

**PODER EXECUTIVO**

**MINISTÉRIO DA EDUCAÇÃO**

**UNIVERSIDADE FEDERAL DE RORAIMA**

**DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO**

**ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES**

**RELATÓRIO DO PROJETO: PROCESSADOR GDZ**

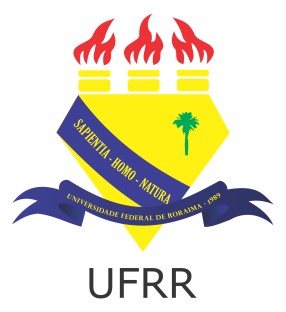
**ALUNOS:**

**Pedro Vinícius da Silva Ribeiro – 2019033903**

**Ozéias Souza - 2019013277**

**Maio de 2021**

**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 GDZ**

**Maio de 2021**

**Boa Vista/Roraima**

**Resumo**

Este projeto tem como objetivo de aprendizado, o desenvolvimento e prototipação de processadores utilizando dispositivos lógicos programáveis, para a implementação de um processador RISC (Reduced Instruction Set Computer), utilizando a linguagem VHDL (VeryHigh-Speed Integrated Circuit Hardware Description Language).

O estudo envolve o conhecimento da arquitetura RISC e da linguagem VHDL para o desenvolvimento do processador com uma arquitetura de 8 bits como critério para avaliação e obtenção de nota na disciplina de Arquitetura e Organização de Computadores, ministrada pelo professor Herbert Oliveira Rocha.

O projeto do processador GDZ foi feito pela plataforma Intel Quartus que é um software de design de dispositivo lógico programável produzido pela Intel, além do uso da linguagem Assembly MIPS.

**Conteúdo**

[1 Especificação 7](#_Toc72079820)

[1.1 Plataforma de desenvolvimento 7](#_Toc72079821)

[1.2 Conjunto de instruções 8](#_Toc72079822)

[1.3 Tipo de Instruções: 8](#_Toc72079823)

[1.4 Descrição do Hardware 9](#_Toc72079824)

[1.4.1 ALU ou ULA 10](#_Toc72079825)

[1.4.2 Banco de Registradores 11](#_Toc72079826)

[1.4.3 Controle 11](#_Toc72079827)

[1.4.4 Memória de dados 13](#_Toc72079828)

[1.4.5 Memória de Instruções 14](#_Toc72079829)

[1.4.6 Somador 15](#_Toc72079830)

[1.4.7 And 15](#_Toc72079831)

[1.4.8 Mult\_2x1 16](#_Toc72079832)

[1.4.9 PC 16](#_Toc72079833)

[1.4.10 Extensor 17](#_Toc72079834)

[2. Datapath 18](#_Toc72079835)

[3. Simulações e Testes 19](#_Toc72079836)

[3.1 Fatorial 19](#_Toc72079837)

[3.1.1 Testes do Fatorial 20](#_Toc72079838)

[3.2 Fibonacci 20](#_Toc72079839)

[3.2.1 Testes Fibonacci 22](#_Toc72079840)

[4. Considerações finais 23](#_Toc72079841)

**Lista de Figuras**

[Figura 1 - Especificações no Quartus 8](#_Toc72082692)

[Figura 2 - Bloco simbólico do componente ALU gerado pelo Quartus 11](#_Toc72082693)

[Figura 3 - Bloco simbólico do componente banco\_reg gerado pelo Quartus 12](#_Toc72082694)

[Figura 4 - Bloco simbólico do componente unidade\_de\_controle gerado pelo Quartus 13](#_Toc72082695)

[Figura 5 - Bloco simbólico do componente ram gerado pelo Quartus 14](#_Toc72082696)

[Figura 6 - Bloco simbólico do componente ram gerado pelo Quartus 15](#_Toc72082697)

[Figura 7 - Bloco simbólico do componente somador8bits gerado pelo Quartus 16](#_Toc72082698)

[Figura 8 - Bloco simbólico do componente p\_and gerado pelo Quartus 16](#_Toc72082699)

[Figura 9 - Bloco simbólico do componente mult\_2x1 gerado pelo Quartus 17](#_Toc72082700)

[Figura 10 - Bloco simbólico do componente pc gerado pelo Quartus 17](#_Toc72082701)

[Figura 11 - Bloco simbólico do componente Extensor gerado pelo Quartus 18](#_Toc72082702)

