



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

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR risc8bitProcessor

**GEORGE LUCAS MONÇÃO ZAMBONIN
2018001262**

**Março de 2022
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 risc8bitProcessor

**Março de 2022
Boa Vista/Roraima**

Resumo

Este trabalho aborda o projeto e implementação de um processador 8-bit MIPS RISC, utilizando a ferramenta de design Intel Quartus Prime Lite, que faz uso de uma linguagem de descrição de hardware, o VHDL (Hardware Description Language).

O relatório abordará e especificará as instruções e componentes do presente processador.

Palavras Chaves: 8-bit Processador, Intel, Quartus Prime, MIPS, RISC, VHDL, hardware.

Conteúdo

Especificação	5
Plataforma de desenvolvimento	5
Conjunto de instruções	6
Descrição do Hardware	8
ALU ou ULA	8
ALU Control	9
BDRRegister	9
Control Unit	10
Memória de dados	12
Memória de Instruções	12
PC	13
PC Add	13
JUMP_MUX	13
BRANCH_MUX	14
Datapath	15
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.	15
RTL Viewer	16
Simulações e Testes	17
Considerações finais	20

1 Especificação

Nesta seção é apresentado o conjunto de itens para o desenvolvimento do processador **risc8bitProcessor**, bem como a descrição detalhada de cada etapa da construção do processador.

1.1 Plataforma de desenvolvimento

Para a implementação do processador **risc8bitProcessor** foi utilizado a IDE: Quartus Prime Lite

Flow Status	Successful - Wed Mar 09 16:51:58 2022
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	risc8bitprocessor
Top-level Entity Name	RISC8BITPROCESSOR
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	1,228 / 56,480 (2 %)
Total registers	2088
Total pins	34 / 268 (13 %)
Total virtual pins	0
Total block memory bits	0 / 7,024,640 (0 %)
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 %)

Figura 1- Especificações no Quartus

1.2 Conjunto de instruções

O processador **risc8bitProcessor** possui 8 registradores: \$0, \$1, \$2 e \$3. 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, tipo I) é o registrador de destino;
- **Reg2:** o registrador contendo o segundo operando fonte;
- **Funct:** responsável por diferenciar operações aritméticas;
- **Immediate:** carrega informação a ser usada de forma imediata (ex: carregar um valor a ser armazenado a um registrador);
- **Address:** carrega o endereço de alguma instrução que esteja na memória.

Tipo de Instruções:

- **Formato do tipo R (Registro):** as instruções tipo R são instrução de registro para registro. São responsáveis por funções aritméticas entre registradores.

3 bits	2 bits	2 bits	1 bit
7-5	4-3	2-1	0
Opcode	Reg1	Reg2	Funct

Tabela 1 - Instruções tipo R

- **Formato do tipo I (Immediate):** instruções deste tipo permitem operações de Branch, operações imediatas na ALU, Load e Store. Nesta instrução, dois tipos de registradores são especificados e um valor de 1 bit usado para operações.

3 bits	2 bits	2 bits	1 bit
7-5	4-3	2-1	1
Opcode	Reg1	Reg2 / Immediate	Immediate_en

Tabela 2 - Instruções tipo I

- **Tipo J (Jump):** instrução responsável por pulos entre a memória de instruções. Possui 5 bits dedicados apenas para o endereço destino.

3 bits	5 bits
7-5	4-0
Opcode	Target

Tabela 3 - Instruções tipo J

- **Tipo Pseudo:** uma pseudo instrução usada exclusivamente para o op li, que é responsável por carregar um valor imediato a um registrador.

3 bits	2 bits	3 bits
7-5	4-3	2-0
Opcode	Reg	Immediate

Tabela 4 - Instruções tipo Pseudo

Visão geral das instruções do Processador risc8bitProcessor:

O número de bits do campo **Opcode** das instruções é igual a três, sendo assim obtemos um total $(Bit(0e1)^{NumeroTotaldeBitsdoOpcode} : 2^3 = 8)$ de 8 **Opcodes (0-8)** que são distribuídos entre as instruções, assim como é apresentado na Tabela 1.

