PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE RORAIMA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR INSALUBYTE

ALUNOS:

Ryan Kayky Marques Rolins Bastos – 202207080
Giovana Oliveira Moraes de Lima – 2022006980
Victor Hugo Costa - 2022010016

Novembro de 2023
Boa Vista/Roraima

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE RORAIMA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR INSALUBYTE

Novembro de 2023
Boa Vista/Roraima

Resumo

Este trabalho aborda o projeto de implementação de um processador 16 bits uni ciclo baseado na arquitetura do processador MIPS, implementado através da linguagem VHDL e a ferramenta Intel Quartus Prime Lite. Ao decorrer do relatório será apresentado a descrição de cada componente necessário para o funcionamento do processador, bem como alguns testes realizados ao decorrer do desenvolvimento.

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Especificação

Nesta seção é apresentado o conjunto de itens para o desenvolvimento do processador INSALUBYTE, bem como a descrição de cada etapa da construção do processador.

1.1 Plataforma de desenvolvimento

Para a implementação do processador INSALUBYTE foi utilizado a IDE Quartus Prime Lite Edition, versão 19.1.0 e simulador ModelSim na versão 19.1.0.670. É uma ferramenta que fornece os recursos necessários para a realização do projeto.

A grande vantagem de usar a IDE é pôr em prática a programação da linguagem VHDL, adquirir experiências na manipulação de hardware e componentes funcionais de um processador.

Flow Status	Successful - Wed Nov 29 14:22:14 2023
Quartus Prime Version	19.1.0 Build 670 09/22/2019 SJ Lite Edition
Revision Name	Processador
Top-level Entity Name	Processador
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	83 / 56,480 (< 1 %)
Total registers	64
Total pins	145 / 268 (54 %)
Total virtual pins	0
Total block memory bits	256 / 7,024,640 (< 1 %)
Total DSP Blocks	0 / 156 (0 %)
Total HSSI RX PCSs	0 / 6 (0 %)
Total HSSI PMA RX Deserializers	0 / 6 (0 %)
Total HSSI TX PCSs	0 / 6 (0 %)
Total HSSI PMA TX Serializers	0 / 6 (0 %)
Total PLLs	0 / 13 (0 %)
Total DLLs	0 / 4 (0 %)

Figura 1 - Especificações no Quartus

1.2 Conjunto de instruções

O processador INSALUBYTE possui 8 registradores: S0, S1, S2, S3, S4, S5, S6, S7. Assim como 3 formatos de instruções de 16 bits cada, Instruções do **tipo R** (operações aritméticas e saltos condicionais) **, I** (operações com valores na memória e valores imediatos) e **J** (saltos incondicionais), seguem algumas considerações sobre as estruturas contidas nas instruções:

- **Opcode**: a operação básica a ser executada pelo processador, tradicionalmente chamado de código de operação;
- **Reg1**: É o registrador de destino para alguns tipos de instruções (ex. instruções do tipo R) e o primeiro operando fonte;
- **Reg2**: o registrador contendo o segundo operando fonte;
- **Reg3**: o registrador contendo o terceiro operando fonte;

Tipo de Instruções:

- Formato do tipo R: Este formato aborda instruções de Load (exceto *load Immediately*), Store e instruções baseadas em operações aritméticas.

Formato para escrita em código binário do tipo R e I:

Tabela 1 – Escrita em binário do tipo R e I

Opcode	Reg1	Reg2	Reg3
4 bits	4 bits	4 bits	4 bits
15-12	11-8	7-4	3-0

Formato para escrita em código binário do tipo J:

Tabela 2 – Escrita em binário do tipo J

Opcode	Label
4 bits	12 bits
15-12	11-0

Visão geral das instruções do Processador INSALUBYTE:

O número de bits do campo **Opcode** das instruções é igual a quatro, sendo assim obtemos um total ($Bit(0e1)^{NumeroTodaldeBitsdoOpcode} \therefore 2^4 = 16$) de 16 **Opcodes (0-12)** que são distribuídos entre as instruções, assim como é apresentado na Tabela 1.