[Figura 12 - Datapath gerado pelo Quartus do processador GDZ 19](#_Toc72082703)

[Figura 13 - Resultado na waveform. 21](#_Toc72082704)

**Lista de Tabelas**

[Tabela 1 - Tabela que mostra a lista de Opcodes utilizadas pelo processador GDZ. 9](#_Toc72082174)

[Tabela 2 - Detalhes das flags de controle do processador. 12](#_Toc72082175)

[Tabela 3 - Detalhes das flags de controle do processador. 12](#_Toc72082176)

[Tabela 4 - Código Fatorial para o processador GDZ/EXEMPLO. 19](#_Toc72082177)

[Tabela 5 - Código Fibonacci para o processador GDZ/EXEMPLO. 21](#_Toc72082178)

# Especificação

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

## Plataforma de desenvolvimento

Para a implementação do processador GDZfoi utilizado a IDE:

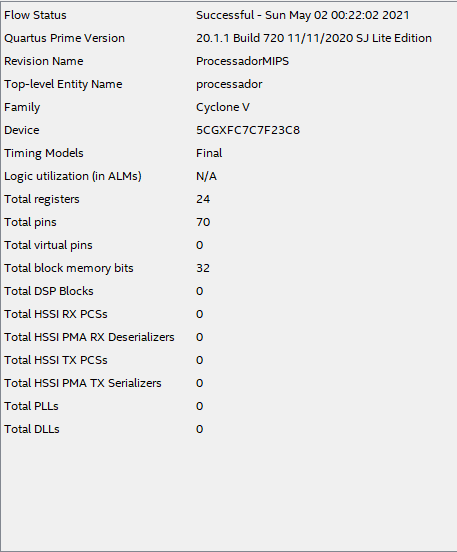


Figura 1 - Especificações no Quartus

## Conjunto de instruções

O processador **GDZ** possui 2 registradores: $s0 e $s1. Assim como 11 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($s0)**: o registrador contendo o primeiro operando fonte e adicionalmente para alguns tipos de instruções (ex. instruções do tipo R) é o registrador de destino;
* **Reg2($s1)**: o registrador contendo o segundo operando fonte;

## Tipo de Instruções:

* Formato do tipo R: Este formato aborda instruções baseadas em operações aritméticas.
* Formato para escrita de código na linguagem GDZ:

|  |  |  |
| --- | --- | --- |
| Tipo da Instrução | $s0 | $s1 |

* Formato para escrita em código binário:

|  |  |  |
| --- | --- | --- |
| Instrução do tipo R | | |
| OPCODE | **rs** | **rt** |
| 4bits | 2bits | 2bits |
| 7-4 | 3-2 | 1-0 |

* Formato do tipo I: Este formato aborda instruções baseadas carregamentos imediatos na memória.
* Formato para escrita de código na linguagem GDZ:

|  |  |
| --- | --- |
| Tipo da Instrução | $s0 |

* Formato para escrita em código binário:

|  |  |  |
| --- | --- | --- |
| Instrução do tipo I | | |
| OPCODE | **rs** | **Imediato** |
| 4bits | 2bits | 2bits |
| 7-4 | 3-2 | 1-0 |

* Formato do tipo J: Este formato aborda instruções baseadas desvios condicionais e incondicionais.
* Formato para escrita de código na linguagem GDZ:

|  |  |
| --- | --- |
| Tipo da Instrução | $s0 |

* Formato para escrita em código binário:

|  |  |
| --- | --- |
| Instrução do tipo J | |
| OPCODE | **Endereço** |
| 4bits | 4bits |
| 7-4 | 3-0 |

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

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.

|  |  |  |  |
| --- | --- | --- | --- |
| OPCODES | Significado | Sintaxe | Exemplos |
| 0000 | Soma | add | add $s0, $s1 |
| 0001 | Soma Imediata | addi | addi $s0, $s1 |
| 0010 | Subtração | sub | sub $s0, $s1 |
| 0011 | Subtração Imediata | subi | subi $s0, $s1 |
| 0100 | Multiplicação | mul | mul $s0, $s1 |
| 0101 | Load Word | lw | lw $s0 **memoria (00)** |
| 0110 | Store Word | sw | sw $s0 **memoria (00)** |
| 0111 | Load Imediato | li | li $s0 2 |
| 1000 | Desvio Condicional | beq | beq **endereço** |
| 1001 | Desvio Condicional | bne | bne **endereço** |
| 1010 | Desvio Incondicional | j | J **endereço** (0000) |
| 1011 | Desvio Condicional (IF) | if | if **endereço** (0000) |

Tabela 1 - Tabela que mostra a lista de Opcodes utilizadas pelo processador GDZ.

