

**PODER EXECUTIVO**

**MINISTÉRIO DA EDUCAÇÃO**

**UNIVERSIDADE FEDERAL DE RORAIMA**

**DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO**

**ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES**

**RELATÓRIO DO PROJETO: PROCESSADOR MASON**

**ALUNOS:**

**Jasson Marques Fontoura Júnior – 2019014031**

**Marcos Vinicius Melo da Silva - 2019017919**

**Maio de 20****21**

**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 MASON**

**Maio de 20****21**

**Boa Vista/Roraima**

**Resumo**

Este trabalho aborda o projeto e implementação do processador MASON uni-ciclo de 8 bits com base na arquitetura do processador MIPS. Todos os componentes necessários para o seu funcionamento e os testes realizados durante a implementação serão descritos. O processador em questão tem a capacidade de executar 12 instruções, incluindo instruções de comparação, instruções de salto e instruções aritméticas na memória. Usando as instruções disponíveis no processador, ele pode realizar uma variedade de operações diferentes, como Código de Fibonacci e outros algoritmos. Toda a parte de implementação é realizada utilizando a linguagem VHDL e o software Quartus Prime Lite, e os testes dos componentes foram realizados por meio da waveform no software ModelSim Altera, mostrando assim todas as funções de cada componente.

**Conteúdo**

[1 Especificação 6](#_Toc72011651)

[1.1 Plataforma de desenvolvimento 6](#_Toc72011652)

[1.2 Conjunto de instruções 6](#_Toc72011653)

[1.3 Descrição do Hardware 8](#_Toc72011654)

[1.3.1 UAL 8](#_Toc72011655)

[1.3.2 Registradores 9](#_Toc72011656)

[1.3.3 Extensores de Sinal 9](#_Toc72011657)

[1.3.4 Controle 10](#_Toc72011658)

[1.3.5 memoriaAdress 11](#_Toc72011659)

[1.3.6 memoriaInst 11](#_Toc72011660)

[1.3.7 addPC 12](#_Toc72011661)

[1.3.8 portAND 13](#_Toc72011662)

[1.3.9 Multiplexador\_2x1\_8bits 13](#_Toc72011663)

[1.3.10 PC 14](#_Toc72011664)

[1.3.11 divisaoDeInstr 14](#_Toc72011665)

[1.4 Datapath 15](#_Toc72011666)

[2 Simulações e Testes 17](#_Toc72011667)

[3 Considerações finais 21](#_Toc72011668)

[4 Referências Bibliográficas 21](#_Toc72011669)

**Lista de Figuras**

[Figura 1 – Especificações no Quartus 6](#_Toc72011638)

[Figura 2 – RTL viewer do componente UAL 9](#_Toc72011639)

[Figura 3 – RTL viewer do componente Registradores 9](#_Toc72011640)

[Figura 4 – RTL viewer do componente Extensor de Sinal 2\_8 10](#_Toc72011641)

[Figura 5 – RTL viewer do componente Extensor de Sinal 4\_8 10](#_Toc72011642)

[Figura 6 – RTL viewer de componente memoriaAdress 11](#_Toc72011643)

[Figura 7 – RTL viewer do componente memoriaInst 12](#_Toc72011644)

[Figura 8 – RTL viewer do componente addPC 13](#_Toc72011645)

[Figura 9 – RTL viewer do componente portAND 13](#_Toc72011646)

[Figura 10 – RTL viewer do componente Multiplexador\_2x1\_8bits 13](#_Toc72011647)

[Figura 11 – RTL viewer do componente PC 14](#_Toc72011648)

[Figura 12 – RTL viewer do componente divisaoDeInstr 14](#_Toc72011649)

[Figura 13 – RTL viewer do processador MASON 15](#_Toc72011650)

**Lista de Tabelas**

Tabela 1 – Tabela que mostra a lista de Opcodes utilizadas pelo processador MASON. 7

Tabela 2 - Detalhes das flags de controle do processador. 10

Tabela 3 - Código Fibonacci para o processador Quantum. 17

# Especificação

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

## Plataforma de desenvolvimento

Para a implementação do processador MASON foi utilizado a IDE: Quartus Prime Lite, versão 20.1.1 e o simulador ModelSim Altera.

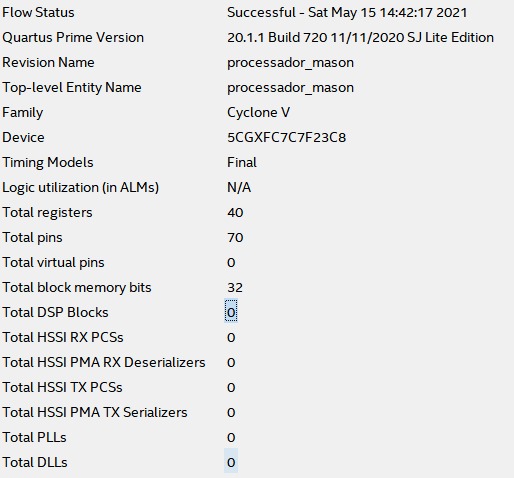


Figura 1 – Especificações no Quartus

## Conjunto de instruções

O processador **MASON** possui 4 registradores: S0, S1, S2 e S3. Assim como 3 formatos de instruções de 8 bits cada, Instruções do **tipo R, I e J**, 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**: o registrador contendo o primeiro operando fonte e adicionalmente para alguns tipos de instruções (ex. instruções do tipo R) é o registrador de destino;
* **Reg2**: o registrador contendo o segundo operando fonte;

Tipo de Instruções:

**- Formato do tipo R:** Este formatado aborda as instruções baseadas em operações aritméticas, condição para desvio, e subtração, soma e multiplicação

- **Formato do tipo I:** Este formato aborda as instruções baseadas em operações com valores imediatos e operações relacionadas à memória, soma e subtração imediata, Load e Store.

- **Formato do tipo J:** Este formato aborda de desvios condicionais e salto incondicionais exemplo o BEQ, BNE e Jump.

Formato para escrita em código binário:

|  |  |  |
| --- | --- | --- |
|  | Tipo R |  |
| OPCODE | REG1 | REG2 |
| 4 bits | 2 bits | 2 bits |
| 7 - 4 | 3 - 2 | 1 - 0 |

|  |  |  |
| --- | --- | --- |
|  | Tipo I |  |
| OPCODE | REG1 | IMEDIATO |
| 4 bits | 2 bits | 2 bits |
| 7 -4 | 3 - 2 | 1 - 0 |

|  |  |  |
| --- | --- | --- |
|  | Tipo J | |
| OPCODE | | ENDEREÇO |
| 4 bits | | 4 bits |
| 7 – 4 | | 3 - 0 |

**Visão geral das instruções do Processador MASON:**

O número de bits do campo **OpCode** das instruções é igual a quatro, sendo assim obtemos um total () de **16** **OpCodes (0-15)** que são distribuídos entre as instruções, assim como é apresentado na Tabela 1.

Tabela 1 – Tabela que mostra a lista de Opcodes utilizadas pelo processador MASON.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Opcode** | **Nome** | **Formato** | **Breve Descrição** | **Exemplo** |
| 0000 | ADICIONAR | R | Soma | **adicionar**$S0, $S1 ,ou seja, $S0 := $S0+$S1 |
| 0001 | SUBTRAIR | R | Subtração | **subtrair** $S0, $S1 ,ou seja, $S0 := $S0 - $S1 |
| 0010 | MULTIPLICAR | R | Multiplicação | **multiplicar** $s0 $s1, ou seja: $s0 := $s0 \* $s1 |
| 0011 | ADD\_IME | I | Soma Imediata | **add\_ime** $s0 3, ou seja: $s0 := $s0 + 3 |
| 0100 | SUB\_IME | I | Subtração Imediata | **sub\_ime** $s0 3, ou seja: $s0 := $s0 - 3 |
| 0101 | LOAD\_IME | I | Load Imediato | **load\_ime** $s0 1, ou seja: $s0 := 1 |
| 0110 | LOADW | I | Load Word | l**oadw** $s0 memória(00), ou seja: $s0 := valor memória(00) |
| 0111 | STOREW | I | Store Word | **storew** $s0 memória(00), ou seja: memória(00) := $s0 |
| 1000 | BEQ | J | Desvio Condicional | **beq** endereço, ou seja: if($s0 == $s1) |
| 1001 | BNE | J | Desvio Condiciomal | **bne** endereço, ou seja: if($s0 != $s1) |
| 1010 | IF | R | Condição para desvio | **if** $s0 $s1, ou seja: if($s0 == $s1) |
| 1011 | JUMP | J | Salto Incondiconal | **jump** endereço(0000) |

## Descrição do Hardware

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

### UAL

A UAL é responsável por executar as operações aritméticas, dentre elas: soma, subtração, multiplicação e logo a UAL também executa operações de comparação de valor para realizar desvios condicionais, a UAL tem 4 entradas, o Clock, ALUOP recebe qual operação ela vai realizar, um portINA e portINB onde vão ser recebidos os dois dados de 8 bits cada para realização das operações, uma OutUalResultado onde vai sair os 8 bits do resultado das operações, zero onde sairá se ocorrer um desvio condicional e a saída overflow que sai 1 se ocorrer um overflow durante a reprodução das operações.

