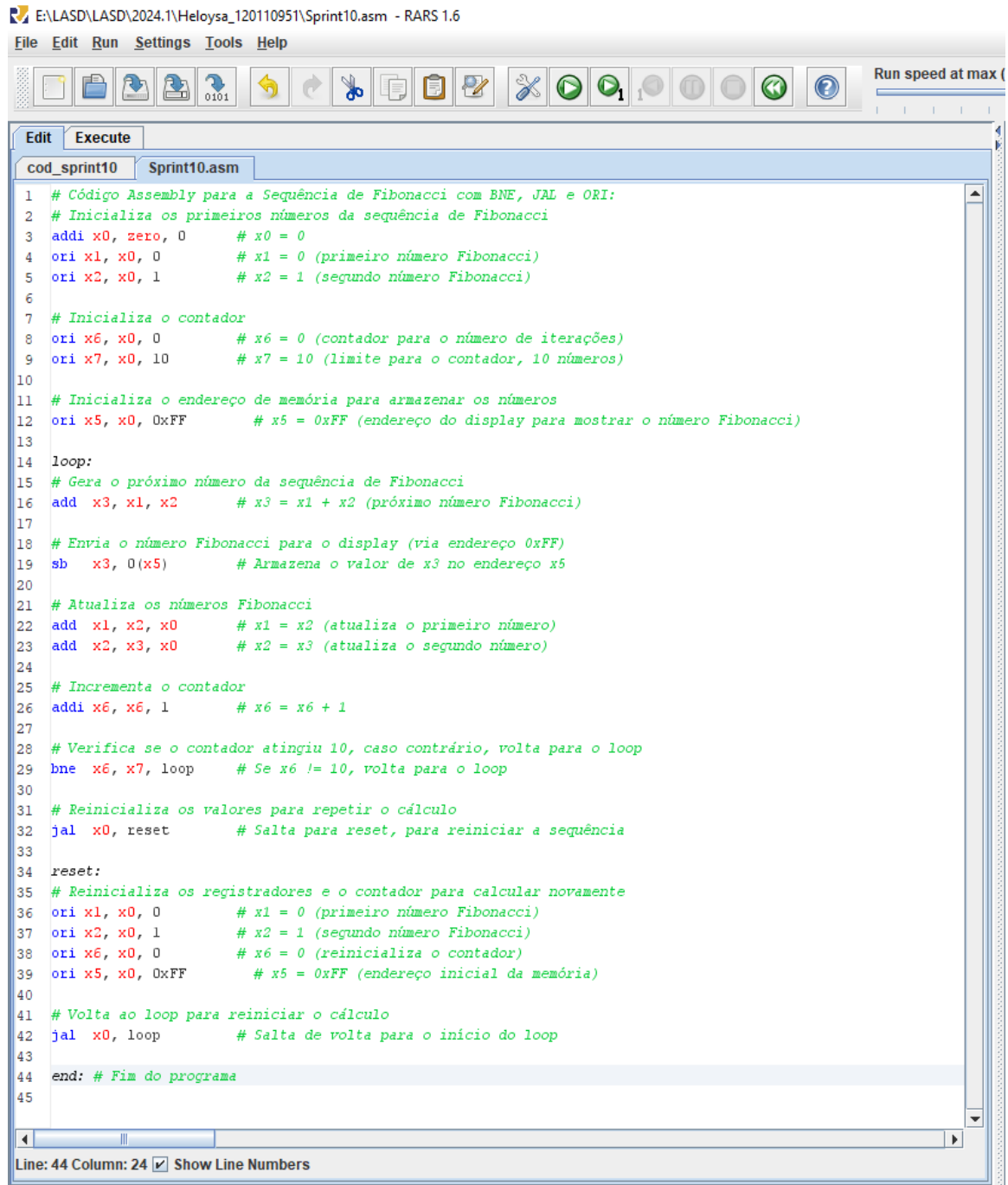


Figura 15: Tipo – J.