## Descrição do Hardware

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

### ALU ou ULA

O componente ALU (Unidade Lógica Aritmética) tem como principal objetivo efetuar as principais operações aritméticas (considerando apenas resultados inteiros), dentre elas:

* Soma;
* Subtração;
* Multiplicação.

Adicionalmente o ALU efetua operações de comparação de valor como igual ou diferente. O componente ALU recebe como entrada três valores:

* **in\_A:** dado de 8bits para operação;
* **in\_B:** dado de 8bits para operação;
* **in\_op:** identificador da operação que será realizada de 4bits.

O ALU também possui três saídas:

* **z\_out:** identificador de resultado (1 bit) para comparações (1 se verdade e 0 caso contrário);
* **ula\_overflow:** identificador de overflow caso a operação exceda os 8bits;
* **S\_out:** saída com o resultado das operações aritméticas.

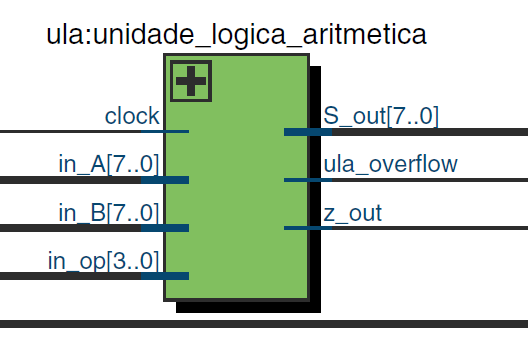


Figura 2 - Bloco simbólico do componente ALU gerado pelo Quartus

### Banco de Registradores

O componente banco\_reg tem como principal objetivo armazenar e dizer aos registradores os valores que eles receberão além de instruir seu destino dentro do barramento.

O componente banco\_reg recebe como entrada 4 valores:

* in\_reg\_A: valor imediato da instrução de 2 bits;
* in\_reg\_B: valor imediato da instrução de 2 bits;
* reg\_write: flag de controle para escrita no registrador;
* write\_data: dado de 8 bits a ser armazenado.

O banco\_reg também possui 2 saídas de 8 bits cada:

* Out\_reg\_A: dado lido pela entrada;
* Out\_reg\_B: dado lido pela entrada;

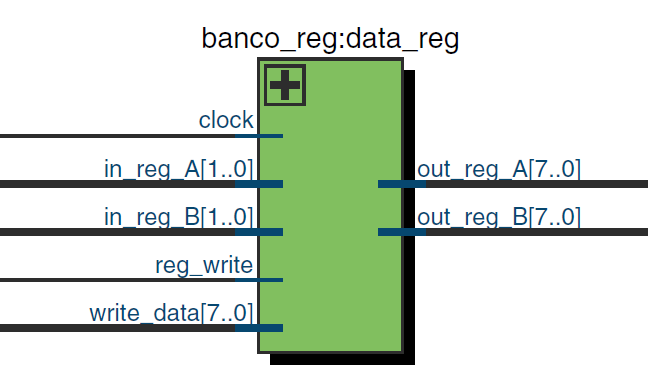
****

Figura 3 - Bloco simbólico do componente banco\_reg gerado pelo Quartus

### Controle

O componente unidade\_de\_controle é o responsável por dizer o que acontece e como se desenvolve dentro processador, portanto, tem como objetivo realizar o controle de todos os componentes do processador de acordo com o opcode, esse controle é feito através das flags de saída abaixo:

* **Jump**: Flag que é utilizada para operação de pulo de memória/desvio condicional.
* **Branch**: É a flag responsável pela decisão de haver ou não um desvio.
* **memRead**: É a flag que decide se um valor será lido ou não na memória RAM.
* **ALUop**: É a flag responsável por mandar o comando de qual operação a ALU irá executar.
* **memWrite**: É a flag responsável pela decisão de escrita na memória RAM.
* **AluSrc:** É o seletor que decide se o valor que a ALU irá receber vem do banco de registradores ou uma inserção imediata.
* **RegWrite**: É o que decide se haverá escrita ou não no banco de registradores.