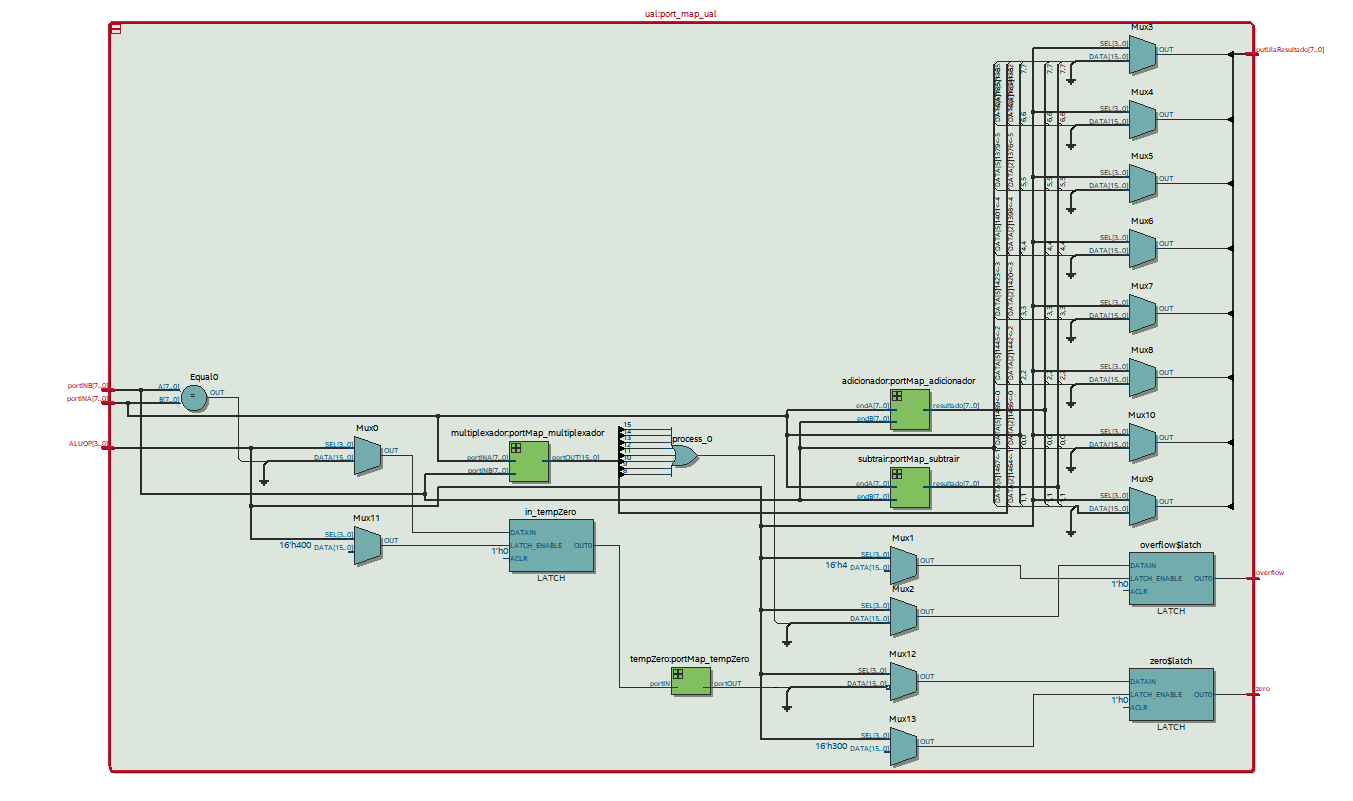


Figura 2 – RTL viewer do componente UAL

### Registradores

Os registradores são responsáveis por armazenar os dados que são usados para executar as operações. Nele, possuímos 4 registradores, que armazenam os valores de 8 bits, contém uma entrada, o Clock, que ativa o componente, REGWRITE que ativa a opção de escrever dados no registrador, writeData que recebe o valor de 8 bits do dado a ser escrito no registrador de destino, o enderecoRegA recebe 2 bits para o endereço do primeiro registrador e o enderecoRegB recebe 2 bits para o endereço do segundo registrador, regOutA resulta em uma saída de 8 bits do valor armazenado no enderecoRegA e o regOutB vai ter uma saída de 8 bits do valor armazenado no enderecoRegB.