Tabela 3 – Lista de Opcodes utilizadas pelo processador INSALUBYTE

Opcode	Nome	Formato	Breve Descrição	Exemplo
0000	ADD	R	Soma	**add** $S0, $S1, $S2, ou seja, $S0 := $S1+$S2
0001	ADDi	I	Soma Imediata	**addi** $S0, $S1, X, ou seja, $S0 := $S1+X
0010	SUB	R	Subtração	**sub** $S0, $S1, $S2, ou seja, $S0 := $S1 - $S2
0011	SUBi	I	Subtração Imediata	**subi** $S0, $S1, X, ou seja, $S0 := $S1-X
0100	LW	I	Load Word	**lw** $S0, 0($0)
0101	SW	I	Store Word	**sw** $S0, 0($0)
0110	LI	I	Load Imediato	**li** $S0, 31
0111	BEQ	R	Salto Condicional	**beq** $S0, $S1, L1
1000	IF	R	Condição	**If** $S0, $S1
1001	J	J	Salto	**J** L1
1010	MULT	R	Multiplicação	**mult** $S0, $S1, $S2, ou seja, $S0 := $S1*$S2
1011	MULTi	I	Multiplicação imetiada	**multi** $S0, $S1, X, ou seja, $S0 := $S1*X

1.3 Descrição do Hardware

Nesta seção são descritos os componentes do hardware que compõem o processador INSALUBYTE, incluindo uma descrição de suas funcionalidades, valores de entrada e saída.

1.3.1 ALU ou ULA

O componente compULA (Unidade Lógica Aritmética) tem como principal objetivo efetuar as principais operações aritméticas, dentre elas: soma, subtração e multiplicação. Adicionalmente o compULA efetua operações de comparação de valor como maior ou igual, menor ou igual, somente maior, menor ou igual. O elemento recebe como entrada três valores:

- **rs** – Dado de 16 bits para operação;
- **rd** - Dado de 16 bits para operação;
- **AluOp** – Identificador da operação que será realizada, com 4 bits.

O membro também possui duas saídas:

- **z** – Identificador de resultado (1bit) para comparações (1 se verdade e 0 caso contrário);
- **resultado** – Saída com o resultado das operações aritméticas.

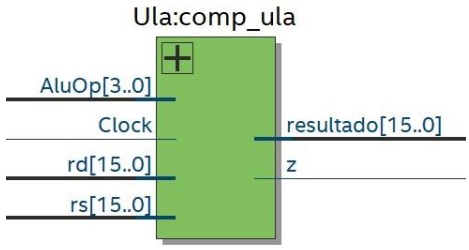

Figura 2 - Bloco simbólico do componente compULA gerado pelo Quartus

1.3.2 Somador

O somador_pc desempenha a função de somar o valor da instrução atual do PC com o valor 1 em binário. Ele possui duas entradas, o "Clock" e a "Entrada", onde recebe o endereço de 16 bits da instrução atual do PC. Além disso, apresenta uma saída denominada "Saida", que receberá o resultado da operação de soma do endereço atual com o valor 1, resultando no próximo endereço de instrução a ser armazenado no PC.

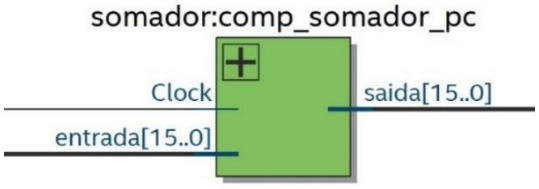

Figura 3 - Bloco simbólico do componente somador gerado pelo Quartus

1.3.3 PC

A função do componente PC é armazenar o endereço de 16 bits da instrução que será executada. Ele possui dois valores de entrada: o Clock, que é responsável por ativar o componente, e o enderecoDeEntrada, onde entra o endereço da instrução a ser executada. Além disso, possui uma saída enderecoDeEntrada, de onde sai o endereço da instrução a ser executada.

