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

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR LP

ALUNOS:

Nataly Almeida dos Santos – 2019016330

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 LP

Novembro de 2023
Boa Vista/Roraima

Resumo

Este trabalho aborda as soluções e métodos feitos para a construção do processador de 16 bits chamado LP (LogPose), desenvolvido por meio da ferramenta chamada *Quartus Lite*. Este nome foi escolhido como referência ao anime *One Piece,* tendo o significado de que LogPose é uma ferramenta que direciona o caminho a próxima ilha pirata segura (ou não) através das configurações magnéticas da própria. Os componentes que serão mostrados neste relatório servirão para base de estudos para alunos da Ciência da Computação juntamente com sua aplicação deste mesmo processador, contando também com avaliação da disciplina de Arquitetura e Organização de Computadores (AOC), ministrada pelo professor Herbert Oliveira Rocha. As informações fornecidas também estarão disponíveis e mais detalhadas no repositório: Repositório LogPose.

Palavras-chave: processador;16 bits; caminho; aplicação; informações; github.

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Especificação

Nesta secção é apresentado o conjunto de itens para o desenvolvimento do processador LP (LogPose), bem como a descrição detalhada de cada etapa da construção do processador.

1.1 Plataforma de desenvolvimento

Para a implementação do processador foi utilizado o IDE Quartus. Este software é usado para programar dispositivos lógicos programáveis, como FPGAs. Ele suporta linguagens como VHDL e Verilog, oferece ferramentas de design, compilação, implementação e análise.

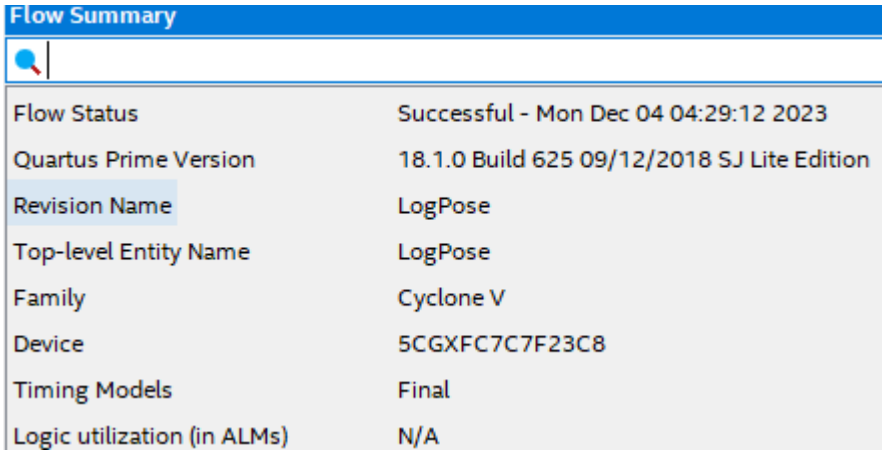


Figura 1 – Especificação

1.2 Conjunto de instruções

Para o funcionamento do Processador LP, de forma que ele possa realizar as operações matemáticas requisitadas, há regras e classificações quanto a formatação das instruções.

1.2.1 Formato de Instrução

O processador LP possui 16 registradores, nomeados: $zero, $highReg, $lowReg, $compareReg, $S0, $S1, $S2, $S3, $S4, $S5, $S6, $S7, $T0, $T1, $T2 e $T3. Cada instrução possui um código de operação (Opcode) composto por 5 bits, resultando em um total de 32 opcodes ($2^5 = 32$). Da mesma forma, o campo de modificação de função (Funct) é constituído por 3 bits, o que implica em 8 Functs para cada Opcode ($2^3 = 8$). Tendo disponível $2^3 * 2^5 = 2^8 = 256$ diferentes operações possíveis. Mais à frente neste relatório, será exibido a tabela sobre o uso dos opcodes.

1.2.2 Classificações de Instrução

- **Formato do tipo R**: Este formato aborda instruções baseadas em operações aritméticas.

Tabela 1 – Tabela de Classe de Instrução - Tipo R

5 bits	4 bits	4 bits	3 bits
Opcode 15-11	Registrador 1 10-7	Registrador 2 6-3	Funct 2-0

- **Formato do tipo I**: Este formato aborda instruções baseadas carregamentos imediatos na memória. Devido à limitação de apenas 16 bits em cada instrução no processador, o tratamento das instruções do tipo I difere significativamente do padrão convencional, como observado no MIPS. No caso dos desvios condicionais, o processo foi subdividido em duas etapas: primeiro, ocorre a avaliação da condição que determinará se o salto será executado ou não. O resultado dessa avaliação é armazenado no registrador $compareReg, designado especificamente para guardar resultados de comparações lógicas. Posteriormente, para realizar o salto condicional, é utilizada uma instrução do tipo J. Essa instrução recebe como argumento o endereço desejado, conforme indicado na tabela 2. O salto condicional ocorre com base nas condições estabelecidas para o registrador $compareReg. Em outras palavras, a instrução do tipo J determina em quais circunstâncias o registrador $compareReg deve estar configurado como True ou False para que o salto seja efetuado. Esse procedimento oferece uma abordagem adaptada à restrição de espaço, permitindo a implementação eficiente de desvios condicionais no processador.

- **Formato do tipo J**: Este formato aborda instruções baseadas desvios condicionais e incondicionais. Exemplo: Jump ou “if ou goTo” (Obs: instruções do tipo J podem ser equivalentes a operações de "jump" ou "salto" no código assembly. Elas são frequentemente utilizadas para implementar desvios condicionais (como um "if") ou desvios incondicionais (como um "goTo"). O termo "jump" é frequentemente usado de maneira mais geral para descrever operações de salto em um programa.

Tabela 2 – Tabela de Classe de Instrução - Tipo J

5 bits	11 bits
Opcode 15-11	Endereço de Destino 10-0

- **Visão geral das instruções do Processador**

Opcode	Nome	Formato	Significado	Funct
00000	add	R	Soma	**0
00000	sub	R	Subtração	**1
00001	addi	R	Soma Imediata	**0
00001	subi	R	Subtração Imediata	**1
00010	and	R	AND logico	*00
00010	andi	R	AND logico imediato	*10
00010	nand	R	NAND logico	*01
00010	nandi	R	NAND lógico imediato	*11

00011	**or**	**R**	**OR logico**	***00**
00011	**ori**	**R**	**OR logico imediato**	*10
00011	**nor**	**R**	**NOR logico**	*01
00011	**nori**	**R**	**NOR logico imediato**	*11
00100	**xor**	**R**	**XOR logico**	*00
00100	**xori**	**R**	**XOR logico imediato**	*10
00100	**xnor**	**R**	**XNOR logico**	*01
00100	**xnori**	**R**	**XNOR logico imediato**	*11
00101	**sra**	**R**	**Shift aritimetico pra direita**	***
00110	**srl**	**R**	**Shift logico pra direita**	***
00111	**slla**	**R**	**Shift para esquerda**	***
01000	**equ**	**R**	**Comparador igual**	**0
01000	**equi**	**R**	**Comparador igual imediato**	**1
01001	**dif**	**R**	**Comparador diferente**	**0
01001	**difi**	**R**	**Comparador diferente imediato**	**1
01010	**sma**	**R**	**Comparador menor que**	**0
01010	**smai**	**R**	**Comparador menor que imediato**	**1
01011	**smeq**	**R**	**Comparador menor que ou igual a**	**0
01011	**smeqi**	**R**	**Comparador menor que ou igual a imediato**	**1
01100	**grt**	**R**	**Comparador maior que**	**0
01100	**grti**	**R**	**Comparador maior que imediato**	**1
01101	**greq**	**R**	**Comparador maior que ou igual a**	**0

01101	**greqi**	**R**	**Comparador maior que ou igual a imediato**	**1
01110	**mult**	**R**	**Multiplicação**	**0
01110	**multi**	**R**	**Multiplicação imediata**	**1
01111	**div**	**R**	**Divisão**	**0
01111	**divi**	**R**	**Divisão Imediata**	**1
10000	**jr**	**I**	**Salto para endereço no registrador**	***
10001	**jim**	**I**	**Salto para endereço imediato**	***
10010	**jrbt**	**I**	**Salto para endereço no registrador se $compareReg = *True***	***
10011	**jrbf**	**I**	**Salto para endereço no registrador se $compareReg = *False***	***
10100	**jimbt**	**I**	**Salto para endereco imediato se $compareReg = *True***	***
10101	**jimbf**	**I**	**Salto para endereco imediato se $compareReg = *False***	***
10110	**ldr**	**R**	**Carrega um dado do endereço no registrador**	***
10111	**ldi**	**R**	**Carrega um dado do endereço imediato**	***
11000	**str**	**R**	**Grava um dado no endereço do registrador**	***
11001	**sti**	**R**	**Grava um dado no endereço imediato**	***
11010	**move**	**R**	**Move um dado para o registrador**	**0
11010	**movi**	**R**	**Move um dado imediato para o registrador**	**1

Tabela 3 – Lista de Opcodes

1.3 Descrição do Hardware

Nesta secção são descritos os componentes do hardware que compõem o Processador LP, incluindo uma descrição de suas funcionalidades, valores de entrada e saída.