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| opcode | Jump | Branch | memRead | memToreg | ALUop |
| 0000 | 0 | 0 | 0 | 0 | 0 |
| 0001 | 0 | 0 | 0 | 0 | 0 |
| 0010 | 0 | 0 | 0 | 0 | 0 |
| 0011 | 0 | 0 | 0 | 0 | 0 |
| 0100 | 0 | 0 | 0 | 0 | 0 |
| 0101 | 0 | 0 | 1 | 1 | 0 |
| 0110 | 0 | 0 | 0 | 0 | 0 |
| 0111 | 0 | 0 | 0 | 0 | 0 |
| 1000 | 0 | 1 | 0 | 0 | 0 |
| 1001 | 0 | 1 | 0 | 0 | 0 |
| 1010 | 1 | 0 | 0 | 0 | 0 |

Tabela 2 - Detalhes das flags de controle do processador.

|  |  |  |  |
| --- | --- | --- | --- |
| opcode | memWrite | AluSrc | RegWrite |
| 0000 | 0 | 0 | 1 |
| 0001 | 0 | 1 | 1 |
| 0010 | 0 | 0 | 1 |
| 0011 | 0 | 1 | 1 |
| 0100 | 0 | 0 | 1 |
| 0101 | 0 | 0 | 1 |
| 0110 | 1 | 0 | 0 |
| 0111 | 0 | 1 | 1 |
| 1000 | 0 | 0 | 0 |
| 1001 | 0 | 0 | 0 |
| 1010 | 0 | 0 | 0 |

Tabela 3 - Detalhes das flags de controle do processador.

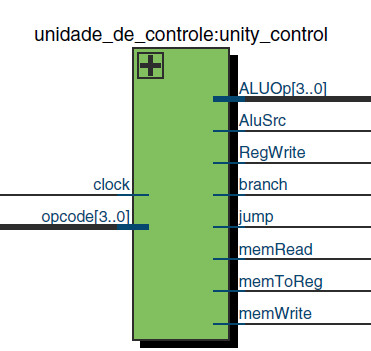
****

Figura 4 - Bloco simbólico do componente unidade\_de\_controle gerado pelo Quartus

### Memória de dados

O componente ram, também conhecido como memória de dados é o responsável por armazenar bits de memória, portanto temos uma matriz de 8x8, ou seja, 8 posições com a capacidade de armazenar 8bits dinamicamente durante a execução do barramento, possui 5 entradas:

* clock: sinal podendo ser de nivel logico alto ou baixo.
* in\_A: valor de 8 bits representando a entrada
* mem\_write: quando ativa, seta a memória ram para modo de escrita (8bits)
* mem\_read: quando ativa, seta a memória ram para leitura, ou seja, a saída terá o valor endereçado pela entrada in\_A (1bit)
* addr: endereço de trabalho (1bit) , e uma única saída:
* S\_out : saida de 8bits

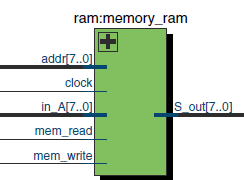
****

Figura 5 - Bloco simbólico do componente ram gerado pelo Quartus

### Memória de Instruções

O componente memoria\_de\_instrução também conhecido como memória ROM é o responsável por dividir a instrução que sai dele em 4 trilhas, portanto possui uma porta de entrada de 8bits:

* in\_port: porta que recebe a instrução que vem da ROM;

E 4 portas de saída:

* out\_jump: o endereço de memória que será usado na instrução do tipo jump e possui 4 bits, caso seja usado, utiliza os bits dos 2 registradores que não serão usados;
* out\_op\_code: o endereço de memória que será usado para o opcode do comando, possui 4 bits;
* out\_rs: o endereço do primeiro registrador, possui 2bits;
* out\_ts: o endereço do segundo registrador, possui 2bits;

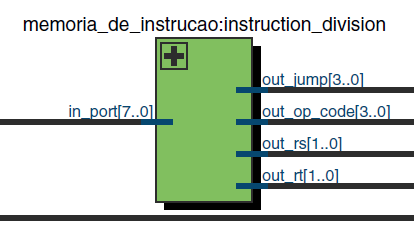
****

Figura 6 - Bloco simbólico do componente ram gerado pelo Quartus

### Somador

O componente somador8bits tem como principal objetivo efetuar operações de soma. É composto por 2 portas de entrada e uma porta de saída.