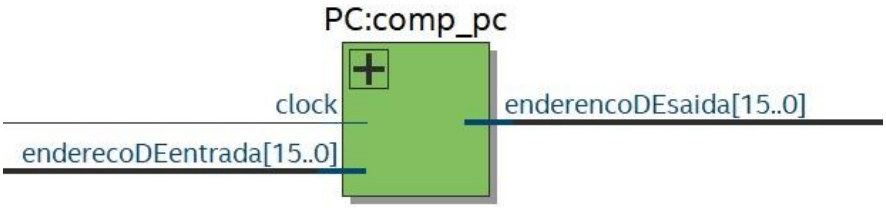

Figura 4 - Bloco simbólico do componente PC gerado pelo Quartus

1.3.4 Unidade de Controle (UC)

A Unidade de Controle (UC) tem como propósito gerenciar todos os componentes do processador com base no opcode de 4 bits recebido no OpCode. Esse controle é realizado por meio das flags de saída a seguir:

- **jump**: sinal que determina a adoção de um desvio incondicional.
- **branch**: sinal que determina se vai ocorrer um desvio condicional.
- **MemRead**: sinal que determina se será lido um dado da memória RAM.
- **MemToReg**: sinal que determina se o dado será escrito no banco de registradores será enviado pela ULA ou pela memória RAM.
- **MemWrite**: sinal para decidir se será escrito um dado da memória RAM.
- **RegWrite**: sinal para determinar se o banco de registradores vai escrever um dado na posição do registrador de destino.
- **AluSrc**: sinal para decidir se o dado que entrará na ULA será enviado pelo banco de registradores ou pelo extensor de sinal.

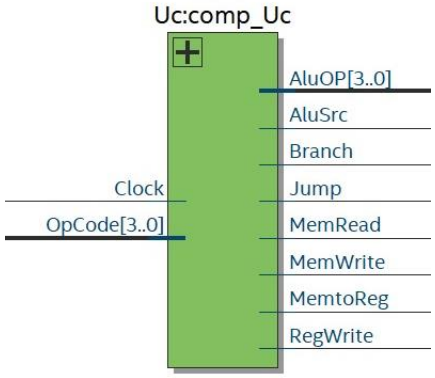

Figura 5 - Bloco simbólico do componente UC gerado pelo Quartus

Abaixo segue a tabela, onde é feita a associação entre os OpCodes e as flags de controle:

Tabela 4 - Detalhes das flags de controle do processador

Comando	AluOp	Regwrite	Jump	Branch	MemRead	MemToRead	MemWrite	Alusrc
Add	0000	1	0	0	0	0	0	0
addi	0001	1	0	0	0	0	0	1
sub	0010	1	0	0	0	0	0	0
subi	0011	1	0	0	0	0	0	1
Lw	0100	1	0	0	1	1	0	0
Sw	0101	0	0	0	0	0	1	0
Li	0110	1	0	0	0	0	0	1
Beq	0111	0	0	1	0	0	0	0
IF_Op	1000	0	0	0	0	0	0	0
Mult	1010	1	0	0	0	0	0	0
Multi	1011	1	0	0	0	0	0	1
J	1001	0	1	0	0	0	0	0

1.3.5 Memória de dados (RAM)

A memória RAM desempenha a função de armazenar temporariamente os dados utilizados durante a execução das instruções. Ela possui cinco entradas:

- Clock: ativa o componente;
- DATA_IN: recebe o dado de 16 bits que será temporariamente armazenado na RAM;
- WRITE_MEMORY: recebe 1 bit para determinar se haverá armazenamento de dados na RAM;
- READ_MEMORY: recebe 1 bit para determinar se haverá leitura de dados na RAM;
- Endereco: entrada de 16 bits que representa a posição na memória RAM onde o dado será escrito ou lido.

A memória RAM também possui uma saída denominada `DATA_OUT`, da qual é emitido um dado de 16 bits da posição indicada pelo endereço, somente se a entrada `READ_MEMORY` estiver recebendo '1'.

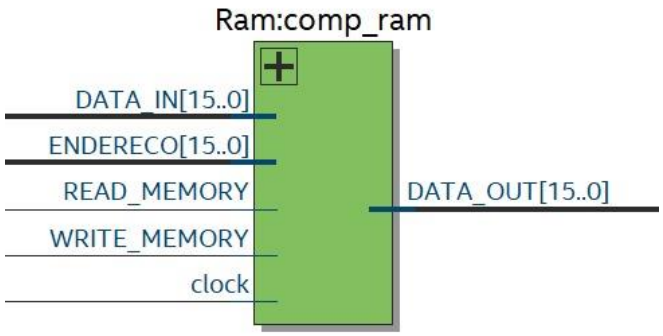

Figura 6 - Bloco simbólico do componente RAM gerado pelo Quartus

1.3.6 Memória de Instruções (ROM)

A Memória de Instrução desempenha a função de armazenar as instruções que serão executadas pelo processador. Esta memória possui duas entradas:

- Clock: responsável por ativar o componente;
- Endereco: endereço de 16 bits que indica a localização da instrução que será executada.