5.4 CÓDIGO ASSEMBLY

O código presente no módulo Instruction Memory se encontra abaixo:



```
E:\LASD\LASD\2024.1\Heloysa_120110951\Sprint10.asm - RARS 1.6
File Edit Run Settings Tools Help

1 # Código Assembly para a Sequência de Fibonacci com BNE, JAL e ORI:
2 # Inicializa os primeiros números da sequência de Fibonacci
3 addi x0, zero, 0      # x0 = 0
4 ori x1, x0, 0         # x1 = 0 (primeiro número Fibonacci)
5 ori x2, x0, 1         # x2 = 1 (segundo número Fibonacci)
6
7 # Inicializa o contador
8 ori x6, x0, 0         # x6 = 0 (contador para o número de iterações)
9 ori x7, x0, 10        # x7 = 10 (limite para o contador, 10 números)
10
11 # Inicializa o endereço de memória para armazenar os números
12 ori x5, x0, 0xFF      # x5 = 0xFF (endereço do display para mostrar o número Fibonacci)
13
14 loop:
15 # Gera o próximo número da sequência de Fibonacci
16 add x3, x1, x2        # x3 = x1 + x2 (próximo número Fibonacci)
17
18 # Envia o número Fibonacci para o display (via endereço 0xFF)
19 sb x3, 0(x5)         # Armazena o valor de x3 no endereço x5
20
21 # Atualiza os números Fibonacci
22 add x1, x2, x0        # x1 = x2 (atualiza o primeiro número)
23 add x2, x3, x0        # x2 = x3 (atualiza o segundo número)
24
25 # Incrementa o contador
26 addi x6, x6, 1        # x6 = x6 + 1
27
28 # Verifica se o contador atingiu 10, caso contrário, volta para o loop
29 bne x6, x7, loop      # Se x6 != 10, volta para o loop
30
31 # Reinicializa os valores para repetir o cálculo
32 jal x0, reset         # Salta para reset, para reiniciar a sequência
33
34 reset:
35 # Reinicializa os registradores e o contador para calcular novamente
36 ori x1, x0, 0         # x1 = 0 (primeiro número Fibonacci)
37 ori x2, x0, 1         # x2 = 1 (segundo número Fibonacci)
38 ori x6, x0, 0         # x6 = 0 (reinicializa o contador)
39 ori x5, x0, 0xFF      # x5 = 0xFF (endereço inicial da memória)
40
41 # Volta ao loop para reiniciar o cálculo
42 jal x0, loop          # Salta de volta para o início do loop
43
44 end: # Fim do programa
45
```

Line: 44 Column: 24 ☒ Show Line Numbers

Figura 16: Código em Assembly RISC-V utilizado para calcular a sequência de Fibonacci.

6 DESENVOLVIMENTO DO PROJETO

Para o desenvolvimento do projeto, foi modificado o módulo *ControlUnit*, o módulo *Instr_Memory* e o módulo *Div_Freq*, como é possível visualizar a seguir:

```
// Clock 3 Hz
module Div_Freq (input CLOCK_50MHz, output reg Clk_3Hz);

    reg [25:0] counter = 0;

always @(posedge CLOCK_50MHz) begin
    if (counter == 8333333) begin
        counter <= 0; //Reinicia o contador
        Clk_3Hz <= ~Clk_3Hz; //Inverte o clock de 2Hz
    end else begin //Código executado se a condição for falsa
        counter <= counter + 1; //Incrementa o contador
    end
end
endmodule
```

Figura 17: Código do Módulo Div_Freq.

```
module Instr_Memory (
    input [7:0] A, // Endereço de 8 bits
    output reg [31:0] RD // Saída de dados de 32 bits
);

always @(*) begin
    case (A)

        8'h00: RD = 32'h00000013;
        8'h04: RD = 32'h00006093;
        8'h08: RD = 32'h00106113;
        8'h0C: RD = 32'h00006313;
        8'h10: RD = 32'h00a06393;
        8'h14: RD = 32'h0ff06293;
        8'h18: RD = 32'h002081b3;
        8'h1C: RD = 32'h00328023;
        8'h20: RD = 32'h000100b3;
        8'h24: RD = 32'h00018133;
        8'h28: RD = 32'h00130313;
        8'h2C: RD = 32'hfe7316e3;
        8'h30: RD = 32'h0040006f;
        8'h34: RD = 32'h00006093;
        8'h38: RD = 32'h00106113;
        8'h3C: RD = 32'h00006313;
        8'h40: RD = 32'h0ff06293;
        8'h44: RD = 32'hfd5ff06f;

        default: RD = 32'h00000000; // NOP (No Operation) por padrão
    endcase
end
endmodule
```