1.3.1 Portas lógicas: AND, OR e XOR

Neste processador, foi implementado as portas AND, OR E XOR (adaptando-as para 16bits.) Estas portas lógicas são os blocos de construção básicos para a construção de circuitos mais complexos. Circuitos lógicos digitais utilizam combinações dessas portas para realizar operações mais elaboradas, como adição, subtração, multiplicação, e assim por diante. As diferenças são os seus respectivos nomes, como é implementada e nos seus comportamentos. É gerado pelo Quartus a figura abaixo, seguindo a ordem das portas lógicas: AND, OR e XOR. Sinais de entrada em 16 bits: entrada_a, entrada_b. Sinais de saída feito comparação bit a bit: saidaComp.

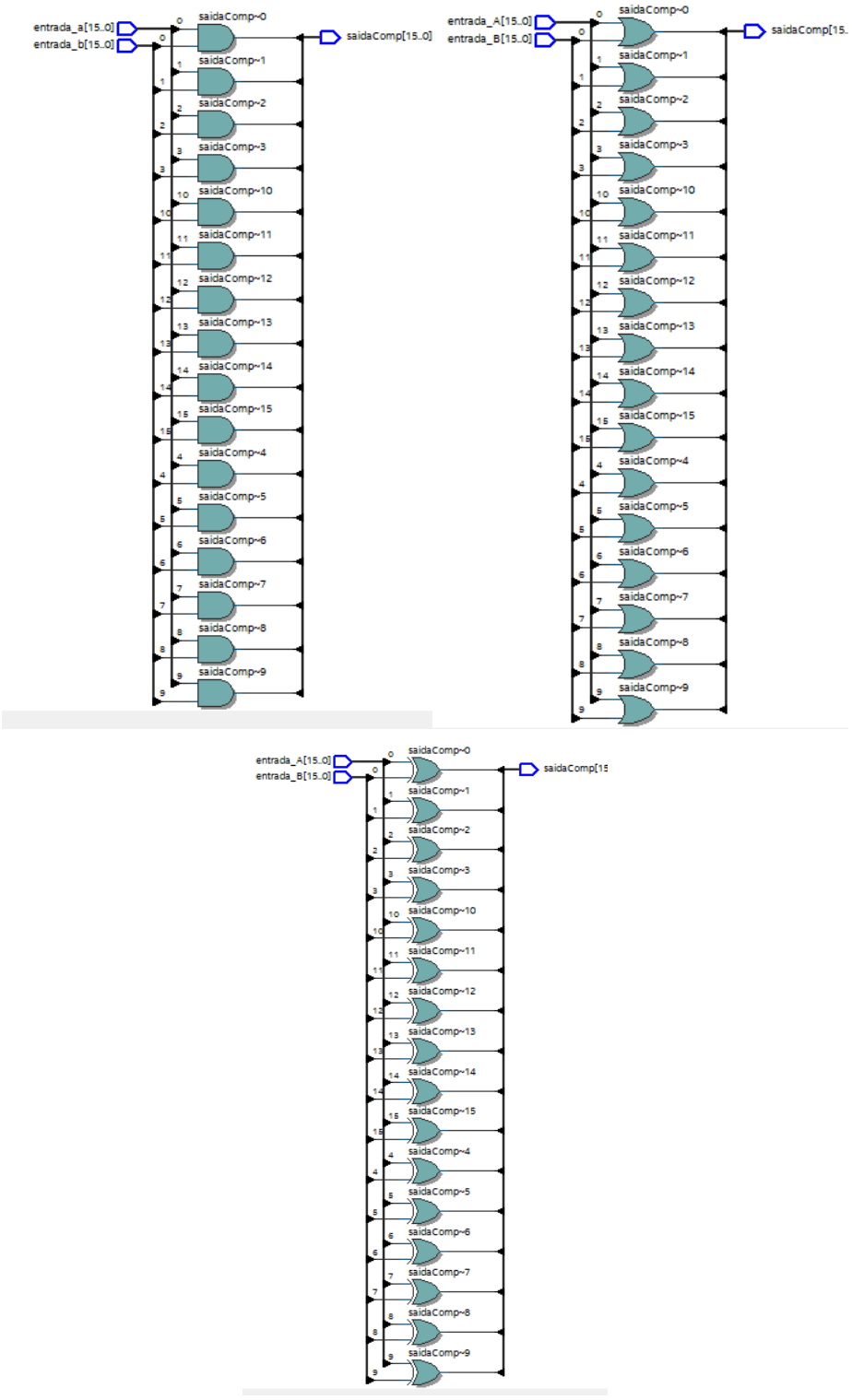


Figura 2 – AND, OR e XOR / Gerado no *Quartus*

1.3.2 *Flip-Flop D*

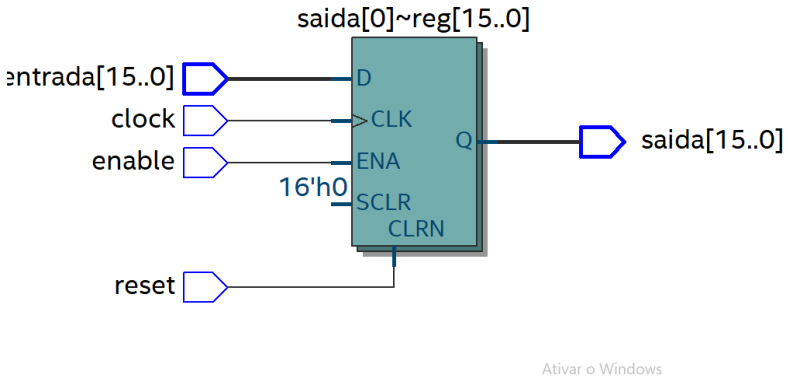


Figura 3 - *Flip-Flop D* / Gerado no *Quartus*

Ele é um tipo de flip-flop que armazena um único bit de informação. A operação básica do flip-flop tipo D envolve a entrada de dados (entrada), a entrada de clock (clock), e muitas vezes uma entrada de controle de habilitação (enable) para determinar quando o flip-flop deve ler a entrada de dados, neste mesmo flip flop há flag de limpeza do registrador (reset), quando 1 o valor armazenado se torna 0's. Como mostrado na figura 3 acima.

1.3.3 Multiplexadores

No processador de 16 bits foi utilizado e implementado 5 tipos de multiplexadores:

- Multiplexador 2x1 – Este possui a capacidade de selecionar entre quatro entradas (entrada_A, entrada_B, entrada_C e entrada_D) com base nos dois bits de Seletor. O valor selecionado é atribuído à saída. Utiliza a instrução with para selecionar qual dos sinais de entrada (entrada_A, entrada_B, entrada_C ou entrada_D) será atribuído à saída com base no valor da entrada de seleção (Seletor.) Quando Seletor é "00", a saída é entrada_A. Quando Seletor é "01", a saída é entrada_B. Quando Seletor é "10", a saída é entrada_C. Para qualquer outro valor de Seletor, a saída é entrada_D.

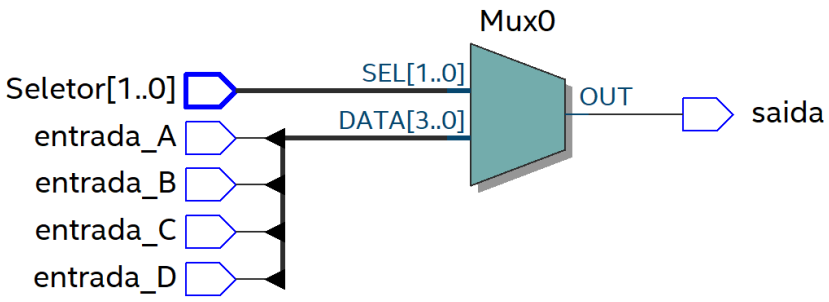


Figura 4 – Multiplexador 2x1 / Gerado no Quartus.

- Multiplexador 1x4 - Nesse multiplexador 1x4, seleciona entre duas entradas de dados (entrada_A e entrada_B) com base no valor da entrada de seleção (Selector). Se Selector for '0', a saída será entrada_A; caso contrário, a saída será entrada_B. Essa estrutura é conhecida como um multiplexador 1x4 porque tem uma entrada de seleção e quatro entradas de dados.

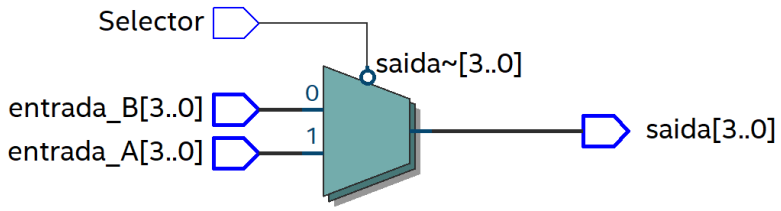


Figura 5 – Multiplexador 1x4 / Gerado no Quartus.

- Multiplexador 3x16 –nesse multiplexador é implementado o seu comportamento. O comando with Seletor select é usado para escolher qual entrada deve ser enviada para a saída com base no valor do seletor. Cada linha dentro do bloco select especifica uma condição para o valor do seletor e a correspondente entrada a ser selecionada. A linha when others é acionada quando nenhuma das condições anteriores é atendida. Basicamente, um multiplexador 3x16 que direciona uma das oito entradas para a saída com base no valor do seletor de 3 bits.