Além disso, a Memória de Instrução possui uma saída de 16 bits, pela qual é emitida a instrução correspondente ao endereço fornecido.

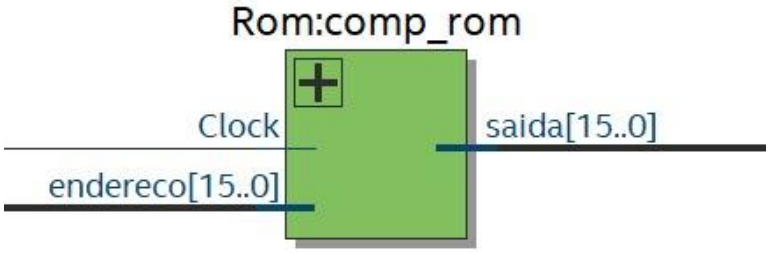

Figura 7 - Bloco simbólico do componente ROM gerado pelo Quartus

1.3.7 Banco de registradores

O banco de registradores é encarregado de armazenar os dados utilizados na execução das operações que o empregam. Este banco possui 8 registradores capazes de armazenar valores de 16 bits. Suas entradas incluem o clock, responsável por ativar o componente, reg_write, que habilita a opção de escrever dados no registrador, e write_data, que recebe o valor de 16 bits a ser gravado no primeiro registrador. Adicionalmente, Reg1 recebe 4 bits para indicar o endereço do primeiro registrador, enquanto Reg2 e reg3 recebem, cada um, 4 bits para indicar os endereços do segundo e terceiro registradores, respectivamente.

Os componentes RegA_de_saida e RegB_de_saida produzem uma saída de 16 bits, representando o valor armazenado nos endereços de Reg2 e Reg3, respectivamente.

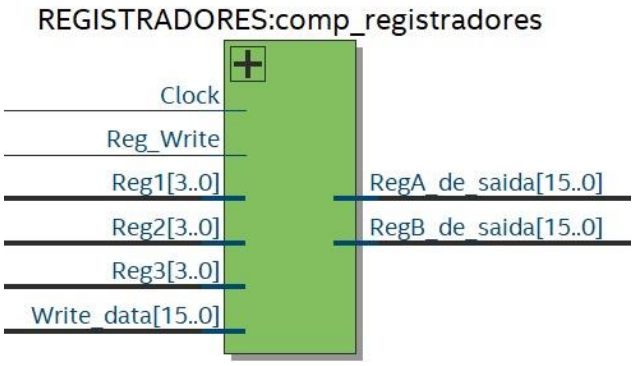

Figura 8 - Bloco simbólico do componente Registradores gerado pelo Quartus

1.3.8 Divisor de instruções

O divisor é utilizado uma única vez ao longo do processador e tem a função de separar a instrução para os componentes subsequentes. A instrução está no formato [OPCODE, R1, R2, R3] quando lidamos com uma instrução do tipo R. Suas entradas compreendem a instrução, enquanto suas saídas incluem opcode, rs, rt, rd e endereço.

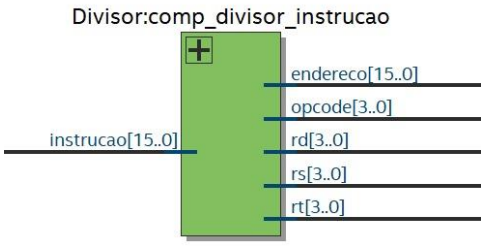

Figura 9 - Bloco simbólico do componente Divisor de Instrução gerado pelo Quartus

1.3.9 Multiplexador

Um multiplexador 2 para 1 possui duas entradas de 16 bits, denominadas a e b, que recebem os dados, e uma entrada s, responsável por determinar qual dado será selecionado como saída. Quando s é igual a 0, o dado de saída será idêntico ao de a; caso s seja 1, o dado de saída será o correspondente a b