As portas de entrada são:

* **A:** recebe a informação de valor do primeiro registrador e possui 8 bits de tamanho;
* **B:** recebe a informação de valor do segundo registrador e possui 8 bits de tamanho, e porta de saída é:
* **S:** recebe o valor da operação de soma efetuada e possui também 8 bits de tamanho;

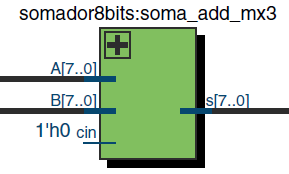
****

Figura 7 - Bloco simbólico do componente somador8bits gerado pelo Quartus

### And

O componente AND é o responsável por realizar a operação lógica “E” e tem como saída o resultado da operação de 2 valores, sejam 0 ou 1, possui duas entradas:

* in\_port\_a: Vai introduzir o primeiro valor para operação;
* in\_port\_a: Vai introduzir o segundo um valor para operação, e tem uma única porta de saída:
* out\_port: porta que vai portar o valor da operação E entre in\_port\_a e in\_port\_b;

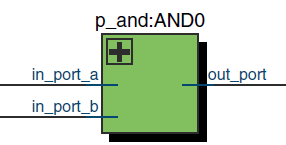
****

Figura 8 - Bloco simbólico do componente p\_and gerado pelo Quartus

### Mult\_2x1

O componente tem a função de um seletor de sinal, vai selecionar um sinal de entrada e vai deixar sair o sinal selecionado, seja ele 0 ou 1, possui 2 entradas de 8bits cada:

* in\_a: porta que vai receber o sinal;
* in\_b: porta que vai receber o sinal, e possui uma única saída de 8 bits:
* out\_port: porta que vai conter o sinal selecionado dentro do multiplexador;

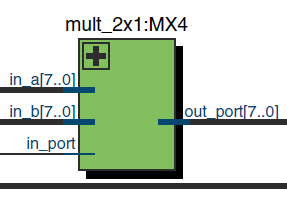
****

Figura 9 - Bloco simbólico do componente mult\_2x1 gerado pelo Quartus

### PC

O componente cprogram\_counter funciona como o GPS dentro do processador, ele é que dirá ao dado o seu destino, de acordo com o seu objetivo, por isso, este componente só possui uma entrada de 8bits:

* in\_port: recebe a informação que será destinada

E uma saída de 8bits também:

* out\_port: é o caminho que a informação já endereçada toma;

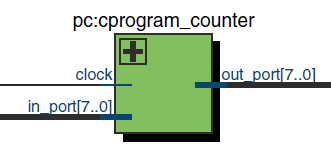
****

Figura 10 - Bloco simbólico do componente pc gerado pelo Quartus

### Extensor

O componente extensor2\_8 funciona como um extensor de sinal, transformando uma entrada de 2 bits em um sinal de 8 bits, possui, portanto, uma única entrada:

* in\_a: recebe o sinal de 2 bits que será amplificado, e uma única saída:
* out\_b: saída do sinal amplificado de 8 bits;