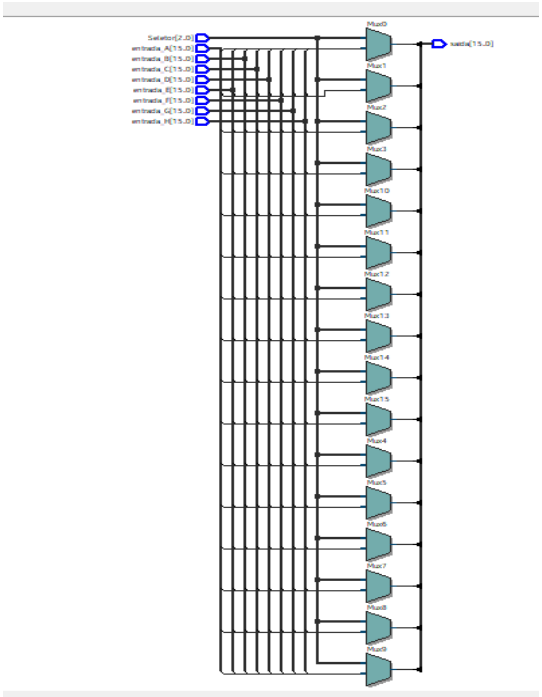


Figura 6 – Multiplexador 3x16 / Gerado no Quartus.

- Multiplexador 1x16 – Abaixo representa um multiplexador 1x16 com duas entradas de 16 bits (entrada_A e entrada_B) e uma saída de 16 bits (saida). O controle é feito por um sinal de seleção de 1 bit (Selector). Se o Selector for '0', a entrada_A é direcionada para a saída; caso contrário, a entrada_B é direcionada para a saída.

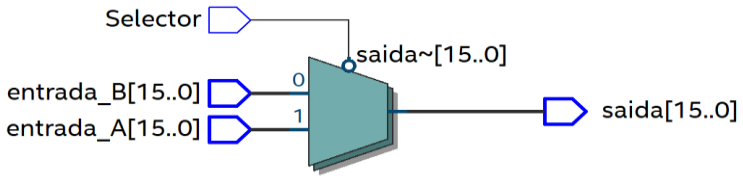


Figura 7 – Multiplexador 1x16 / Gerado no Quartus.

- Multiplexador 4x16: Abaixo representa um multiplexador 4x16 com dezesseis entradas de 16 bits cada (entrada_A até entrada_P) e um seletor de 4 bits (Selector). A saída (saida) é determinada pelo valor do seletor, que escolhe uma das entradas para ser direcionada para a saída.

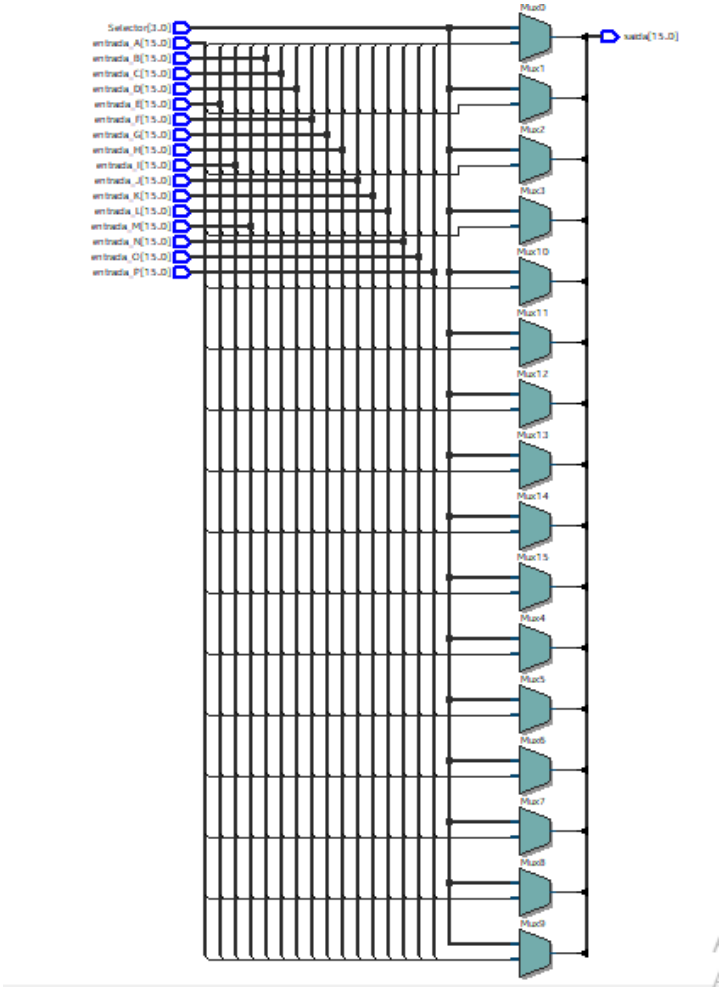


Figura 8 – Multiplexador 4x16 / Gerado no Quartus.

1.3.4 Demultiplexador

Um demultiplexador, comumente chamado de "demux," é um circuito digital que tem como função direcionar um único sinal de entrada para uma das várias saídas possíveis com base em um sinal de controle. No processador há dois “demul”:

- Demultiplexador 4x1: Neste circuito, um sinal de entrada (entrada) é direcionado para uma das 16 saídas possíveis (saida_A a saida_P) com base em um sinal de seleção de 4 bits (Selector). Cada saída é ativada quando a combinação correspondente ao valor do sinal de seleção é encontrada. Se nenhuma combinação for correspondida, todas as saídas são definidas como '0'. No código usado para implementação do Demultiplexador, é utilizado múltiplas declarações with Selector select para associar o valor de entrada às saídas correspondentes, com uma condição específica para cada saída com base no valor do Selector. A atribuição padrão para as saídas é '0' quando nenhuma das condições específicas é atendida (Exemplo: Se Selector for "0101" (binário), então saida_F será atribuído o valor de entrada, e todas as outras saídas serão '0'.)

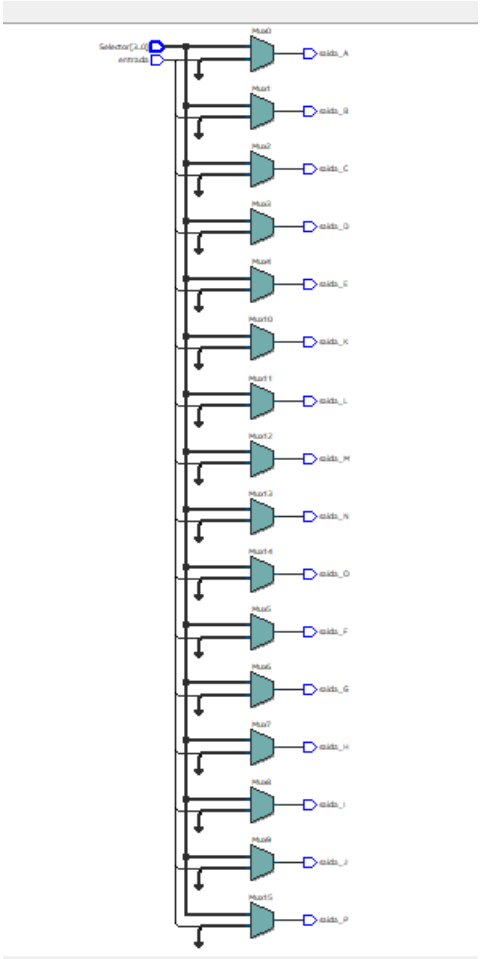


Figura 9 –Demultiplexador 4x1 / Gerado no Quartus

- Demultiplexador 4x16: Este circuito digital recebe um sinal de entrada de 16 bits (entrada) e o direciona para uma das 16 saídas possíveis (saida_A a saida_P) com base em um sinal de seleção de 4 bits (Selector). Cada saída de 16 bits é ativada quando a combinação correspondente ao valor do sinal de seleção é encontrada. Se nenhuma combinação for correspondida, todas as saídas são definidas como um vetor de 16 bits contendo zeros. A entidade demultiplexador4x16 possui três portas de entrada (Selector e entrada) e 16 portas de saída de vetor de 16 bits (saida_A a saida_P). O código utilizado usa-se múltiplas declarações with Selector select para associar o valor de entrada às saídas correspondentes, com uma condição específica para cada saída com base no valor do Selector. Para cada saída (saida_A a saida_P), há uma declaração correspondente with Selector select. A sintaxe é output <= input when "XXXX", "0000000000000000" when others;, onde "XXXX" representa o valor binário do sinal de seleção para aquela saída específica. Para todas as saídas, a atribuição padrão é "0000000000000000" when others;, o que significa que, se nenhuma das condições específicas coincidir, todas as saídas serão definidas como um vetor de 16 bits contendo zeros. Este demultiplexador 4 para 16 permite rotear o sinal de entrada para uma das 16 saídas possíveis com base na configuração do sinal de seleção. Mais na frente será mostrado sua utilização.

1.3.5 PC

O objetivo principal de um contador de programa é rastrear a posição da instrução sendo executada em um programa armazenado na memória de um computador. Ele mantém um valor que representa o endereço da próxima instrução a ser buscada e executada. O componente ProgramCounter possui três portas: clock (entrada), entrada (entrada) e saida (saída). No código implementado, há um processo sensível a mudanças no sinal de clock e entrada. Quando ocorre um flanco de subida no sinal de clock (verificado pela condição if (clock'event AND clock = '1')), a saída (saida) é atualizada com o valor presente na entrada (entrada). O comportamento geral do contador de programa é sincronizado com o sinal de clock, transferindo o valor da entrada para a saída a cada flanco de subida.