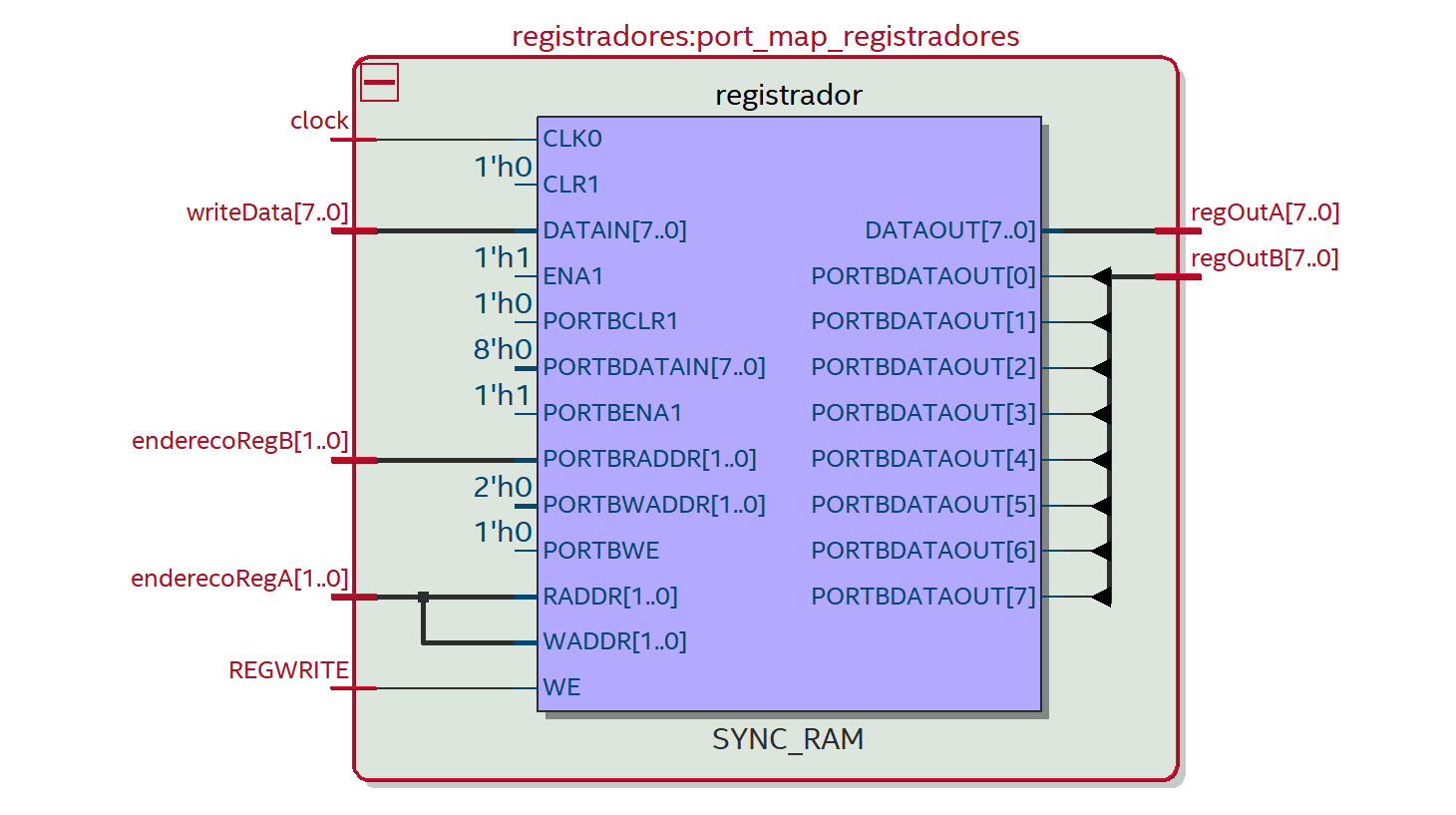


Figura 3 – RTL viewer do componente Registradores

### Extensor de Sinal

**Extensor de Sinal 2­­\_8:**

O extensor de sinal é responsável por estender o número de bits da entrada portIN de 2 bits para 8 bits para a saída portOUT, seu resultado é usado para operações na UAL e como endereço da memória RAM.

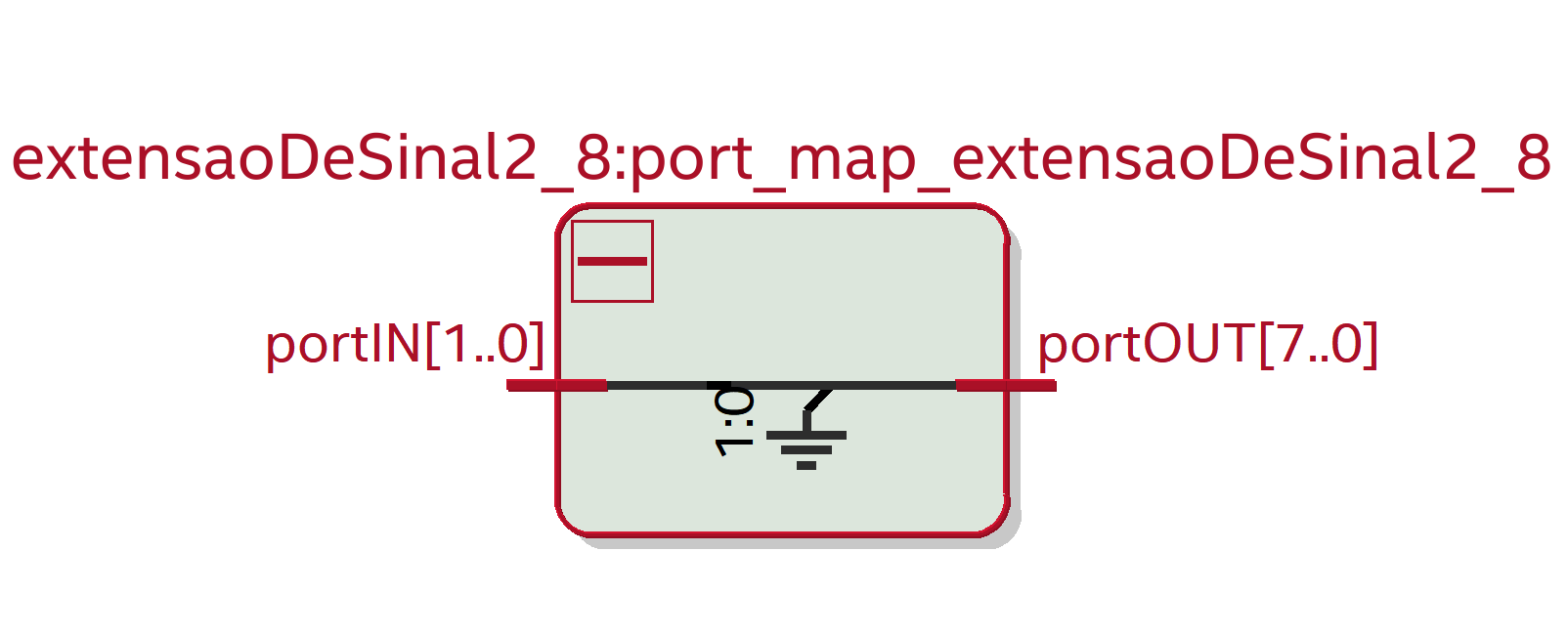


Figura 4 – RTL viewer do componente Extensor de Sinal 2\_8

**Extensor de Sinal 4­\_8:**

O extensor de sinal é responsável por estender o número de bits da entrada portIN de 4 bits para 8 bits para a saída portOUT, seu resultado é usado para desvios condicionais e saltos incondicionais.

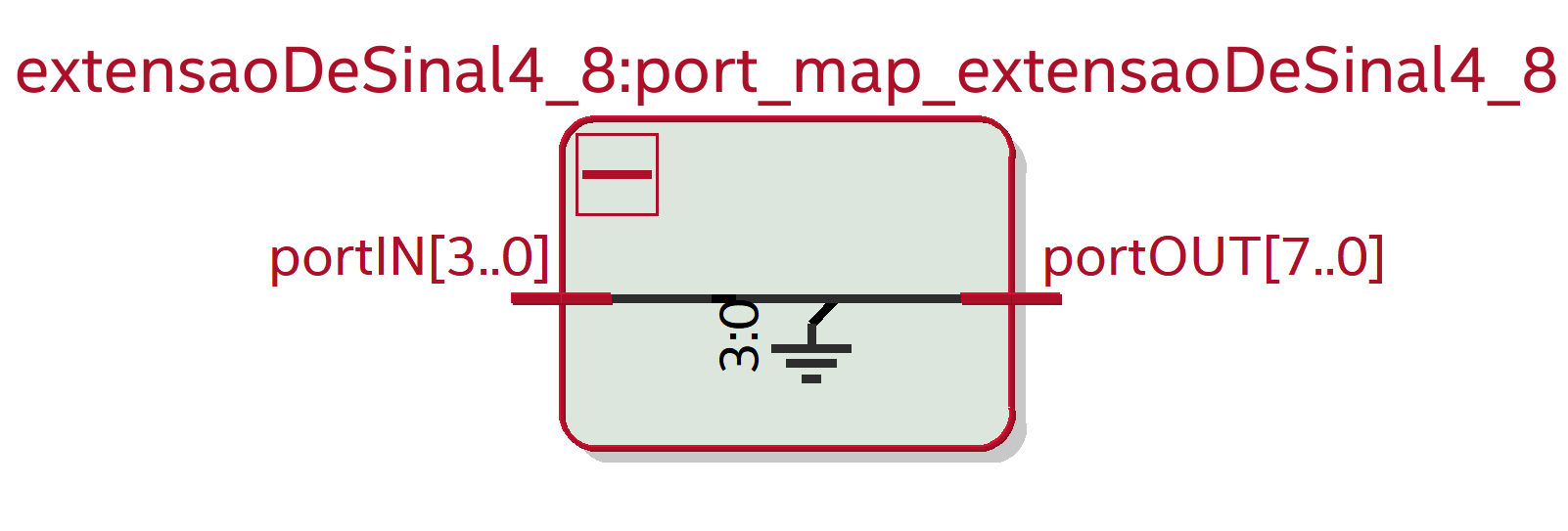
****

Figura 5 – RTL viewer do componente Extensor de Sinal 4\_8

### Controle

O componente Controle tem como objetivo realizar o controle de todos os componentes do processador de acordo com o OpCode 4 bits que é recebido no Opcode. Esse controle é feito através das flags de saída abaixo:

* **JUMP**: sinal que vai decidir se vai ocorrer um desvio incondicional.
* **BRANCH**: sinal para decidir se vai ocorrer um desvio condicional.
* **MEMREAD**: sinal para decidir se será lido um dado da memória RAM.
* **MEMTOREG**: sinal para decidir se o dado que será escrito no banco de registradores vai ser enviado pela ULA ou pela memória RAM.
* **UALOP**: sinal de 4 bits que será enviado para a ULA decidir qual operação será realizada.
* **MEMWRITE**: sinal para decidir se será escrito um dado da memória RAM
* **UALSRC:** sinal para decidir se o dado que entrará na ULA vai ser enviado pelo banco de registradores ou pelo extensor de sinal.
* **REGWRITE**: sinal para decidir se o banco de registradores vai escrever um dado na posição do registrador de destino.