Figura 18: Código do Módulo Inst_Memory.

```
module ControlUnit (
    //Entradas
    input [6:0] Op,           // Opcode (bits 6:0)
    input [2:0] Funct3,       // Funct3 (bits 14:12)
    input [6:0] Funct7,       // Funct7 (bits 31:25)

    // Sairas
    output reg RegWrite,      // Sinal de escrita no registrador
    output reg ULASrc,        // Sinal para o MUX que seleciona o segundo operando da ULA
    output reg [2:0] ULACtrl, // Sinal de controle da ULA
    output reg [1:0] ImmSrc,   // Seleciona o tipo de imediato
    output reg MemWrite,       // Habilita a escrita na memória de dados
    output reg [1:0] ResultSrc, // Seleciona a origem do resultado (da ULA ou memória)
    output reg Branch,         // Sinal de branch para instruções de salto condicional
    output reg PCSrc,
    output reg Jump
);

    reg [16:0] IN_unControl;

    always @(*) begin

        IN_unControl = {Op, Funct3, Funct7};

        casex (IN_unControl)

            // ADD - Tipo R
            17'b0110011_000_0000000: begin
                RegWrite = 1'b1; ImmSrc = 2'bxxx; ULASrc = 1'b0;
                ULACtrl = 3'b000; MemWrite = 1'b0; ResultSrc = 2'b00;
                Branch = 1'b0; Jump = 1'b0; PCSrc = 1'b0;
            end

            // SUB - Tipo R
            17'b0110011_000_0100000: begin
                RegWrite = 1'b1; ImmSrc = 2'bxxx; ULASrc = 1'b0;
                ULACtrl = 3'b001; MemWrite = 1'b0; ResultSrc = 2'b00;
                Branch = 1'b0; Jump = 1'b0; PCSrc = 1'b0;
            end

            // AND - Tipo R
            17'b0110011_1111_0000000: begin
                RegWrite = 1'b1; ImmSrc = 2'bxxx; ULASrc = 1'b0;
                ULACtrl = 3'b010; MemWrite = 1'b0; ResultSrc = 2'b00;
                Branch = 1'b0; Jump = 1'b0; PCSrc = 1'b0;
            end

        endcase
    end
end
```

```
// OR - Tipo R
17'b011001111100000000: begin
RegWrite = 1'b1; ImmSrc = 2'bxx; ULASrc = 1'b0;