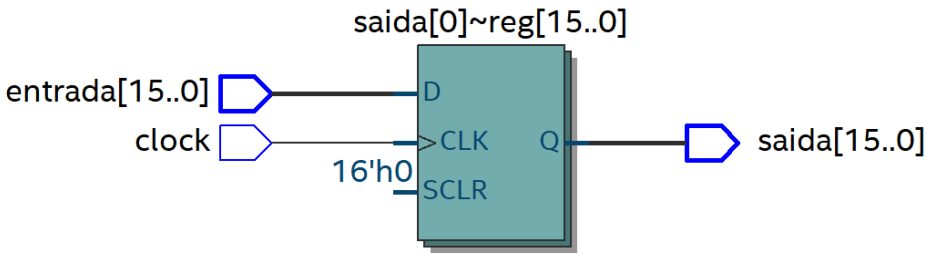


Figura 10 – PC / Gerado no Quartus

1.3.6 Memória ROM

A memória somente de leitura ou ROM é um tipo de memória que permite apenas a leitura, ou seja, as suas informações são gravadas pelo fabricante uma única vez e após isso não podem ser alteradas ou apagadas, somente acessadas. A memória ROM deste processador possui uma porta de entrada address de 16 bits (representando o endereço da memória) e uma porta de saída data de 16 bits (representando os dados lidos da memória ROM). Ainda na implementação, há uma declaração de tipo ROM_Array que define um array que representa o conteúdo da memória ROM. A constante ROM_Content é uma instância desse array que pode ser iniciado com os dados desejados. O processo simplesmente utiliza o valor do endereço para acessar a posição correspondente na memória ROM e coloca os dados associados na porta de saída data. Mais à frente, veremos sua utilização quanto os testes feitos no processador.

1.3.7 Extensor de sinal

Temos dois **extensores** de sinal:

- Extensor 4x16: A função principal deste componente é replicar os 4 bits da entrada nos 4 bits menos significativos da saída (saida(3 downto 0)). Os bits de 15 a 4 da saída são fixados em zeros. Um exemplo é que se a entrada for "1101", a saída seria "0000000000001101". Ou seja, os 4 bits de entrada são replicados nos bits menos significativos da saída, e os bits mais significativos são definidos como zeros.

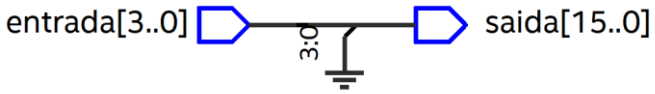


Figura 11 – Extensor 4x16/ Gerado no Quartus

- Extensor 11x16: Estende uma entrada de 11 bits para uma saída de 16 bits. Os 11 bits originais são replicados nos bits menos significativos da saída, e os bits adicionais (de 15 a 10) são preenchidos com o valor do bit mais significativo da entrada. O componente realiza uma extensão de sinal, mantendo os bits originais na posição mais baixa e preenchendo os bits adicionais com o bit mais significativo.

Figura 12 – Extensor 11x16/ Gerado no Quartus

1.3.8 Encurtador de sinal

O encurtador atua como um encurtador de sinal de 16 para 1 bit. Ele recebe uma entrada de 16 bits e gera uma saída de 1 bit. O componente verifica se há pelo menos um bit com valor '1' na entrada de 16 bits. A saída será '1' se pelo menos um bit da entrada for '1'; caso contrário, a saída será '0'.

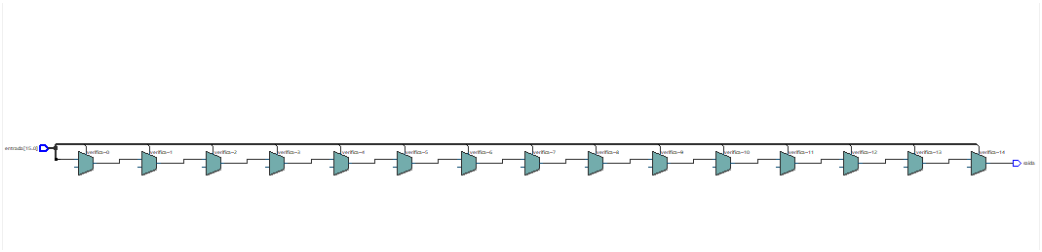


Figura 13 – *Encurtador 16x1/ Gerado no Quartus*

1.3.9 Deslocadores de bit para esquerda/direita

Deslocadores para **esquerda** realizam um deslocamento para a esquerda '***shift left'*** em uma entrada de 16 bits. Esse deslocamento move cada bit da entrada uma posição para a esquerda, e o bit mais à esquerda (MSB) é preenchido com '0'. Já realizando um deslocamento para a **direita** em uma entrada de 16 bits,o processo é que cada bit da saída é movido uma posição para a direita em relação ao bit correspondente na entrada, mantendo o bit mais à direita na mesma posição. O componente não realiza extensão de sinal e utiliza um loop para realizar o deslocamento. O componente efetua um '***shift right'*** em uma entrada de 16 bits. A seguir temos a representação. Mais abaixo, temos o deslocador de bit logico para direita.

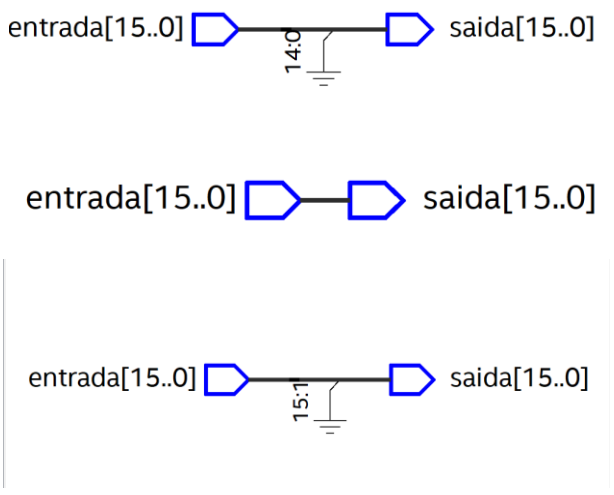


Figura 14 – *Deslocadores de bit esquerda (cima) e direita (embaixo) lógico direita (último abaixo)/ Gerado no Quartus*

1.3.10 Memória RAM

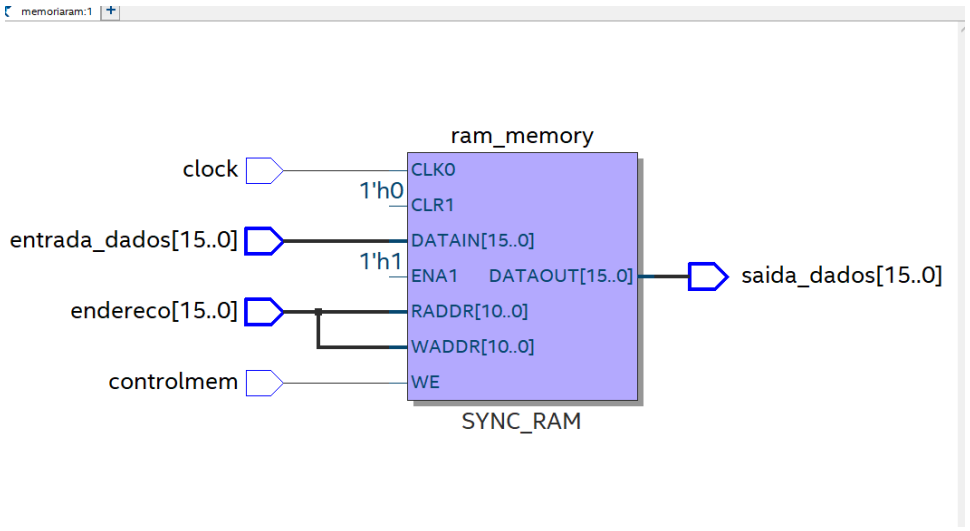
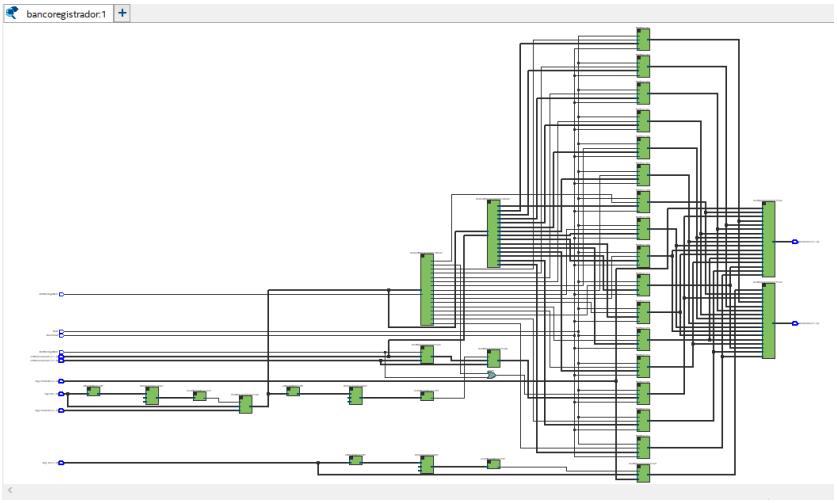


Figura 15 – Memória RAM / Gerado no Quartus.