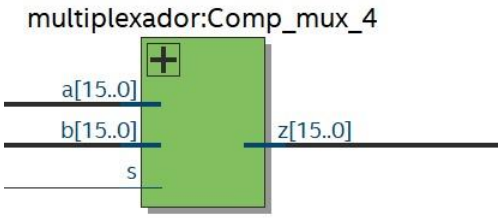

Figura 10 - Bloco simbólico do componente MUX gerado pelo Quartus

1.3.10 Extensor de sinal 4x16

O extensor de sinal desempenha a função de ampliar o número de bits da entrada, que é de 4 bits, para 16 bits na saída. Seu resultado é empregado nas operações na ULA e também para fornecer o endereço à memória de dados.

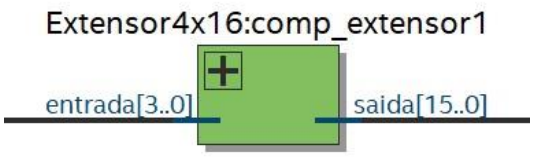

Figura 11 - Bloco simbólico do componente Extensor gerado pelo Quartus

1.3.11 Branch Helper

O Branch helper é responsável por realizar a comparação nos jumps condicionais, como Beq e If. Ele possui uma entrada A de 1 bit e uma saída S de 1 bit. Seu resultado é utilizado quando a condição Beq é verdadeira

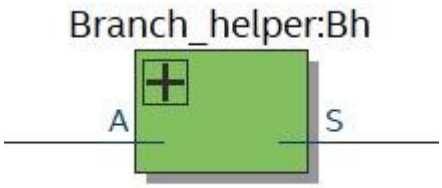

Figura 12 - Bloco simbólico do componente Branch gerado pelo Quartus

1.4 Datapath

É a conexão entre as unidades funcionais formando um único caminho de dados e acrescentando uma unidade de controle responsável pelo gerenciamento das ações que serão realizadas para diferentes classes de instruções.