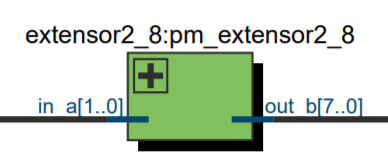


Figura 11 - Bloco simbólico do componente Extensor gerado pelo Quartus

# Datapath

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

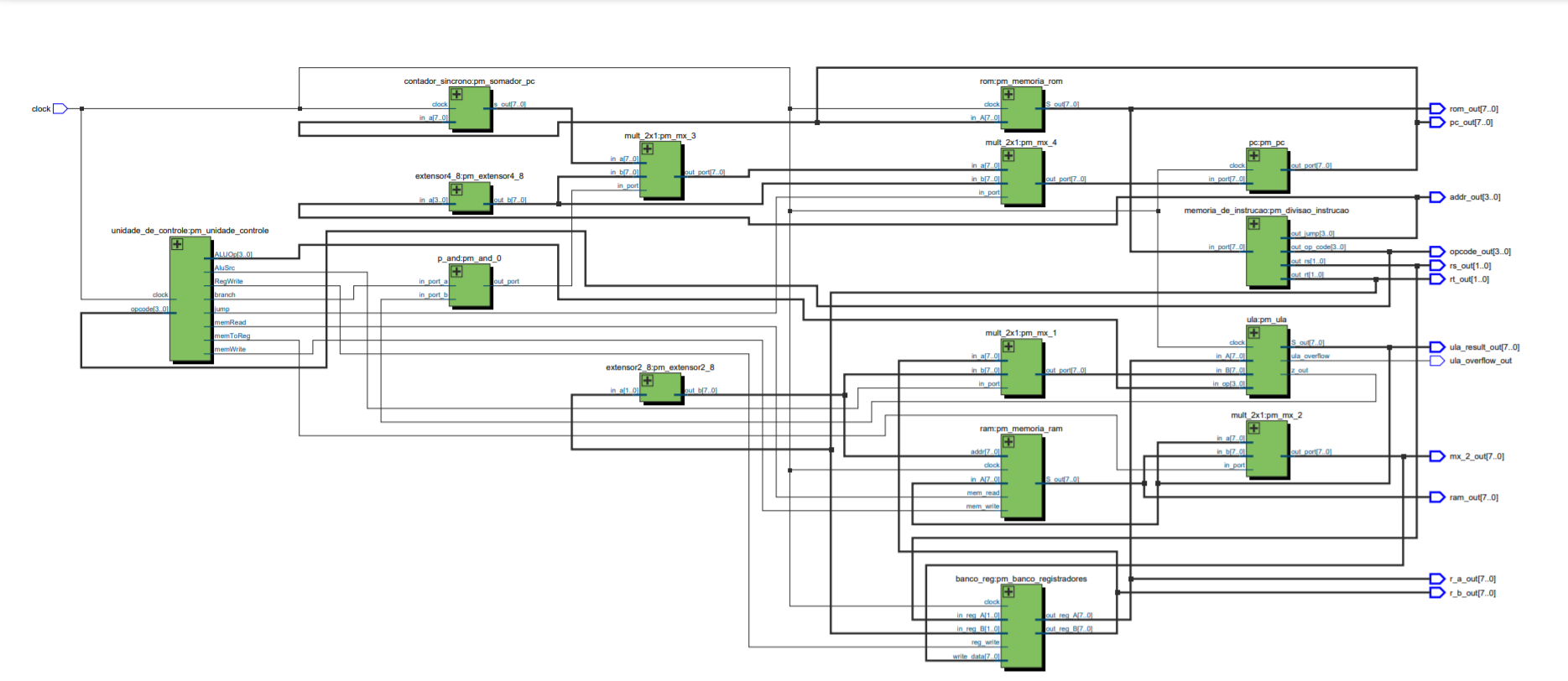


Figura 12 - Datapath gerado pelo Quartus do processador GDZ

# Simulações e Testes

Objetivando analisar, verificando 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. Com a finalidade de demonstrar o funcionamento do processador GDZ. Utilizando como exemplo o código para calcular o número da sequência de Fatorial e Fibonacci.

## Fatorial

Começamos o código do fatorial carregando o valor 1 em $s0 pela instrução li (load immediate), em seguida carregamos o valor 1 em $s1 da mesma maneira, prosseguindo podemos ver que ocorre uma multiplicação pelo uso da instrução mul (multiplicação) entre os dois registradores $s1 e $s0 guardando assim o resultado em $s1, portanto dando continuidade para a próxima instrução temos apenas uma soma imediata no registrador $s0 com 1 e logo após finalizamos com uma instrução jump (pulo incondicional) que irá repetir o mesmo processo até que ocorra uma limitação no processo de calcular o fatorial de 6 (6! = 720) no qual ocorrerá um estouro de memória pois nosso banco de memória tem a capacidade de armazenar números de até 8bits.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Código detalhado em GDZ para a realização do Fatorial | | | | |
| Endereço | Sintaxe | instrução | | |
| Opcode | Rt | rs |
| 0 | li $s0 1 | 0111 | 00 | 01 |
| 1 | li $s1 1 | 0111 | 01 | 01 |
| 2 | mul $s1 $0 | 0100 | 01 | 00 |
| 3 | addi $s0 1 | 0001 | 00 | 01 |
| 4 | j 0010 | 1010 | 00 | 10 |

|  |  |
| --- | --- |
| Código em binário para a realização do Fatorial | |
| 01110001 | li $s0 1 |
| 01110101 | li $s1 1 |
| 01000100 | mul $s1 $0 |
| 00010001 | addi $s0 1 |
| 10100010 | j 0010 |

Tabela 4 - Código Fatorial para o processador GDZ/EXEMPLO.