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

Tabela 2 - Detalhes das flags de controle do processador.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Comando | JUMP | BRANCH | MEMREAD | MEMTOREG | UALOP | MEMWRITE | UALSRC | REGWRITE |
| ADICIONAR | 0 | 0 | 0 | 0 | 0000 | 0 | 0 | 1 |
| SUBTRAIR | 0 | 0 | 0 | 0 | 0001 | 0 | 0 | 1 |
| MULTIPLICAR | 0 | 0 | 0 | 0 | 0010 | 0 | 0 | 1 |
| ADD\_IME | 0 | 0 | 0 | 0 | 0011 | 0 | 1 | 1 |
| SUB\_IME | 0 | 0 | 0 | 0 | 0100 | 0 | 1 | 1 |
| LOAD\_IME | 0 | 0 | 0 | 0 | 0101 | 0 | 1 | 1 |
| LOADW | 0 | 0 | 1 | 1 | 0110 | 0 | 0 | 1 |
| STOREW | 0 | 0 | 0 | 0 | 0111 | 1 | 0 | 0 |
| BEQ | 0 | 1 | 0 | 0 | 1000 | 0 | 0 | 0 |
| BNE | 0 | 1 | 0 | 0 | 1001 | 0 | 0 | 0 |
| IF | 0 | 0 | 0 | 0 | 1010 | 0 | 0 | 0 |
| JUMP | 1 | 0 | 0 | 0 | 1011 | 0 | 0 | 0 |

### memoriaAdress

A memoriaAdress, ou memória RAM, é responsável por armazenar por tempo limitado os dados que são usados durante a execução das instruções, possui 5 entradas, o Clock que ativa o componente, portIN recebe o dado de 8 bits que será armazenado temporariamente na memória RAM, MEMWRITE recebe 1 bit para saber se vai armazenar dados na memória RAM, MEMREAD recebe 1 bit para saber se será lido algum dos dados na memória RAM, endereco recebe 8 bits da posição na memória RAM onde o dado deve ser lido ou escrito e possui 1 saida, portOUT onde sai um dado de 8 bits da posição que foi recebida no endereço, se a MEMREAD estiver setada em 1.

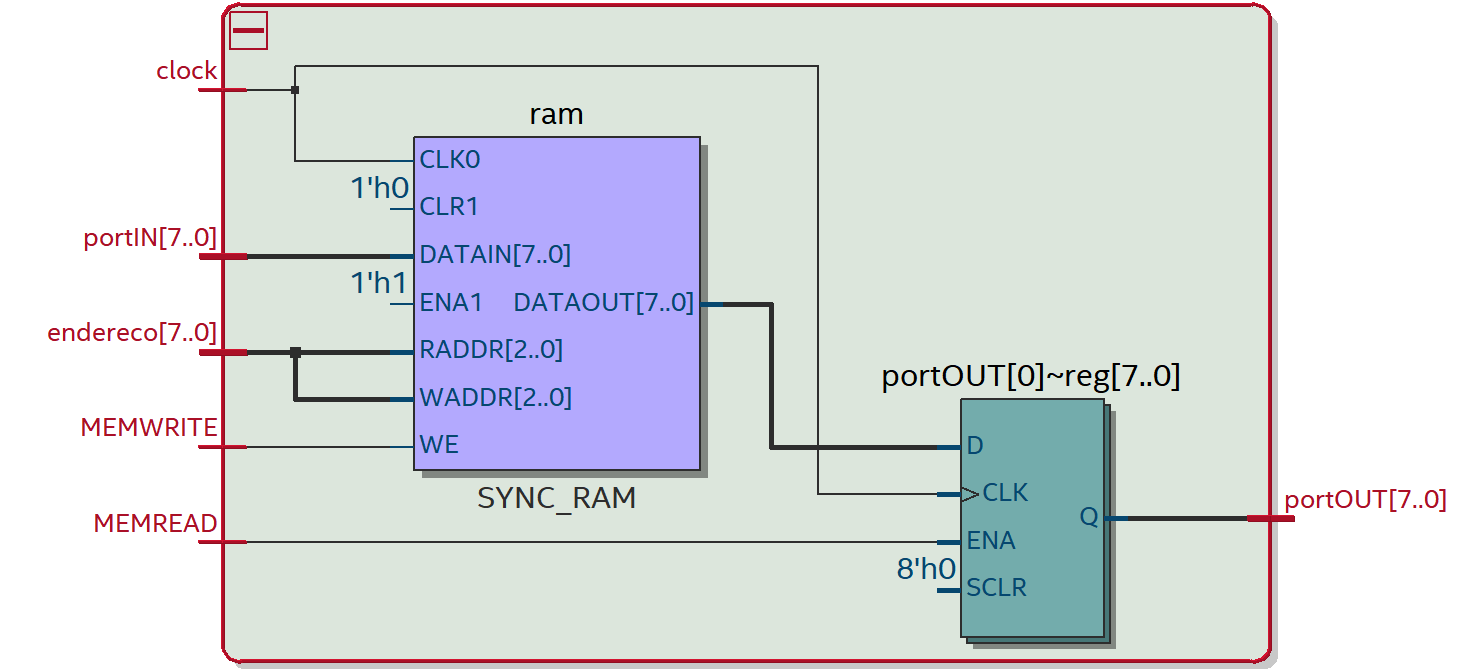


Figura 6 – RTL viewer de componente memoriaAdress

### memoriaInst

A memória de Instrução, ou memoriaInst, tem como função o armazenamento das instruções que vão ser executadas pelo processador, possui duas entradas, o Clock e o portIN, que é o endereço de 8 bits da instrução que será enviada para a execução, possui uma saída portOUT de 8 bits da instrução que será executada.

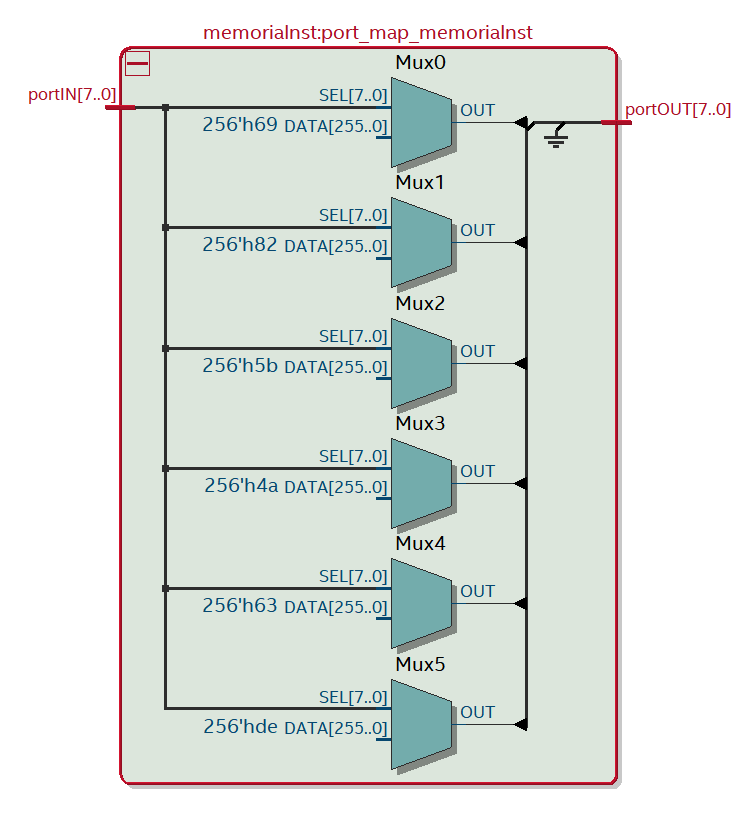


Figura 7 – RTL viewer do componente memoriaInst

### addPC

O addPC é responsável por adicionar o valor da instrução atual no PC contendo o valor binário de 1, ele contém uma entrada, portIN, recebendo o endereço de 8 bits da instrução atual do PC e tem uma saída portOUT, que receberá o resultado da soma do endereço atual com um valor de 1 e assim gerando o próximo endereço de instrução, que será armazenado no PC.