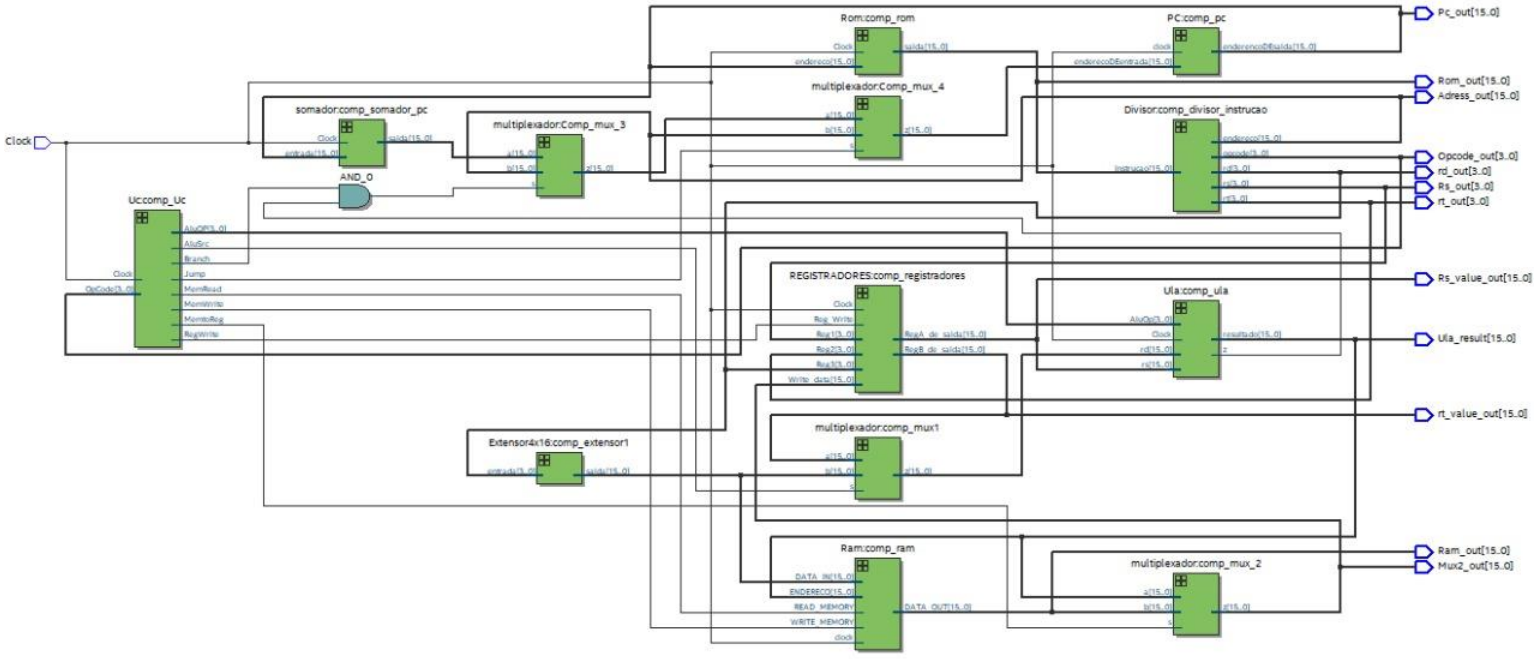

Figura 13 - Bloco simbólico da CPU gerado pelo Quartus

2 Simulações e Testes

Objetivando analisar e verificar o funcionamento do processador, efetuamos alguns testes analisando cada componente do processador em específico, em seguida efetuamos testes de cada instrução que o processador implementa. Para demonstrar o funcionamento do processador INSALUBYTE utilizaremos como exemplo o código para calcular o número da sequência de Fibonacci.

Tabela 5 - Código Fibonacci para o processador Quantum/EXEMPLO.

Endereço	Linguagem de Alto Nível	Binário			
		Opcode	Reg1	Reg2	Reg3
			Endereço		
		Dado			
0	**Addi** $S0, $S0, 0	0001	0000	0000	0000
		0001000000000000			
1	**Sw** $S0 0	0101	0000	0000	0000
		0101000000000000			
2	**Addi** $S0, $S0, 1	0001	0000	0000	0001
		0001000000000001			
3	**Addi** $S1, $S1, 1	0001	0010	0001	0001
		0001000100010001			
4	**Lw** $S2, 0	0100	0010	0010	0000
		0100001000100000			
5	**Add** $S2, $S2, $S1	0000	0010	0010	0001
		0000001000100001			
6	**Add** $S1, $S1, $S0	0000	0001	0001	0000
		0000000100010000			
7	**Lw** $S0, 0	0100	0000	0000	0000
		0100000000000000			
8	**Add** $S0, $S0, $S2	0000	0000	0000	0010
		0000000000000010			
9	**J** 0100	1001	0000	0000	0100
		1001000000000100			

Explicando o funcionamento do código, primeiramente ocorre uma adição imediata em $S0, tornando-o igual a 0. Em seguida, temos a instrução de armazenamento (store - sw) que armazena o valor de $S0 na posição 0 da RAM. No início da sequência de Fibonacci, adicionamos 1 a $S0 e 1 a $S1. Em seguida, carregamos o valor armazenado na RAM da posição 0 em $S2.

As etapas subsequentes envolvem operações aritméticas: somamos $S2 a $S1, somamos $S1 a $S0 e carregamos o valor de $S0 na RAM (Posição 0). Adicionalmente, somamos $S0 a $S2 e, por fim, saltamos para a linha 3, mantendo um ciclo até atingir o limite do processador. Este ciclo representa a iteração do código na execução da sequência de Fibonacci

Verificação dos resultados no relatório da simulação: Após a compilação e execução da simulação, o seguinte relatório é exibido.