ULAControl = 3'b011; MemWrite = 1'b0; ResultSrc = 2'b00;
Branch = 1'b0; Jump = 1'b0; PCSrc = 1'b0;
end

// XOR - Tipo R
17'b0110011_100_0000000: begin
RegWrite = 1'b1; ImmSrc = 2'bxx; ULASrc = 1'b0;
ULAControl = 3'b100; MemWrite = 1'b0; ResultSrc = 2'b00;
Branch = 1'b0; Jump = 1'b0; PCSrc = 1'b0;
end

// SLT - Tipo R
17'b0110011_010_0000000: begin
RegWrite = 1'b1; ImmSrc = 2'bxx; ULASrc = 1'b0;
ULAControl = 3'b101; MemWrite = 1'b0; ResultSrc = 2'b00;
Branch = 1'b0; Jump = 1'b0; PCSrc = 1'b0;
end

// ADDI - Tipo I
17'b0010011_000_0000000: begin
RegWrite = 1'b1; ImmSrc = 2'b00; ULASrc = 1'b1;
ULAControl = 3'b000; MemWrite = 1'b0; ResultSrc = 2'b00;
Branch = 1'b0; Jump = 1'b0; PCSrc = 1'b0;
end

// LB - Tipo S
17'b0000011_000_0000000: begin
RegWrite = 1'b1; ImmSrc = 2'b00; ULASrc = 1'b1;
ULAControl = 3'b000; MemWrite = 1'b0; ResultSrc = 2'b01;
Branch = 1'b0; Jump = 1'b0; PCSrc = 1'b0;
end

// SB - Tipo S
17'b0100011_000_0000000: begin
RegWrite = 1'b0; ImmSrc = 2'b01; ULASrc = 1'b1;
ULAControl = 3'b000; MemWrite = 1'b1; ResultSrc = 2'bxx;
Branch = 1'b0; Jump = 1'b0; PCSrc = 1'b0;
end

//BEQ - Tipo B
17'b1100011_000_0000000: begin
RegWrite = 1'b0; ImmSrc = 2'b10; ULASrc = 1'b0;
ULAControl = 3'b001; MemWrite = 1'b0; ResultSrc = 2'bxx;
Branch = 1'b1; Jump = 1'b0; PCSrc = 1'b1;
end
```

```
//BNE - Tipo B
17'b1100011_001_xxxxxxx: begin
RegWrite = 1'b0; ImmSrc = 2'b10; ULASrc = 1'b0;
ULAControl = 3'b001; MemWrite = 1'b0; ResultSrc = 2'bx;
Branch = 1'b1; Jump = 1'b0; PCSrc = 1'b1;
end

//JAL - Tipo J
17'b1101111_xxx_xxxxxxx: begin
RegWrite = 1'b1; ImmSrc = 2'b11; ULASrc = 1'bx;
ULAControl = 3'bxxx; MemWrite = 1'b0; ResultSrc = 2'b10;
Branch = 1'b0; Jump = 1'b1; PCSrc = 1'b1;
end

// ORI - Tipo I
17'b0010011_110_xxxxxxx: begin
RegWrite = 1'b1; ImmSrc = 2'b00; ULASrc = 1'b1;
ULAControl = 3'b011; MemWrite = 1'b0; ResultSrc = 2'b00;
Branch = 1'b0; Jump = 1'b0; PCSrc = 1'b0;
end
```