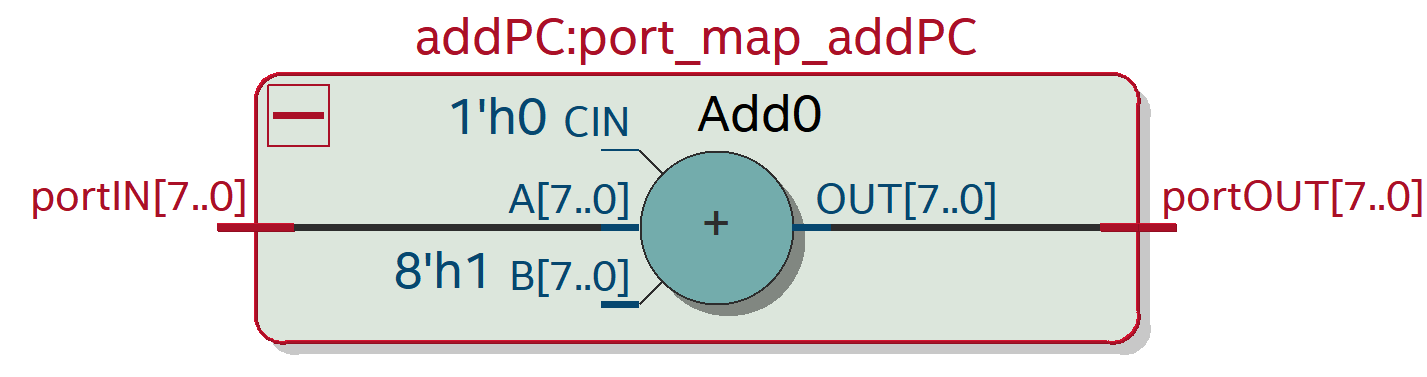


Figura 8 – RTL viewer do componente addPC

### portAND

O portAND tem como a sua função de decidir se vai acontecer um desvio condicional, ele tem duas entradas, in\_port\_A e in\_port\_B se as duas estiverem recebendo 1 o mesmo sairá 1 no out\_port e com isso ele passa o valor 1 para o multiplexador fazendo com que ocorra um desvio condicional.

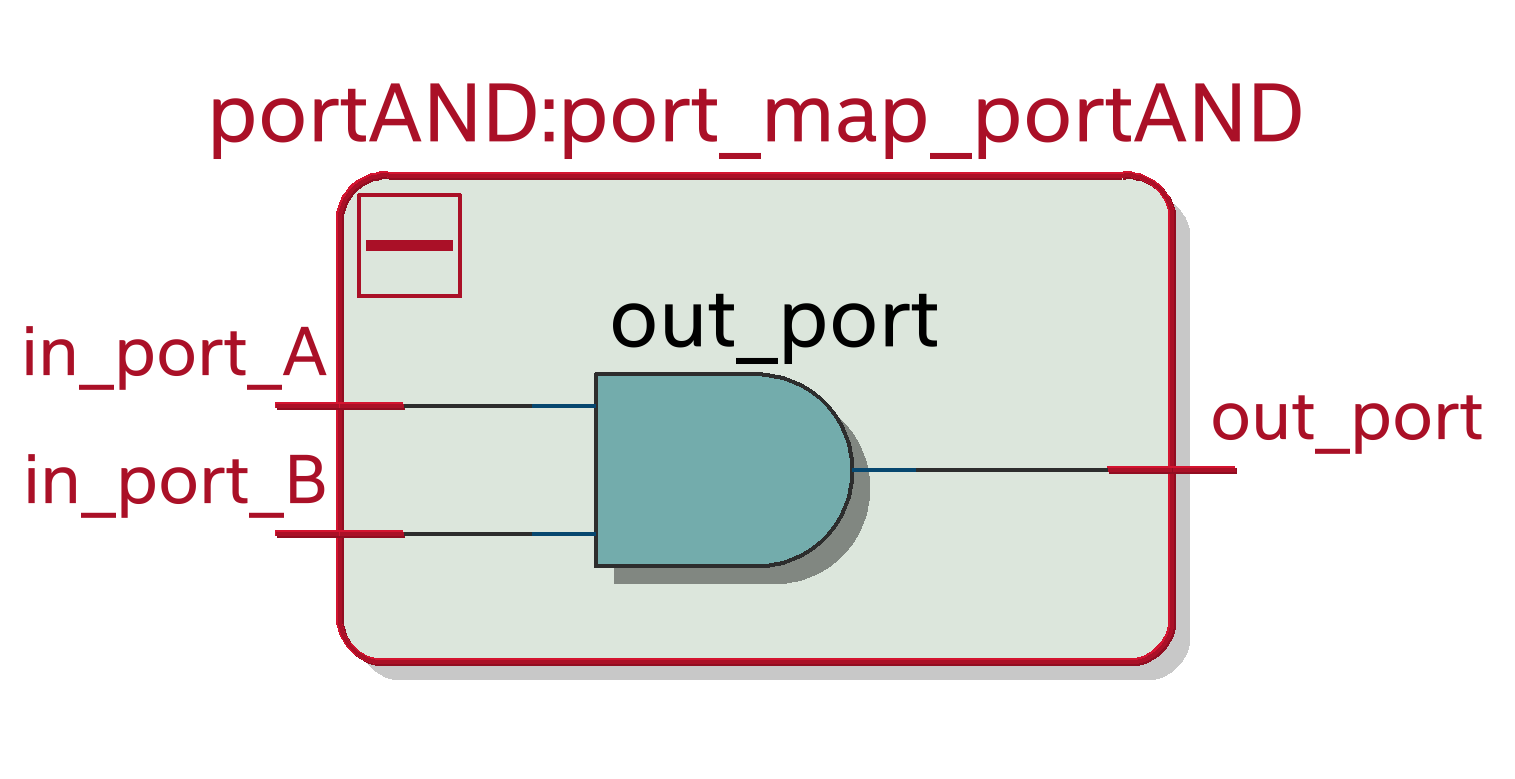


Figura 9 – RTL viewer do componente portAND

### Muxtiplexador\_2x1\_8bits

O multiplexador tem duas entradas de 8 bits, inA e inB que recebe os dados e uma entrada portIN que decidirá qual dado vai sair, caso essa entrada for 0 o dado que sairá na portOUT será o dado inA se for 1 sairá o dado inB.

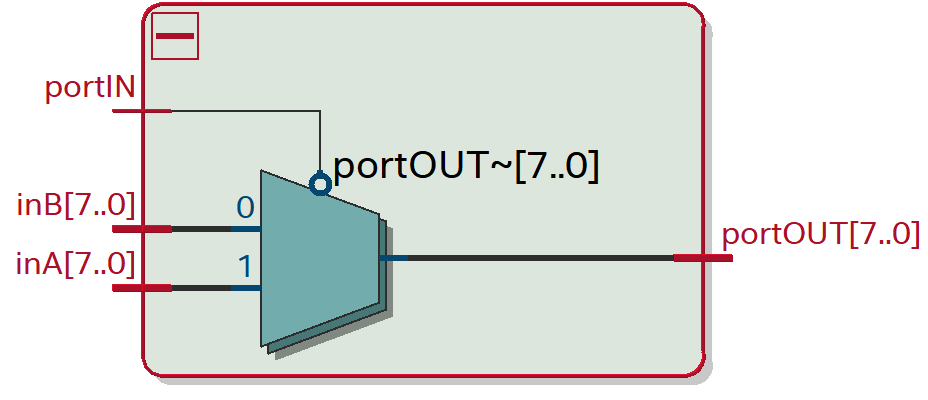


Figura 10 – RTL viewer do componente Multiplexador\_2x1\_8bits

### PC

A função do componente do PC é armazenar o endereço de 8 bits da instrução. Existem dois valores de entrada que serão executados, o clock, que é responsável por ativar o componente, o portIN, onde entra o endereço que vai ser executado, e a saída no portOUT, onde o endereço da instrução é realizada.