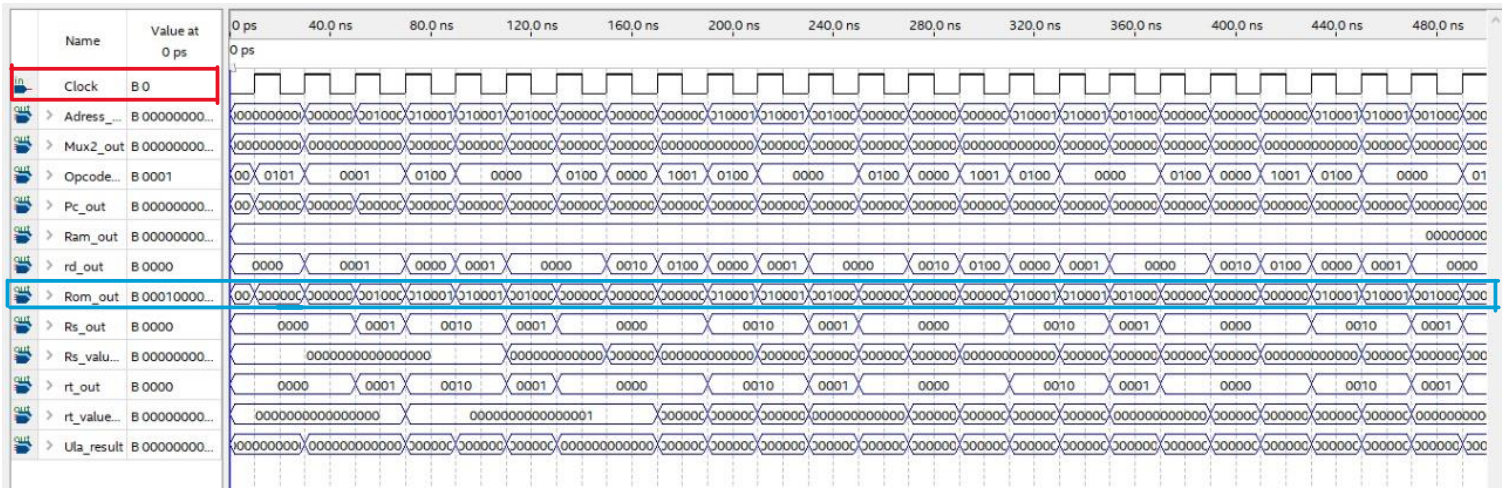

Figura 14 - Resultado na waveform (Fibonacci).

Estes são os pinos de saída para observação dos resultados, entre eles nós podemos citar: PC, Memória de Instruções, ULA, Unidade de Controle e Registradores.

Nesta parte, podemos ver que o processador está pegando as instruções corretamente da ROM e realizando as devidas operações.

Tabela 6 – Código do fatorial.

Endereço	Linguagem de Alto Nível	Binário			
		Opcode	Reg1	Reg2	Reg3
			Endereço		
		Dado			
0	**Li** $S0 8	0110	0000	0000	1000
		0110000000001000			
1	**Li** $S1 1	0110	0001	0001	0001
		0110000100010001			
2	**Li** $S2 1	0110	0001	0001	0001
		0110001000100001			
3	**If** $S0 == $S2	1000	0000	0000	0010
		1000000000000010			
4	**beq** $s2 $s0 jump 1000	0111	0010	0000	1000
		0111001000001000			
5	**mult** $S1 $S1 $S2	1010	0001	0001	0010
		1010000100010010			
6	**addi** $S2 $S2 1	0001	0010	0010	0001
		0001001000100001			
7	**J** 0011	1001	0000	0000	0011
		1001000000000011			