Figura 19: Código do Módulo ControlUnit.

6.1 CÓDIGO MOD_TESTE

As instanciações dos módulos criados ficaram da seguinte maneira no Mod_Teste:

```
wire w_PCSrc;
wire [1:0] w_ResultSrc;
wire w_MemWrite;
wire [2:0] w_ULAControl;
wire w_ULASrc;
wire [1:0] w_ImmSrc;
wire w_RegWrite;
wire w_Zero;

wire w_Branch;
wire w_Jump;
wire w_Funct3;

wire [7:0] w_PCp4;
wire [7:0] w_PC;
wire [7:0] w_rdlSrcA;
wire [7:0] w_rd2;
wire [7:0] w_SrcB;
wire [7:0] w_ULAResult;
wire [31:0] w_Inst;

wire [7:0] w_Wd3;
wire [7:0] w_Imm;
wire [7:0] w_RData;

wire [7:0] w_ImmPC;
wire [7:0] w_PCn;

wire [7:0] w_DataOut;
wire [7:0] w_Detain;
wire [7:0] w_RegData;

assign w_PCp4 = w_PC + 3'h4;

assign w_ImmPC = w_Imm + w_PC;
```

Figura 20: Mod_Teste – Parte 1.

```
// Divisor de frequência
//wire clk_3Hz = KEY[1];
wire clk_3Hz;
Div_Freq div_freq (.CLOCK_50MHz(CLOCK_50), .Clk_3Hz(clk_3Hz));

// Instanciação do Program Counter (PC)
ProgramCounter pc (.clk(clk_3Hz), .rst(KEY[2]), .PCin(w_PCn), .PC(w_PC));

// Instanciação da Memória de Instruções
Instr_Memory memory (.A(w_PC), .RD(w_Inst));

// Instanciação da Unidade de Controle
ControlUnit control_inst (
    .Op(w_Inst[6:0]),
    .Funct3(w_Inst[14:12]),
    .Funct7(w_Inst[31:25]),
    .Branch(w_Branch),
    .ResultSrc(w_ResultSrc),
    .MemWrite(w_MemWrite),
    .ULAControl(w_ULAControl[2:0]),
    .ULASrc(w_ULASrc),
    .ImmSrc(w_ImmSrc),
    .RegWrite(w_RegWrite),
    .PCSrc(w_PCSrc),
    .Jump(w_Jump)
);
```

Figura 21: Mod_Teste – Parte 2.

```
// Instanciação do Banco de Registradores - New
RegisterFile_NEW registradores (
    .clk(clk_3Hz),
    .rst(KEY[2]),
    .ra1(w_Inst[19:15]),
    .ra2(w_Inst[24:20]),
    .wa3(w_Inst[11:7]),
    .wd3(w_Wd3),
    .we3(w_RegWrite),
    .rd1(w_rdlSrcA),
    .rd2(w_rdl2),
    // Saídas auxiliares para depuração, mostrando os valores dos registradores
    .x0(),
    .x1(),
    .x2(),
    .x3(),
    .x4(),
    .x5(),
    .x6(),
    .x7()
);
```

Figura 22: Mod_Teste – Parte 3.

```
// Instanciação da ULA
ULA ula (
    .SrcA(w_rdlSrcA),
    .SrcB(w_SrcB),
    .ULAControl(w_ULAControl[2:0]),
    .ULAResult(w_ULAResult),
    .Z(w_Zero)
);
```

Figura 23: Mod_Teste – Parte 4.


```
// Instanciação do ParallelOUT
ParallelOUT ParaOUT (
    .Address(w_ULAResult),
    .RegData(w_rd2),
    .EN(w_MemWrite),
    .clk(clk_3Hz),
    .rst(KEY[2]),
    .DataOut(w_DataOut)
);

// Instanciação do ParallelIN
ParallelIN ParaIN (
    .Address(w_ULAResult),
    .MemData(w_RData),
    .DataIn(w_Detain),
    .RegData(w_RegData)
);

//Instanciação da Data_Mem
Data_Mem data_Mem (
    .A(w_ULAResult),
    .WD(w_rd2),
    .rst(KEY[2]),
    .clk(clk_3Hz),
    .WE(w_MemWrite),
    .RD(w_RData)
);

// Instanciação do MUX 4x1 de 8 bits - MuxImmSrc
Mux_4x1 MuxImmSrc (
    .in0(w_Inst[31:20]),
    .in1({w_Inst[31:25], w_Inst[11:7]}),
    .in2({w_Inst[7], w_Inst[30:25], w_Inst[11:8], 1'b0}),
    .in3(),
    .sel(w_ImmSrc),
    .out(w_Imm)
);
```

Figura 24: Mod_Teste – Parte 5.

```
// Instanciação do MUX 2x1 de 8 bits - MuxULASrc
mux2x1 MuxULASrc (.in0(w_rd2), .in1(w_Imm), .sel(w_ULASrc), .out(w_SrcB));

// Instanciação do MUX 4x1 de 8 bits - MuxResSrc
Mux_4x1 MuxResSrc (
    .in0(w_ULAResult),
    .in1(w_RegData),
    .in2(w_PCp4),
    .in3(),
    .sel(w_ResultSrc),
    .out(w_Wd3)
);

// Instanciação do MUX 2x1 de 8 bits - MuxPCSrc
mux2x1 MuxPCSrc (.in0(w_PCp4), .in1(w_ImmPC), .sel(w_PCSrc), .out(w_PCn));
```