Conceitualmente falando, a RAM (Random Access Memory) é um tipo de memória volátil utilizada em sistemas computacionais para armazenar dados temporariamente. Ela é fundamental para o funcionamento dos computadores e outros dispositivos eletrônicos. Sobre a implementação feito acima no processador LP (LogPose), é visto que representa uma memória RAM de 16 bits com capacidade para 2001 endereços. A memória é síncrona, operando na borda de subida do clock. A entidade possui portas para entrada de clock, controle de memória, endereço, dados de entrada e saída de dados. O código implementa uma matriz (ram_memory) que representa a memória física e um processo sensível à borda de subida do clock. Durante uma operação de escrita (quando controlmem é '1'), os dados de entrada são escritos no endereço especificado. Durante uma operação de leitura (quando controlmem é '0'), a saída de dados é atribuída ao valor armazenado no endereço de leitura.

1.3.11 Banco de Registradores

Para o banco de registradores, foi implementado juntamente com outros componentes: multiplexadores, extensores e encurtadores de sinais, comparadores aritmeticos, demultiplexadores, porta OR e flipflop. Todos eles foram descritos e implementados neste relatório. Abaixo temos a visualização do banco de registradores do processador LP.



- clock: Sinal de clock para sincronização.
- clearReset: Sinal de reset.
- controlRegister1, controlRegister2: Sinais de controle para operações no banco de registradores.
- regEndereco1, regEndereco2: Endereços dos registradores.
- regLer1: Sinal de controle para leitura de registradores.
- regEsc: Sinal de controle para escrita em registradores.
- entradaDadosR1, entradaDadosR2: Dados a serem escritos nos registradores.
- lerDados1, lerDados2: Saídas para os dados lidos dos registradores.

1.3.12 Unidade de Controle

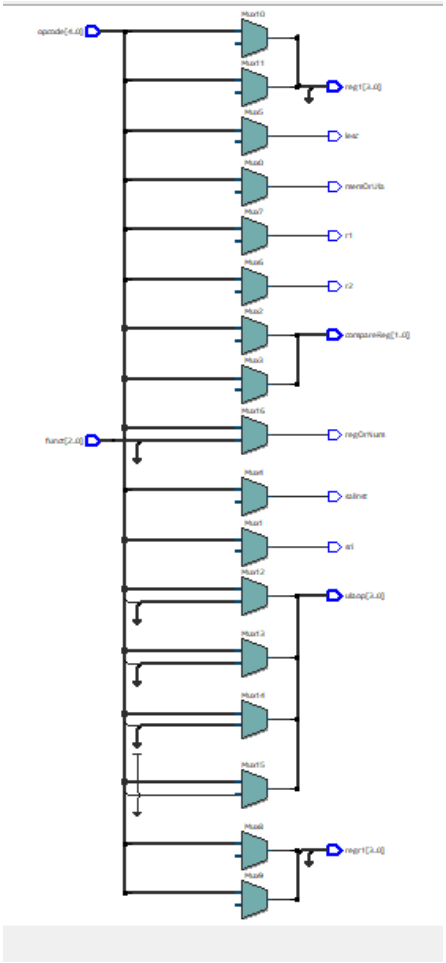


Figura 16 – Unidade de Controle / Gerado no Quartus.

A unidade de controle desempenha um papel crucial, coordenando e controlando as diversas operações internas com base nas instruções recebidas. Sua função principal é interpretar os códigos de operação (opcode) e funções associadas, provenientes das instruções do programa, para gerar sinais de controle que dirigem o comportamento do processador. No processador LP, a unidade de controle gera sinais para indicar à Unidade Lógica Aritmética (ULA) qual operação deve ser realizada, dependendo do opcode. Flags como r1, r2, regr1, reg1 indicam operações de leitura e escrita em registradores, dependendo das instruções. “Sri”, “compareReg”, “salInst” estão relacionadas ao controle de instruções de salto, indicando se o salto é direto, se deve comparar registradores, ou se uma instrução de salto está presente. A “lesc” indicam se a instrução envolve leitura ou escrita na memória RAM. E por último, “regOrNum” indica se a ULA deve utilizar o valor de um registrador ou um número como constante em operações específicas.

1.3.13 Somador/Sub

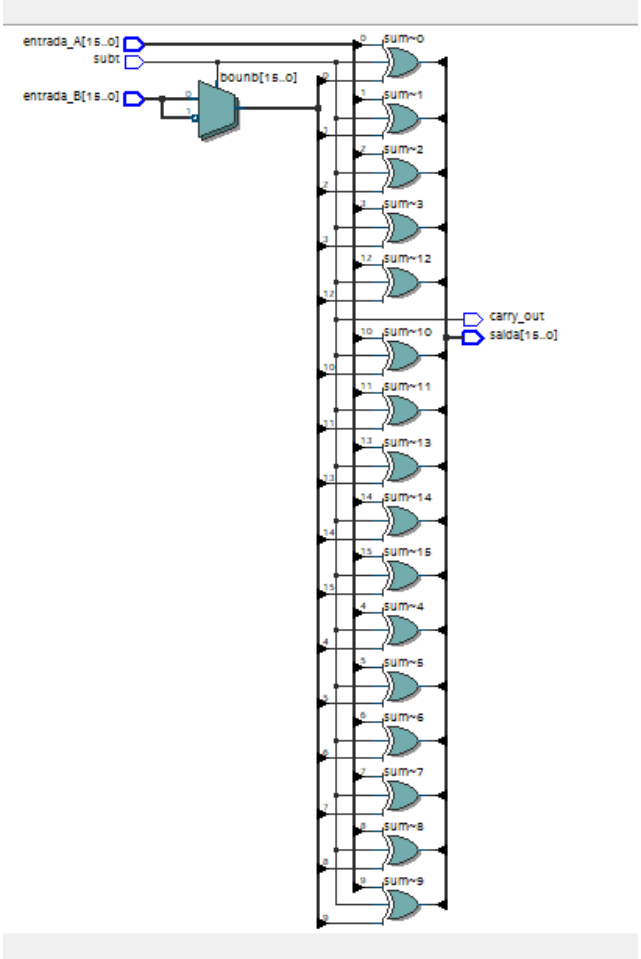


Figura 17 – Somador Sub / Gerado no Quartus.

É importante ressaltar para que ocorra as operações na ULA no processador LP, também é necessário adicionar o somador. Como visto na figura acima, é representado um somador/subtrator de números binários de 16 bits. A entidade somador possui entradas para os operandos A e B, um sinal de subtração (subt), e saídas para o resultado da operação (saida) e um sinal de carry (carry_out). O processo dentro da arquitetura somadorsub realiza a lógica de soma/subtração, onde a subtração ocorre se o sinal subt é '1'. O código utiliza operações lógicas bit a bit, como XOR, para calcular a soma/subtração e atualiza as saídas correspondentes.

1.3.14 Multiplicador por Algoritmo de Booth

Outra implementação fortíssima que faz parte da ULA do processador LP, é o Multiplicador. Para introduzir o conceito, foi utilizado o Algoritmo de Booth que nada mais é que uma multiplicação que multiplica dois números binários com sinal na notação de complemento de 2. Booth usou calculadoras de mesa que eram mais rápidas na mudança do que na adição e criou o algoritmo para

aumentar sua velocidade. No contexto desta implementação, é esperado que haja uma melhor otimização quanto a sua execução no geral. Abaixo, vemos a implementação deste pelo Quartus:

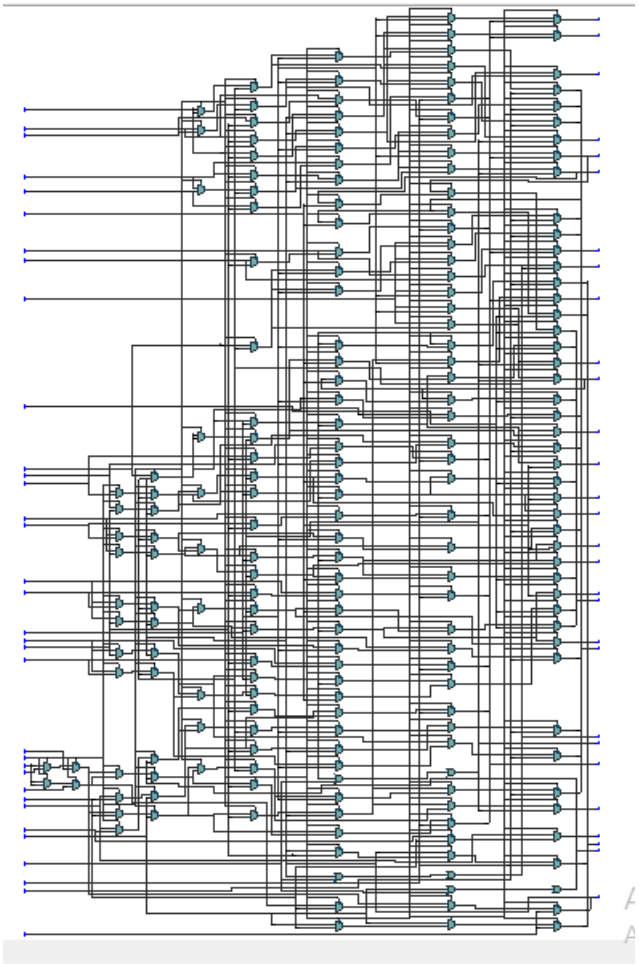


Figura 18 – Multiplicador/ Gerado no Quartus.