### Testes do Fatorial

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

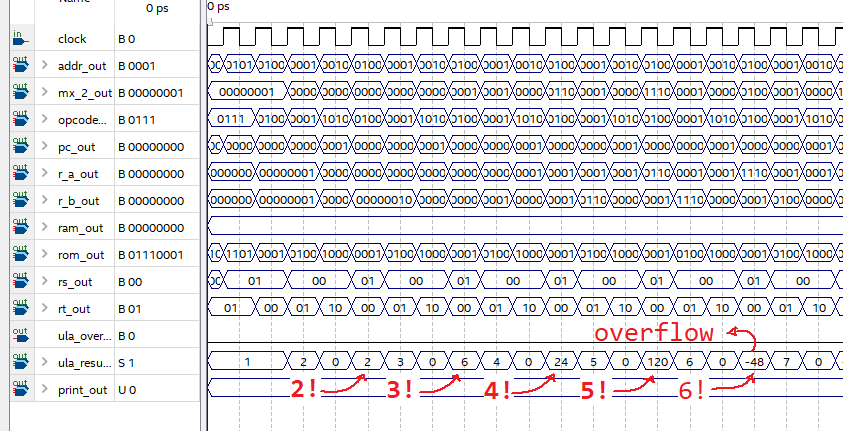


Figura 13 - Resultado na waveform.

## Fibonacci

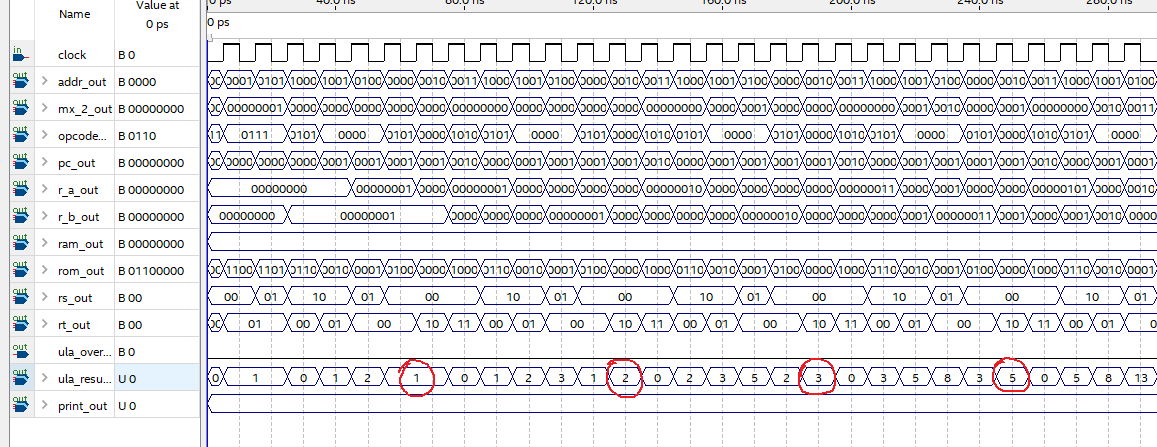
Na implementação do Fibonacci, observamos que a nossa arquitetura não suporta a operação move, portanto temos que salvar em memória uma constante zero que nos servirá somente para limpeza de registrador ao usarmos uma operação lw (load word), iniciamos dois registradores $s0 e $s1 com valores 0 e 1 respectivamente, prosseguindo para a próxima instrução iremos fazer uso do lw do endereço 00 no registrador $s2, ou seja, reescrevemos o registrador $s2 com o valor da constante 0 para então guardamos o valor de $s1, agora com todos esses estados executados podemos dar continuidade a operação de soma entre $s1 e $s0 que de fato irá dar origem ao Fibonacci, após o valor resultante da soma ser devolvida ao registrador $s1, temos que zerar o valor do registrador $s0 para então reatribuirmos o antigo valor de $s1 que está guardado no registrador $s2 e finalizamos com uma instrução jump para o endereço 0011.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Código detalhado em GDZ para a realização do Fibonacci | | | | |
| Endereço | Sintaxe | Instrução | | |
| Opcode | Rt | Rs |
| 0 | sw $s0 00 | 0110 | 00 | 00 |
| 1 | li $s0 1 | 0111 | 00 | 01 |
| 2 | li $s1 1 | 0111 | 01 | 01 |
| 3 | lw $s2 00 | 0101 | 10 | 00 |
| 4 | add $s2 $s1 | 0000 | 10 | 01 |
| 5 | add $s1 $s0 | 0000 | 01 | 00 |
| 6 | lw $s0 00 | 0101 | 00 | 00 |
| 7 | add $s0 $s2 | 0000 | 00 | 10 |
| 8 | j 0011 | 1010 | 0011 | |