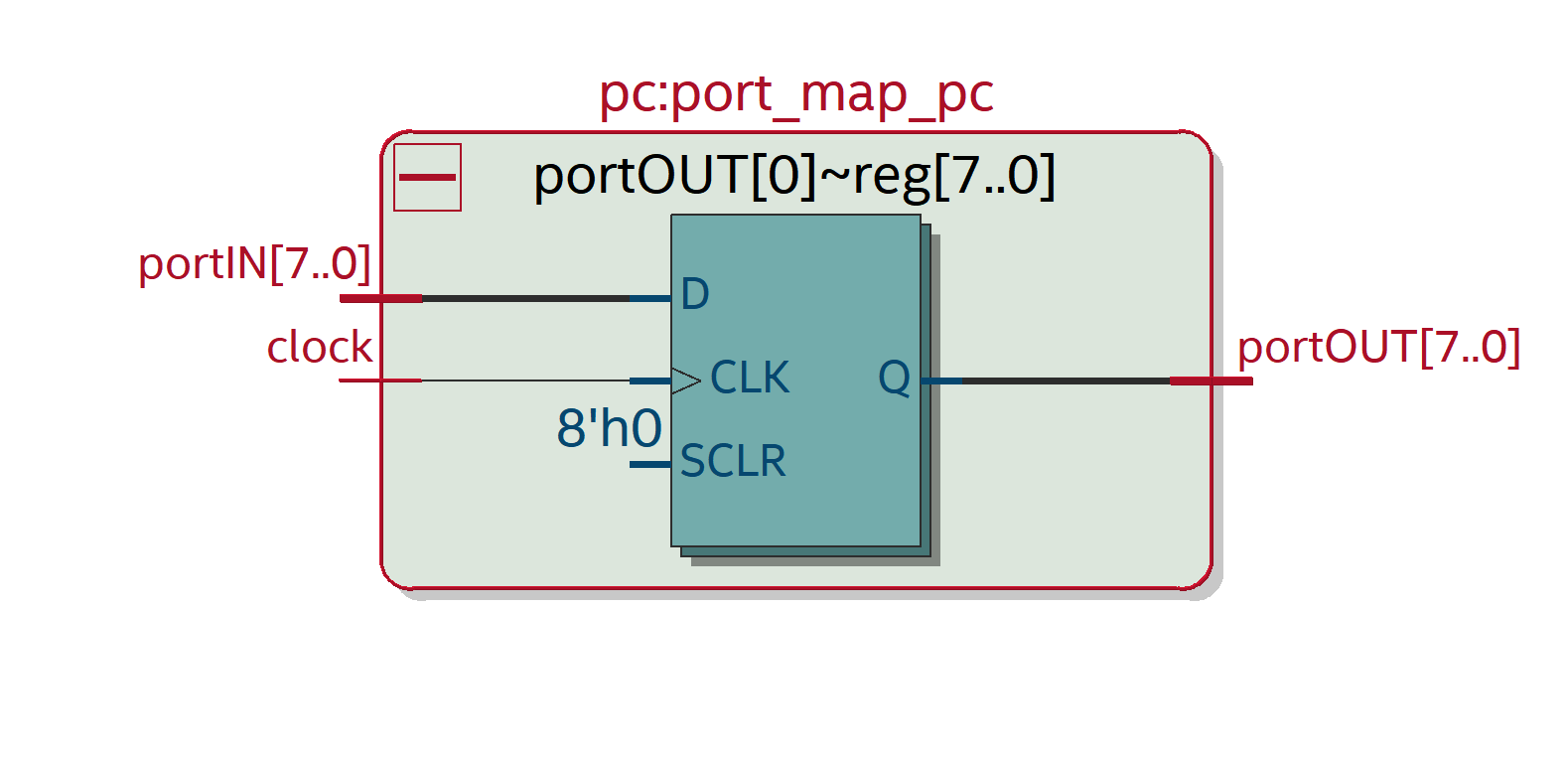


Figura 11 – RTL viewer do componente PC

### divisaoDeInstr

A divisaoDeInstr é responsável por dividir os 8 bits que recebe da memória de dados no portIN, o outOpCode recebe os bits que entrarão na Unidade de Controle, outJump recebe os últimos 4 bits da instrução que são usados para o desvio condicional e o salto incondicional, outRS recebe os 2 bits do registrador onde será armazenado o resultado das operações, outRT pode ser tanto o dado do registrador ou o endereço da memória de dados para realizar load e store.