Acima é representado um multiplicador de números sem sinal de 16 bits, empregando o método de multiplicação parcial. Ele utiliza um registrador de produto de 32 bits e realiza uma iteração sobre os bits do multiplicador, acumulando o resultado parcial. Elw incorpora uma função de deslocamento à esquerda e verificações condicionais para determinar as adições necessárias no registrador de produto. As partes superiores e inferiores do resultado são atribuídas aos sinais de saída (saidMaior e saidMenor). Especificações melhores sobre como foi feito passo –a-passo no repositório.

1.3.15 Divisor

No processador, há o divisor de números de 16 bits usando o algoritmo da divisão longa. O divisor é recebido nas entradas entrada_A e entrada_B, e o resultado da divisão é enviado para as saídas saidMenor (quociente) e saidMaior (resto). É utilizado um processo sensível às mudanças nas entradas e tem uma função para realizar deslocamentos de bits para a esquerda. Se o divisor for

zero, as saídas são definidas como zero. O algoritmo realiza a divisão bit a bit, atualizando o resto e o quociente em cada iteração.

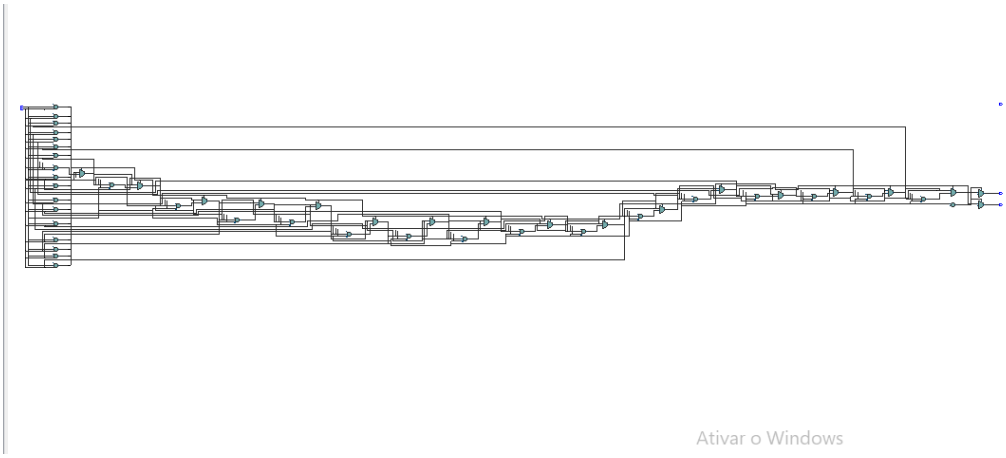


Figura 19 – Divisor / Gerado no Quartus.

1.3.16 Comparador

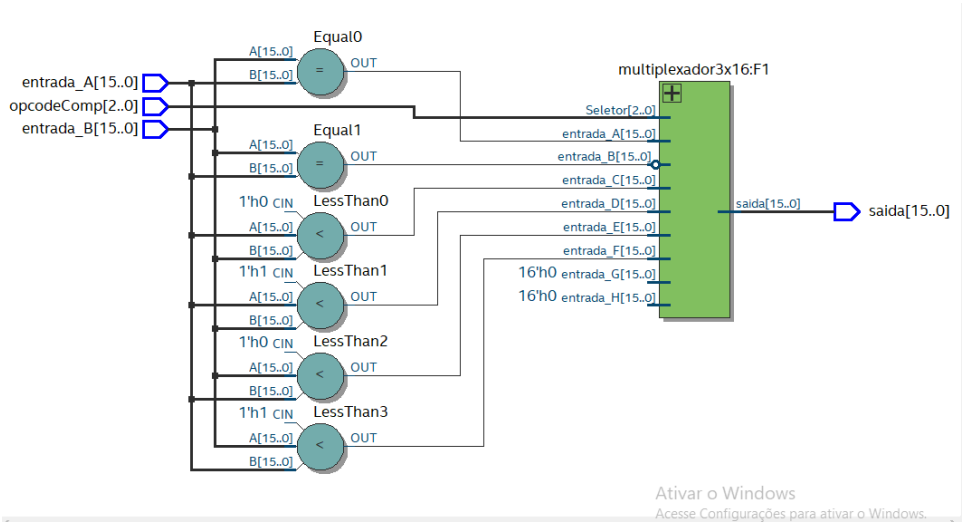


Figura 20 – Comparador / Gerado no Quartus.

A figura acima é um comparador de números de 16 bits que compara duas entradas, entrada_A e entrada_B, e produz uma saída com base no código de operação opcodeComp. É utilizado juntamente com o multiplexador3x16 apresentado no início do relatório. A lógica do comparador é a seguinte:

- saida000: É "1111111111111111" quando entrada_A é igual a entrada_B, senão é "0000000000000000".

- saida001: É "1111111111111111" quando entrada_A é diferente de entrada_B, senão é "0000000000000000".
- saida010: É "1111111111111111" quando entrada_A é menor que entrada_B, senão é "0000000000000000".
- saida011: É "1111111111111111" quando entrada_A é menor ou igual a entrada_B, senão é "0000000000000000".
- saida100: É "1111111111111111" quando entrada_A é maior que entrada_B, senão é "0000000000000000".
- saida101: É "1111111111111111" quando entrada_A é maior ou igual a entrada_B, senão é "0000000000000000".
- saida111: Sempre é "0000000000000000".

As comparações são definidas com base no valor das entradas e são selecionadas pelo componente multiplexador3x16 com a ajuda do opcodeComp. Como um multiplexador de 3 entradas, seleciona a saída correspondente à operação que deseja.

1.3.17 ULA

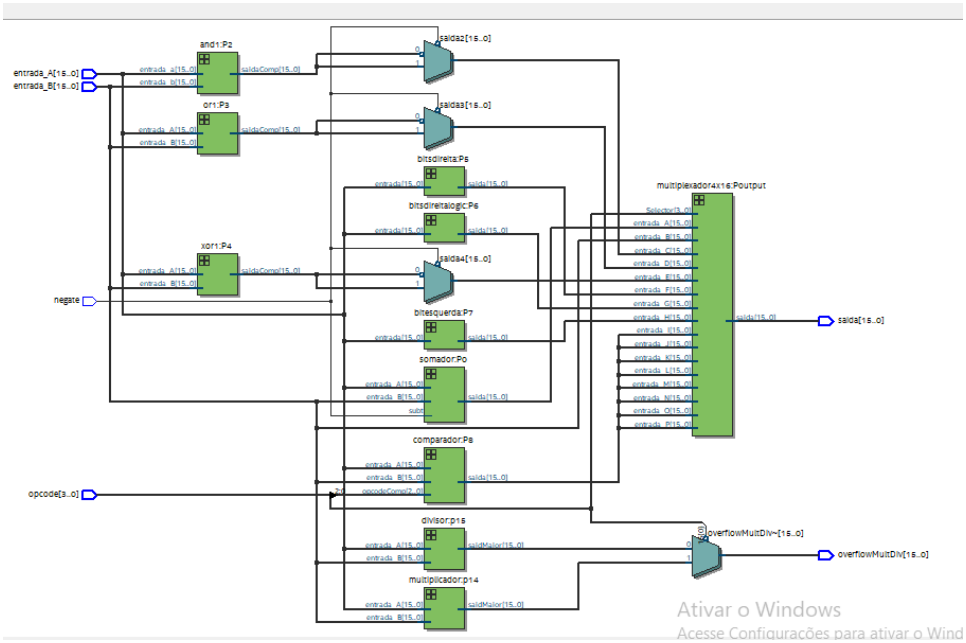


Figura 21– ULA / Gerado no Quartus.

No processador LP, para construção da nossa ULA (acima) foram necessários alguns componentes essenciais para seu funcionamento, tais como: somador, subtrador, comparador, deslocadores de bit esquerdo e direito (incluindo o logico), multiplicador, divisor, multiplexador e as portas logicas and, or e xor, todas apresentadas anteriormente durante o relatório. Basicamente, a ULA é responsavel por fazer essas operações e armazenar os valores conforme vai sendo pedido e executado. No processo, ocorre a declaração de sinais internos que armazenarão os resultados intermediários, implementação de instâncias desses componentes (p0, p2, p3, ..., p15), utiliza-se

multiplexadores pra poder selecionar a saída com base no Opcode e o resultado final é selecionado por um multiplexador que é a saída da ULA. Aqui estão as operações nesta ULA:

Opcode	Descrição	Negate
0000	SOMA	0
0000	SUBTRAÇÃO	1
0001	VOLTA VALOR 2º OP	*
0010	AND LÓGICO	0
0010	NAND LOGICO	1
0011	OR LOGICO	0
0011	NOR LOGICO	1
0100	XOR LOGICO	0
0100	XNOR LOGICO	1
0101	DES. BIT DIREITA	*
0110	DES. LOG. BIT DIREITA	*
0111	DES. BIT ESQUERDA	*
1000	COMPARADOR IGUAL	*
1001	COMPARADOR DIFERENTE	*
1010	COMPARADOR MENOR QUE	*
1011	COMP. MENOR QUE OU IGUAL	*
1100	COMPARADOR MAIOR QUE	*
1101	COMP. MAIOR QUE OU IGUAL	*
1110	MULTIPLICAÇÃO	*
1111	DIVISÃO	*

Tabela 4 – Operações na ULA.