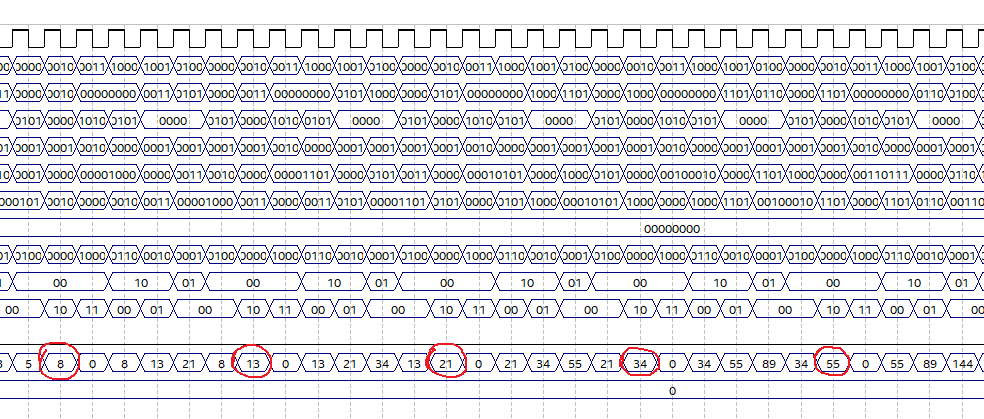
|  |  |
| --- | --- |
| Código em binário para a realização do Fibonacci | |
| 01100000 | sw $s0 00 |
| 01110001 | li $s0 1 |
| 01110101 | li $s1 1 |
| 01011000 | lw $s2 00 |
| 00001001 | add $s2 $s1 |
| 00000100 | add $s1 $s0 |
| 01010000 | lw $s0 00 |
| 00000010 | add $s0 $s2 |
| 10100011 | j 0011 |

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

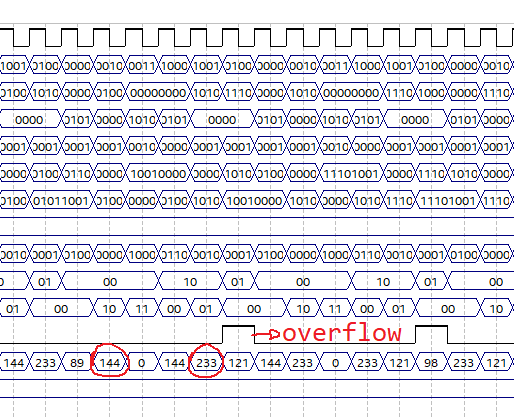
### Testes Fibonacci



**Figura 3 - Resultado na waveform.**



**Figura 3 - Resultado na waveform.**



**Figura 3 - Resultado na waveform.**

# 4. Considerações finais

A realização desse processador (GDZ) nos esclareceu as mais diversas dúvidas sobre questões de implementação de um grande projeto, fases de pesquisa e trabalho em equipe, fatores importantes que farão a diferença como Cientistas da Computação, logicamente passamos por dificuldades em virtude de trabalharmos apenas com 8 bits nas instruções.

Porém, devido algumas decisões do projeto nosso processador não possui algumas operações extras como por exemplo o **move**, esse que conseguimos simular através de um **sw** e **lw,** outro fator limitante em nossa arquitetura são as operações de desvio condicionais e incondicionais, onde somente podemos saltar 15 endereços (de 0000 a 1111).

# 5. Referências Bibliográficas

PATTERSON, D.; HENESSY, J. L. Organização e projeto de computadores: a interface hardware/software. 4ª Edição. São Paulo: Elsevier, 2013, 484 p.