Figura 25: Mod_Teste – Parte 6.

```
assign w_DataOut = w_dlx4;

assign w_Detain = SW[7:0];

// Conexão dos sinais de controle aos LEDs de depuração
//assign LEDR[0] = w_Branch;
//assign LEDR[1] = w_ResultSrc;
//assign LEDR[2] = w_MemWrite;
//assign LEDR[5:3] = w_ULAControl;
//assign LEDR[6] = w_ULASrc;
//assign LEDR[8:7] = w_ImmSrc;
//assign LEDR[9] = w_RegWrite;
//assign LEDR[17] = w_Zero;

// Instanciação do módulo hex7seg
hex7seg HX0 (.hex(w_Inst[3:0]), .seg(HEX0[0:6]));
hex7seg HX1 (.hex(w_Inst[7:4]), .seg(HEX1[0:6]));
hex7seg HX2 (.hex(w_Inst[11:8]), .seg(HEX2[0:6]));
hex7seg HX3 (.hex(w_Inst[15:12]), .seg(HEX3[0:6]));
hex7seg HX4 (.hex(w_Inst[19:16]), .seg(HEX4[0:6]));
hex7seg HX5 (.hex(w_Inst[23:20]), .seg(HEX5[0:6]));
hex7seg HX6 (.hex(w_Inst[27:24]), .seg(HEX6[0:6]));
hex7seg HX7 (.hex(w_Inst[31:28]), .seg(HEX7[0:6]));

assign LEDG[8] = clk_3Hz;

assign LEDG[5] = ~KEY[2];

endmodule
```

Figura 26: Mod_Teste – Parte 7.

7 CONCLUSÃO

O desenvolvimento deste projeto teve como objetivo a implementação de um processador RISC-V de ciclo único capaz de calcular a sequência de Fibonacci, além de incorporar novas instruções e melhorias ao circuito do processador. Ao longo do projeto, foi possível consolidar diversos conceitos de arquitetura de processadores, incluindo controle de fluxo, operações lógicas e o uso eficiente dos recursos de hardware.

No entanto, apesar dos avanços alcançados, o projeto apresentou dificuldades que impediram seu funcionamento pleno. Infelizmente, o código não executou completamente da maneira esperada. Uma das instruções adicionadas, que deveria desempenhar um papel fundamental no cálculo e no controle de fluxo do processador, não funcionou corretamente. Várias tentativas de diagnóstico e correção do erro foram realizadas, inclusive com o apoio das aulas de monitoria, mas o problema persiste, e a origem exata do erro não pôde ser identificada a tempo para a entrega desse relatório.

Apesar dessas dificuldades, o projeto serviu como uma excelente oportunidade de aprendizado. Ele proporcionou um ambiente prático para testar e aplicar os conhecimentos adquiridos sobre a arquitetura RISC-V e os desafios envolvidos no desenvolvimento de sistemas digitais. As lições aprendidas com os problemas encontrados serão valiosas em projetos futuros, tanto para melhorar a capacidade de análise de erros quanto para aprimorar o desenvolvimento de hardware.

8 REFERÊNCIAS

ASTH, R. C. Sequência de Fibonacci. Disponível em: <<https://www.todamateria.com.br/sequencia-de-fibonacci/>>. Acesso em: 9 out. 2024.

ÁVILA, L. Sequência de Fibonacci. Conhecendo a Sequência de Fibonacci. Disponível em: <<https://mundoeducacao.uol.com.br/matematica/sequencia-fibonacci.htm>>. Acesso em: 9 out. 2024.

CAETANO, E. Proporção áurea e sua influência nas construções. Disponível em: <<https://gerenciadeobras.com.br/proporcao-aurea-nas-construcoes/>>. Acesso em: 11 out. 2024.

HARRIS, S. L.; HARRIS, D. Digital Design and Computer Architecture, RISC-V Edition. 1. ed. [s.l.] Morgan Kaufmann Publishers, 2021. p. 592

MARIA BEATRIZ. Sequência de Fibonacci. Disponível em: <<https://brasilescola.uol.com.br/matematica/sequencia-fibonacci.htm>>. Acesso em: 8 out. 2024.

O Que é a Sequência de Fibonacci? Para Que Serve? Disponível em: <<https://www.brasilparalelo.com.br/artigos/o-que-e-fibonacci>>. Acesso em: 10 out. 2024.