1.4 Datapath

Por fim, temos a visão geral o processador LP na sua forma, com junção de todos os componentes citados anteriormente.

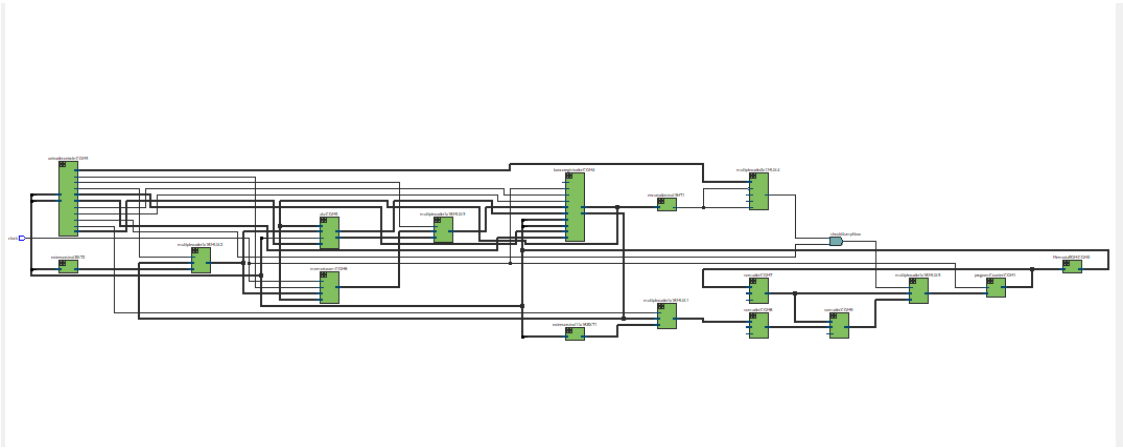


Figura 22– Processador LP (LogPose) / Gerado no Quartus

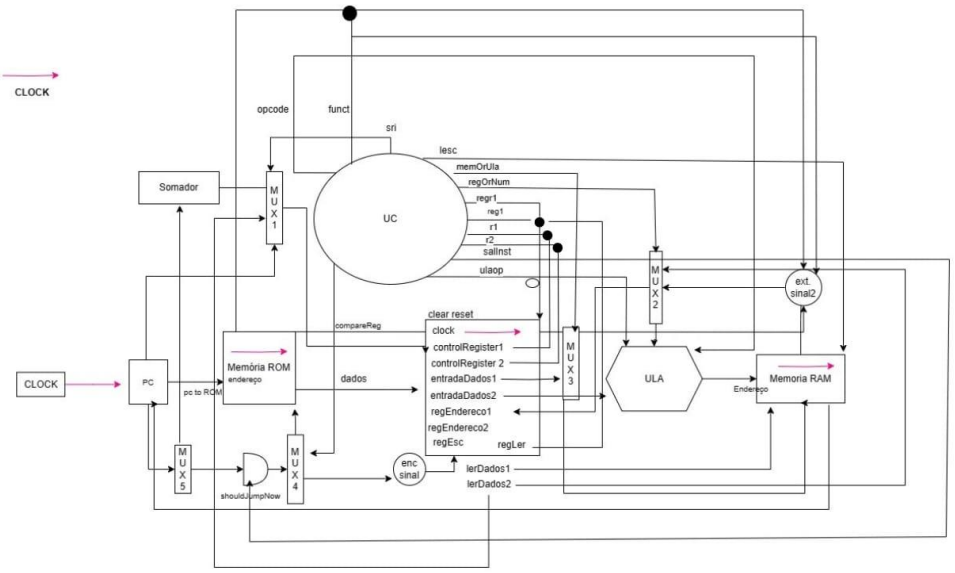


Figura 23– Processador LP (LogPose) / Feito no Draw.io

2 Simulações e Testes

Objetivando analisar, verificando o funcionamento do processador, foi efetuado testes analisando cada componente do processador em específico, em seguida efetuamos testes de cada instrução que o processador implementa. Com a finalidade de demonstrar o funcionamento do processador LP. Utilizando como exemplo o código para calcular o número da sequência de Fibonacci e Bubble Sort.

2.1 Fibonacci

A sequência de Fibonacci é uma sequência de números, onde cada número é a soma dos 2 números anteriores, abaixo vemos a implementação completa dos primeiros 20 números da sequência de Fibonnaci no processador LP:

SINTAXE	OPCODE R1 R2 FUNCT INSTRUÇÃO
01 main : movi $s0 , 0;	11010 0100 0000 001 1101001000000001
02 movi $s1 , 1;	11010 0101 0001 001 1101001010001001
03 movi $s3 , 15;	11010 0111 1111 001 1101001111111001
04 addi $s3 , 5;	00001 0111 0101 000 0000101110101000
05 movi $s4 , 2;	11010 1000 0010 001 1101010000010001
06 smeq $s3 , $zero ;	01011 0111 0000 000 0101101110000000
07 jimbt Endg ;	10100 00000001111 1010000000001111
08 move $s2 , $s0 ;	11010 0110 0100 001 1101001100100000
09 equ $s3 , $s1 ;	01000 0111 0101 000 0100001110101000
10 jimbt Endg ;	10100 00000001100 1010000000001100
11 move $s2 , $s1 ;	11010 0110 0101 001 1101001100101000
12 equi $s3 , 1;	01000 0111 0001 001 0100001110001001
13 jimbt Endg ;	10100 00000001001 1010000000001001
14 lpng : move $t0 , $s0 ;	11010 1100 0100 000 1101011000100000
15 add $t0 , $s1 ;	00000 1100 0101 000 0000011000101000
16 move $s2 , $t0 ;	11010 0110 1100 000 1101001101100000
17 move $s0 , $s1 ;	11010 0100 0101 000 1101001000101000
18 move $s1 , $s2 ;	11010 0101 0110 000 1101001010110000
19 addi $s4 , 1;	00001 1000 0001 000 0000110000001000
20 equ $s3 , $s4 ;	01000 0111 1000 000 0100001111000000
21 jimbt lpng ;	10100 11111111001 1010011111111001
22 Endg :	

2.2 Simulação Fibonacci

Na primeira simulação, é possivel ver os valores sendo alocados inicialmente, como o 5, 8 e 13 (5+8=13). Nesta situação, o teste foi feito entre 90 e 120 n/s.

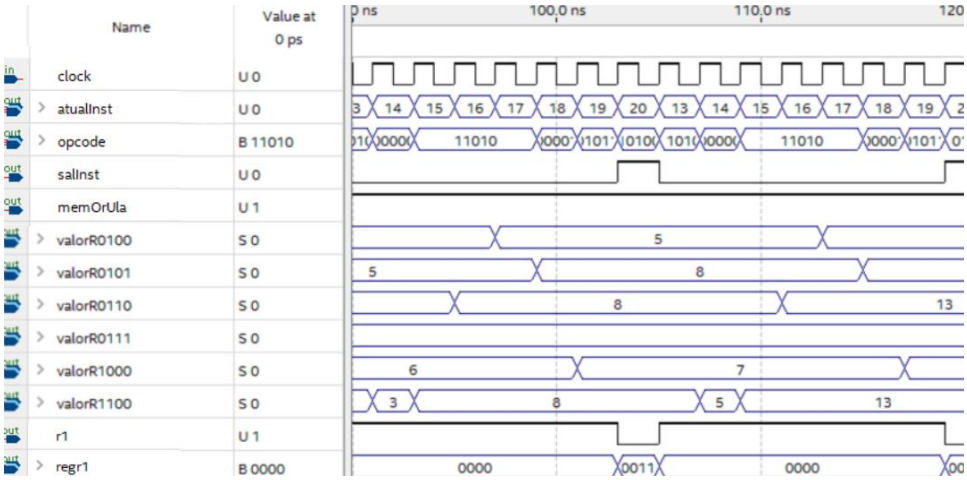


Figura 24– Fibonacci/ Feito no Waveform

Na próxima simulação é possivel ver todos os valores gerados a partir de 120 a 320 n/s. Levando em consideração a subida de clock.