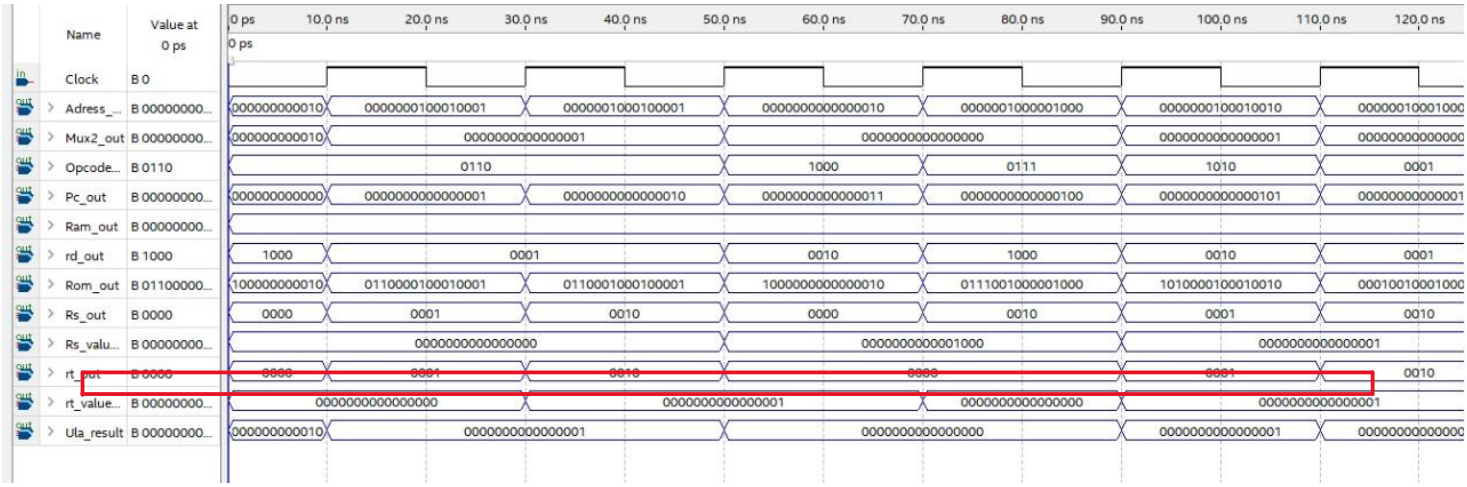

Figura 15 – Resultado na waveform (fatorial)

Ao observarmos a saída da ULA, podemos perceber que o código funciona.

Explicando o funcionamento do código, inicialmente utilizamos a instrução load imediato para carregar um valor que servirá como critério de parada para o cálculo do fatorial; no exemplo, esse valor é 8. Posteriormente, carregamos nos registradores $S1 e $S2 o valor 1, onde um será usado para armazenar o resultado do fatorial e o outro para representar o contador (variável "i").

Em seguida, o código inclui uma estrutura condicional (if) que verifica se $S0 é igual a $S2, indicando que "i" atingiu o valor estabelecido no $S0. Logo após, temos uma instrução de branch equal (beq) que utiliza o resultado da condição do if para determinar se o fluxo de controle deve pular para o label 8, encerrando o laço. Caso a condição não seja satisfeita, o programa prossegue para a próxima instrução.

Posteriormente, o código realiza a multiplicação do valor em $S1 (representando o fatorial) pelo valor em $S2 (representando "i"). Em seguida, incrementa 1 em $S2, indicando o aumento do contador "i". Finalmente, o programa realiza um salto incondicional (jump) de volta à linha 3, reiniciando o ciclo até que a condição de parada seja atendida. Este processo continua até que o fatorial seja calculado conforme o critério estabelecido.

3 Considerações finais

Este trabalho apresentou o projeto e implementação do processador RISC de 16 bits denominado INSALUBYTE. Ficando claro, a descrição de todos os componentes, suas características e respectivas operações fundamentais exigidas nas orientações do projeto.

No entanto, foram encontradas dificuldades no decorrer do projeto. Dentre elas estão:

- Não opera com valores negativos.
- ULA sem suporte para overflow.
- O número de Opcodes é limitado a 16.

Sendo evidente que algumas dessas limitações apresentadas são reflexo da adoção de uma arquitetura baseada em 16 bits. Mesmo com essas limitações apresentadas, o processador INSALUBYTE é totalmente funcional quando utilizamos os 3 formatos de instruções (R, I, J), além de ter mostrado o desenvolvimento de circuitos utilizando VHDL e a utilização do Quartus.

No geral, foi uma boa experiencia voltada ao funcionamento e criação de um processador e a utilização de linguagens de baixo nível, onde foram utilizados os conceitos apresentados em aula.

4 Referências

PATTERSON, D.; HENESSY, J. L. Organização e projeto de computadores: a interface hardware/software. 3ª Edição. São Paulo: Elsevier, 2005.

nataliaalmada - Overview. Disponível em: <https://github.com/nataliaalmada>

ed-henrique - Overview. Disponível em: <https://github.com/ed-henrique>

VHDL code for MIPS Processor. Disponível em: <https://www.fpga4student.com/2017/09/vhdl-code-for-mips-processor.html>