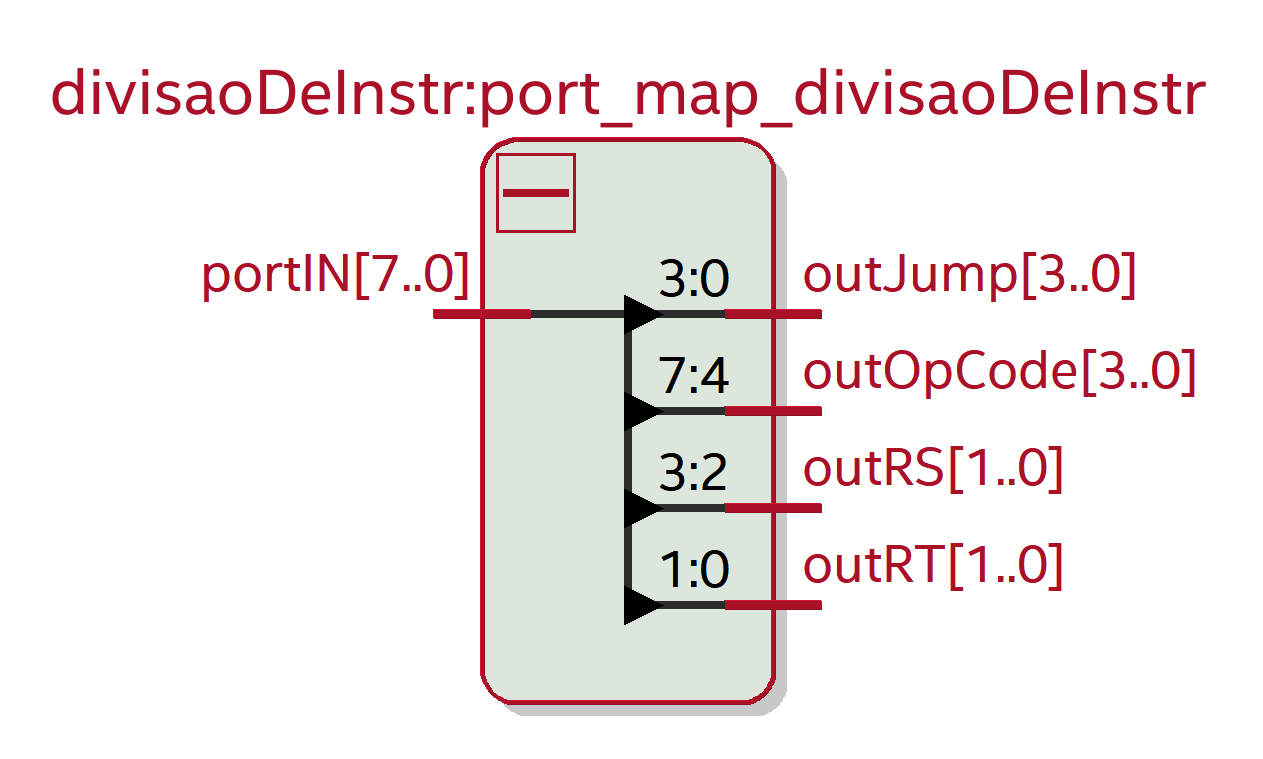
****

Figura 12 – RTL viewer do componente divisaoDeInstr

## 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.

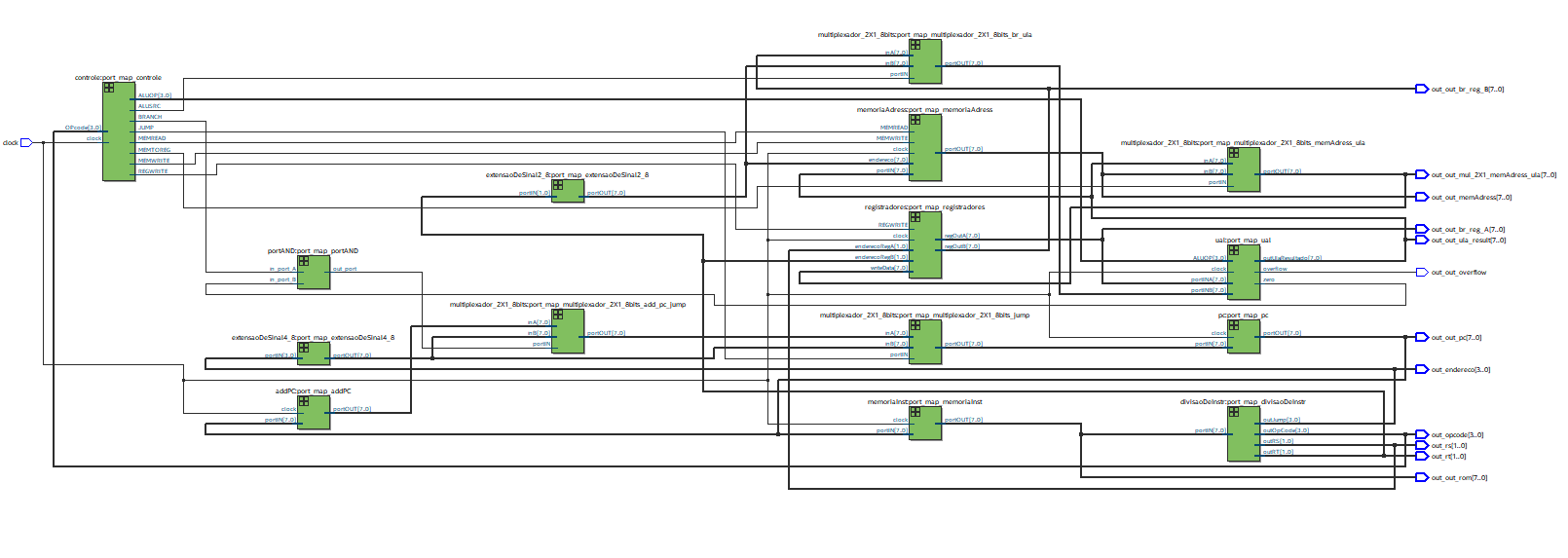


Figura 13 – RTL viewer do processador MASON

# Simulações e Testes

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

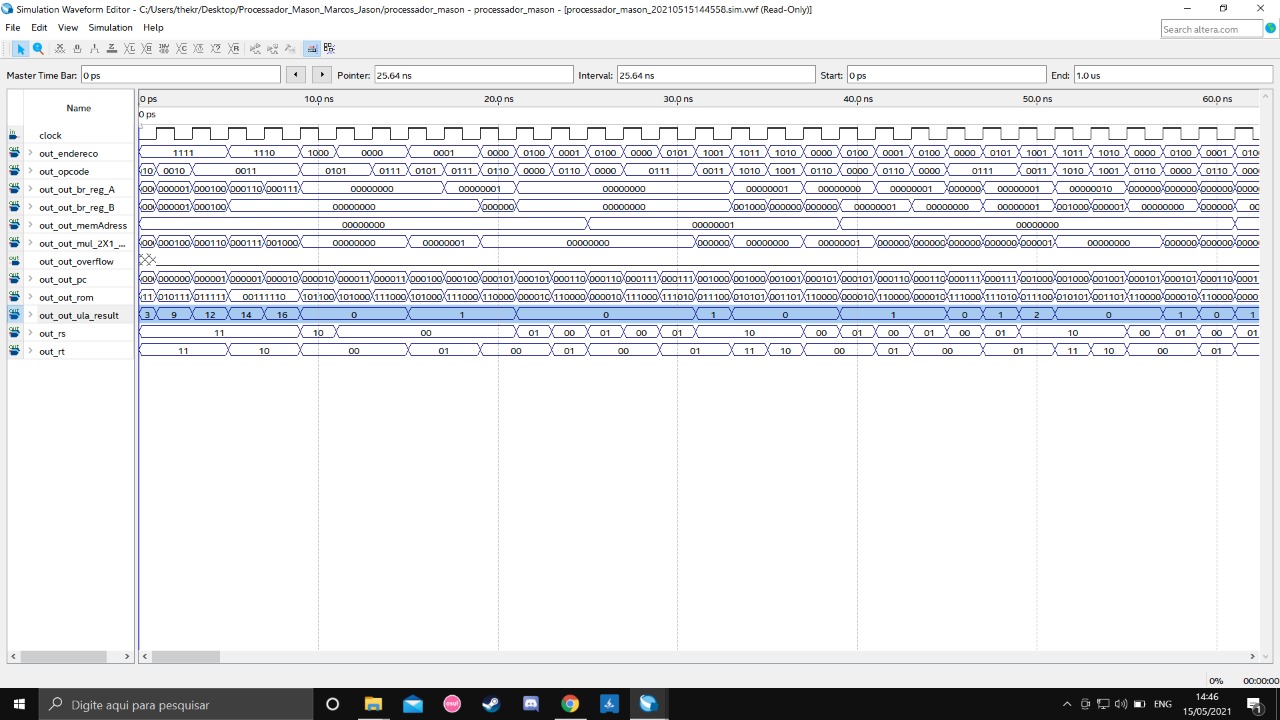
Tabela 3 - Código Fibonacci para o processador MASON.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Endereço** |  | **Binário** | | | |
| **Linguagem de Alto Nível** | Opcode | Reg1 | Reg2 | |
|  | Endereço | | |
|  |  | | | |
| 0 | **load\_ime** S3 3 | 0101 | 11 | 11 | |
| 1 | **multiplicar** S3 S3 | 0010 | 11 | 11 | |
| 2 | **add\_ime** S3 1 | 0011 | 11 | 01 | |
| 3 | **add\_ime** S3 2 | 0011 | 11 | 10 | |
| 4 | **add\_ime** S3 2 | 0111 | 11 | 10 | |
| 5 | **load\_ime** S2 0 | 0101 | 10 | 00 | |
| 6 | **load\_ime** S0 0 | 0101 | 00 | 00 | |
| 7 | **storew** S0 ram(00) | 0111 | 00 | 00 | |
| 8 | **load\_ime S**0 1 | 0101 | 00 | 01 | |
| 9 | **storew** S0 ram(01) | 0111 | 00 | 01 | |
| 10 | **loadw** S0 ram(00) | 0110 | 00 | 00 | |
| 11 | **adicionar** S1 S0 | 0000 | 01 | 00 | |
| 12 | **loadw** S0 ram(01) | 0110 | 00 | 01 | |
| 13 | **adicionar S**1 S0 | 0000 | 01 | 00 | |
| 14 | **storew** S0 ram (00) | 0111 | 00 | 00 | |
| 15 | **storew** S1 ram(01) | 0111 | 01 | 01 | |
| 16 | **add\_ime** S2 1 | 0011 | 10 | 01 | |
| 17 | **If** S2== S3 | 1010 | 10 | 11 | |
| 18 | **bne** S2 !=S3 | 1001 | 1010 | |