Figura 25– Fibonacci 2 / Feito no Waveform

2.3 Algoritmo de Bubble Sort

O Bubble Sort é um algoritmo de ordenação que pode ser aplicado a Arrays e Listas dinâmicas. Para ordenar em forma decrescente, ele compara a posição atual com a próxima. Se a posição atual for maior que a posição posterior, os valores são trocados; caso contrário, não há troca, e o algoritmo passa para o próximo par de comparações. Para ordenar em forma crescente, a lógica é semelhante, mas a troca ocorre se a posição atual for menor que a posição posterior. O processo continua até que nenhum elemento seja trocado em uma passagem completa pelos dados. Este

algoritmo tem uma eficiência geralmente baixa, sendo mais eficaz em conjuntos de dados pequenos.
Código para realização do Bubble Sort:

SINTAXE	OPCODE	R1	R2	FUNCT	INSTRUÇÃO
01 jim main ;	10001	00000	10001		1000100000010001
02 prt1 : movi $t1 , 0;	11010	1101	0000	001	1101011010000001
03 lpn1 : equ $t1 , $s1 ;	01000	1101	0101	000	0100011010101000
04 jimbt bck1 ;	10100	00000	10000		1010000000100000
05 move $t3 , $s0 ;	11010	1111	0100	000	1101011110100000
06 add $t3 , $t1 ;	00000	1111	1101	000	0000011111101000
07 ldr $s7 , $t3 ;	10110	1011	1111	000	1011010111111000
08 addi $t1 , 1;	00001	1101	0001	000	0000111010001000
09 jim lpn1 ;	10001	11111111010			1000111111111010
10 prt2 : movi $t1 , 0;	11010	1101	0000	001	1101011010000001
11 lpn2 : equ $t1 , $s1 ;	01000	1101	0101	000	0100011010101000
12 jimbt bck2 ;	10100	00000101110			1010000000101110
13 move $t3 , $s0 ;	11010	1111	0100	000	1101011110100000
14 add $t3 , $t1 ;	00000	1111	1101	000	0000011111101000
15 ldr $s7 , $t3 ;	10110	1011	1111	000	1011010111111000
16 addi $t1 , 1;	00001	1101	0001	000	0000111010001000
17 jim lpn2 ;	10001	11111111010			1000111111111010
18 main : movi $s0 , 0;	11010	0100	0000	001	1101001000000001
19 movi $s1 , 5;	11010	0101	0101	001	1101001010101001
20 move $t0 , $s0 ;	11010	1100	0100	000	1101011000100000
21 movi $t1 , 5;	11010	1101	0101	001	1101011010101001
22 str $t1 , $t0 ;	11000	1101	1100	000	1100011011100000
23 addi $t0 , 1;	00001	1100	0001	000	0000111000001000
24 movi $t1 , 3;	11010	1101	0011	001	1101011010011001
25 str $t1 , $t0 ;	11000	1101	1100	000	1100011011100000
26 addi $t0 , 1;	00001	1100	0001	000	0000111000001000
27 movi $t1 , 9;	11010	1101	1001	001	1101011011001001
28 str $t1 , $t0 ;	11000	1101	1100	000	1100011011100000
29 addi $t0 , 1;	00001	1100	0001	000	0000111000001000
30 movi $t1 , 7;	11010	1101	0111	001	1101011010111001
31 str $t1 , $t0 ;	11000	1101	1100	000	1100011011100000
32 addi $t0 , 1;	00001	1100	0001	000	0000111000001000
33 movi $t1 , 1;	11010	1101	0001	001	1101011010001001
34 str $t1 , $t0 ;	11000	1101	1100	000	1100011011100000
35 jim prt1 ;	10001	11111011111			1000111111011111
36 bck1 : movi $t0 , 0;	11010	1100	0000	001	1101011000000001
37 move $t1 , $s1 ;	11010	1101	0101	000	1101011010101000
38 subi $t1 , 1;	00001	1101	0001	001	0000111010001001
39 lpn3 : movi $t2 , 0;	11010	1110	0000	001	1101011100000001
40 move $t3 , $t1 ;	11010	1111	1101	000	1101011111101000
41 sub $t3 , $t0 ;	00000	1111	1100	001	0000011111100001
42 lpn4 : move $s2 , $s0 ;	11010	0110	0100	000	1101001100100000
43 add $s2 , $t2 ;	00000	0110	1110	000	0000001101110000
44 ldr $s7 , $s2 ;	10110	1011	0110	000	1011010110110000
45 addi $s2 , 1;	00001	0110	0001	000	0000101100001000
46 ldr $s6 , $s2 ;	10110	1010	0110	000	1011010100110000

```
47 sma $s6 , $s7 ;          01010 1010 1011 000 0101010101011000
48 jimbf nswp ;             10101 00000000100 1010100000000100
49 str $s7 , $s2 ;          11000 1011 0110 000 1100010110110000
50 subi $s2 , 1;            00001 0110 0001 001 0000101100001001
51 str $s6 , $s2 ;          11000 1010 0110 000 1100010100110000
52 nswp : addi $t2 , 1;     00001 1110 0001 000 0000111100001000
53 sma $t2 , $t3 ;          01010 1110 1111 000 0101011101111000
54 jimbt lpn4 ;             10100 11111110100 1010011111110100
55 addi $t0 , 1;            00001 1100 0001 000 0000111000001000
56 sma $t0 , $t1 ;          01010 1100 1101 000 0101011001101000
57 jimbt lpn3 ;             10100 11111101110 1010011111101110
58 jim prt2 ;               10001 11111010000 1000111111010000
59 bck2 :
```

Explicando as linhas no geral:

- 2 a 17 -Funções de impressão, mostra a disposição dos elementos no vetor
- 18 a 34 – Vetor preenchido
- 36 a 57 – Algoritmo, ordenação

2.4 Simulações Bubble Sort

Neste processo com a subida de clock, e de 0 a 30 n/s e a figura 28 de 90 a 120 n/s , segue-se o algoritmo:

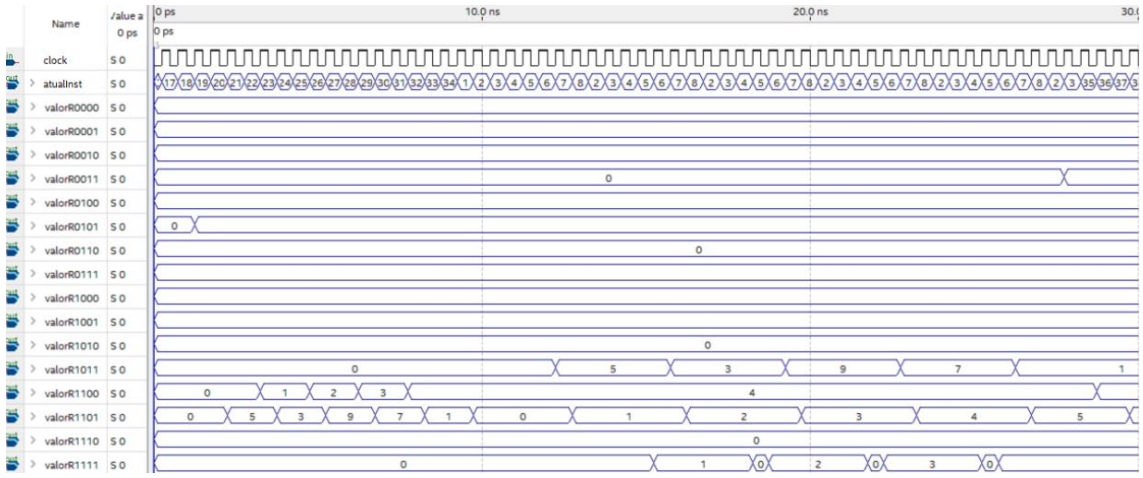


Figura 27–Bubble Sort / Feito no Waveform

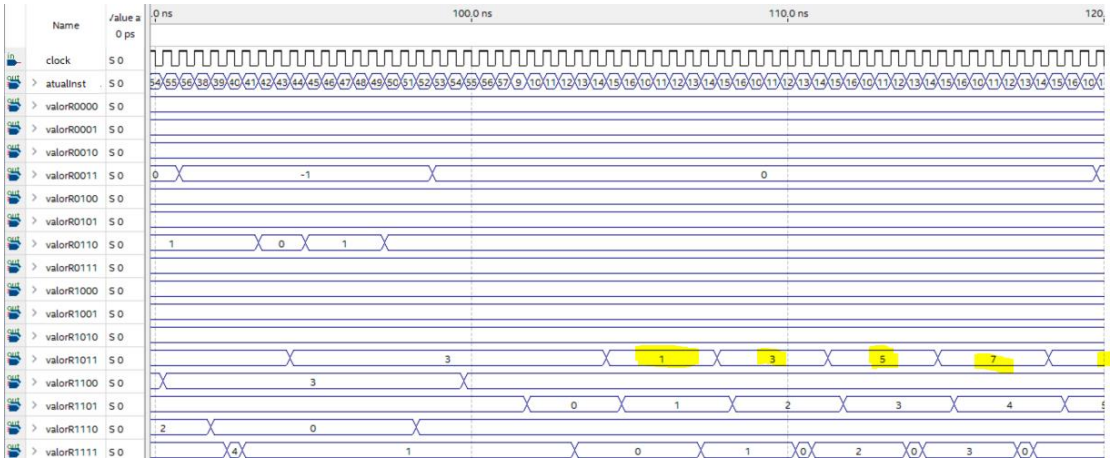


Figura 28–Bubble Sort 2 / Feito no Waveform

3 Considerações finais

Este trabalho destacou o desenvolvimento do processador de 16 bits denominado LogPose, representando um marco significativo no entendimento da arquitetura de computadores. Ao longo da disciplina, assimilamos a importância dessa disciplina não apenas dentro da sala de aula, mas também em sua aplicação prática além dos limites acadêmicos. O projeto aprofundou ainda mais nosso conhecimento, oferecendo uma experiência valiosa aos alunos, facilitada pela orientação do Prof. Herbert Oliveira. Essa imersão prática não apenas fortaleceu nosso aprendizado, mas também ressaltou a relevância e complexidade da arquitetura de computadores em diversos contextos, preparando-nos de maneira abrangente para os desafios futuros.