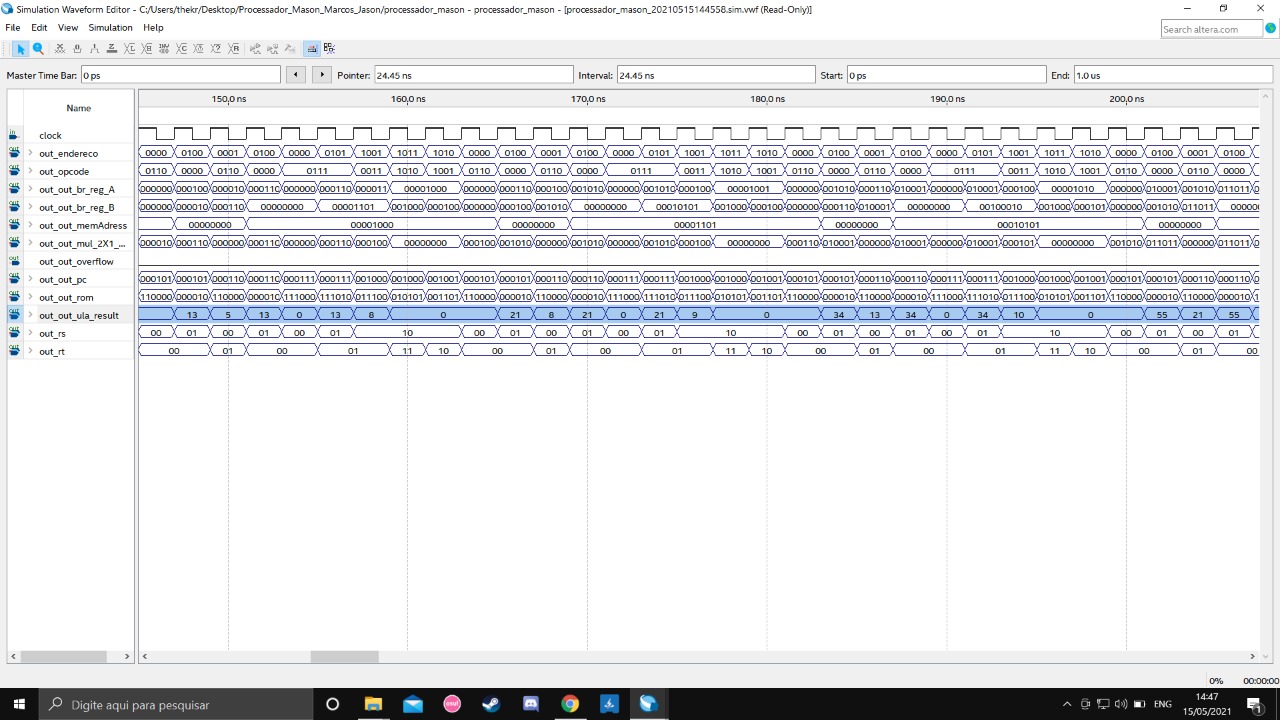
Conforme podemos ver na Tabela 3 o código para a implementar o fibonacci, o endereço 0 ao endereço 9 são preparações para executar o fibonacci, no endereço 10 é carregado o valor da posição 0 da memória RAM para o registrador S0 e já no endereço 11 é feito a soma entre os registradores S1 e S0, enquanto no endereço 12 é carregado o valor do registrador S0 na posição 01 da memória RAM, a soma entre os registradores S1 e S0 é feita no endereço 13, no endereço 14 é carregado o valor do registrador S0 na posição 00 da memória RAM, no endereço 15 é carregado o valor do registrador S1 na posição 01 da

memória RAM, no endereço 16 é feito uma soma imediata do valor 1 no registrador S2, para ser usado no loop, no endereço 17 é feito a comparação entre os registradores S2 e S3, no endereço 18 está a instrução bne, se o valor do registrador S2 for diferente do valor no registrador S3, ou seja, enquanto o valor do S2 for menor que o valor do S3, o código salta para a posição 10, se os valores forem iguais encerra a execução do programa.

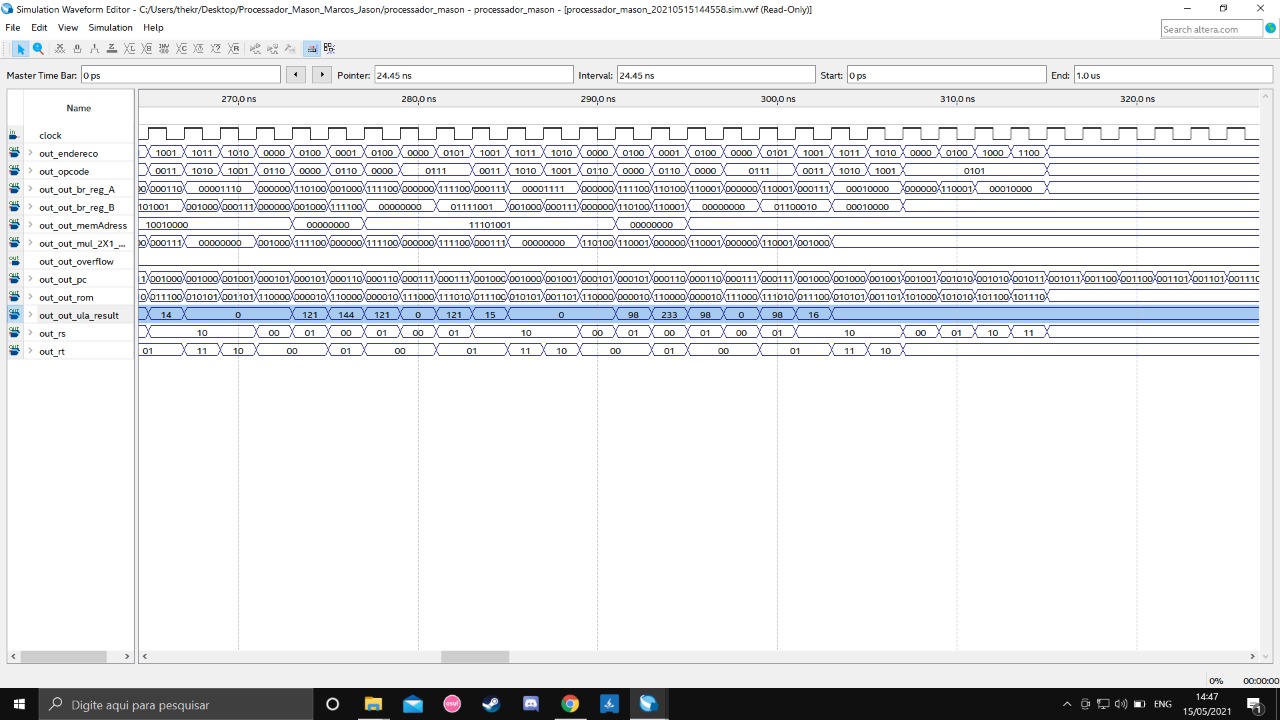
**Verificação dos resultados no relatório da simulação:** Após a compilação e execução da simulação, o seguinte relatório é exibido.



Podemos ver por aqui a saída do Fibonacci da UAL. No começo dela vemos que é carregado vários valores até chegar o 16 para calcular o Fibonacci.



Aqui podemos ver a execução do Fibonacci



Aqui chegamos ao fim do Fibonacci, ocorrendo um overflow logo após tentar calcular até 16

# Considerações finais

Este trabalho apresentou o projeto e implementação do processador de 8 bits denominado de **MASON.** O processo de criação foi difícil e complicado de início, mas logo esses problemas foram enfrentados e vencidos com muito esforço e dedicação. Vale ressaltar que, o processador consegue executar operações básicas, por isso hoje ele pode ser um processador simples, mas ele é funcional e atende os requisitos que foram pedidos.

Porém devemos também ressaltar a limitação do processador onde o Load e Store funcionam apenas em uma instrução devido a limitação de 8 bits, somente dois bits são usados para acessar a memória RAM, o que nos limita a poder acessar somente as 4 primeiras posições da memória RAM. Outra situação encontrada que seria uma pequena limitação, é no desvio condicional e o salto incondicional, onde ele apenas salta entre os endereços de 0000 e 1111, os 16 primeiros endereços da memória ROM, e a última limitação que foi encontrada está na flag do overflow, que pode ter resultados falsos ao realizar operações com números negativos.

Apesar de tudo, foi um processo interessante e inovador e com muito aprendizado. Nós, Jasson e Marcos, aprendemos muitas coisas durante a criação do processador e com certeza serão informações muito valiosas no futuro.

# Referências Bibliográficas

LAMERES, Brock J. **Introduction to Logic Circuits e Logic Design With VHDL**, 2. ed. Bozeman, MT, USA: Springer, 2017, 485 p.

PATTERSON, D.; HENESSY, J. L. **Organização e projeto de computadores: a interface hardware/software**. 3ª Edição. São Paulo: Elsevier, 2005, 484 p