Opcode	Func / Immediate_en	Nome	Tipo	Breve Descrição	Exemplo
000	0	ADD	R	Soma	add \$S0, \$S1
000	1	SUB	R	Soma	sub \$S0, \$S1

001	Não possui	J	J	Jump	j \$S0, target_address
010	0	AND	R	Condicional E	div \$S0, \$S1
011	0	BEQ	R	Condicional Igual	beq \$S0, \$S1
100	0	SLT	R	Condicional Menor que	slt \$S0, \$S1
101	1 / 0	LW	I	Load Word	lw \$S0, \$S1 , lw \$S0, address
110	1 / 0	SW	I	Store Word	sw \$S0, \$S1 , sw \$S0, address
111	Não possui	LI	Pseudo	Load Immediately	li \$S0, 7

Tabela 5 – Tabela que mostra a lista de Opcodes utilizadas pelo processador risc8bitProcessor.

1.3 Descrição do Hardware

Nesta seção são descritos os componentes do hardware que compõem o processador **risc8bitProcessor**, incluindo uma descrição de suas funcionalidades, valores de entrada e saída.

1.3.1 ALU ou ULA

O componente ALU (Unidade Lógica Aritmética) tem como principal objetivo efetuar as principais operações aritméticas, dentre elas: soma, subtração. Adicionalmente o ALU efetua operações de comparação igual e somente menor. E operações lógicas, como E e OU. O componente ALU recebe como entrada três valores: **A** – dado de 8 bits para operação; **B** - dado de 8 bits para operação e **OP** – identificador da operação que será realizada de 4 bits. O ALU também possui duas saídas: **zero** – identificador de resultado (2bit) para comparações (1 se verdade e 0 caso contrário); e **result** – saída com o resultado das operações aritméticas.

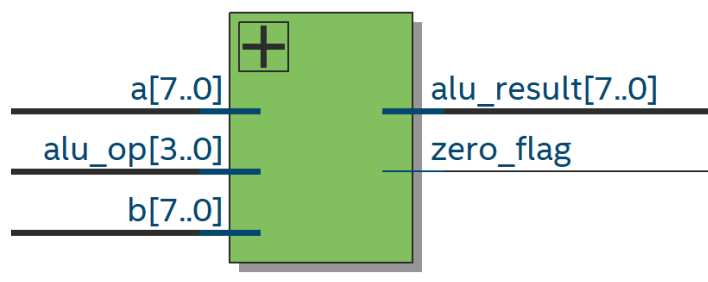


Figura 2 - Bloco simbólico do componente ALU gerado pelo Quartus

1.3.2 ALU Control

O componente **ALU Control** (Controle da ALU) é responsável por controlar as operações a serem executadas na ALU. Ele possui como entrada o **ALU_Funct** é responsável por diferenciar funções do mesmo opcode de instruções do tipo R e, também como entrada, possui o opcode da instrução sendo executada. Com ambas informações, o **ALU Control** manda um opcode mais detalhado para ALU, este opcode tem 1 bit a mais do que o da instrução, 4 bits no total.

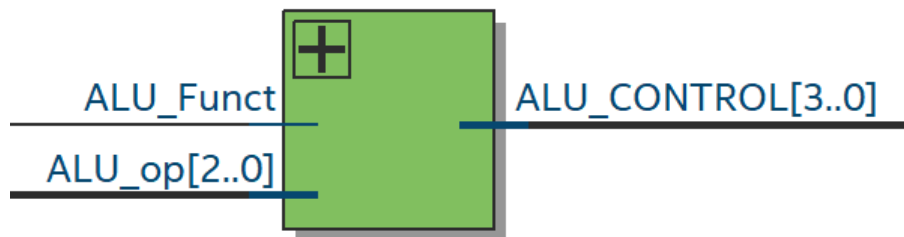


Figura 3 - Bloco simbólico do componente ALU Control gerado pelo Quartus

1.3.3 BDRRegister

O Banco de Registradores é um componente responsável por armazenar valores de fácil acesso, para uso em outros componentes. Possuindo 4 registradores de 8 bits, o banco de registradores possui como saída 2 registradores (**reg1** e **reg2**). Como entrada, o componente possui o **clock**, **reset**, os endereços de ambos registradores a serem acessados e, para ser sobrescrito, o registrador conta com um sinal de entrada **enable** responsável por permitir a escrita de um registrador, dois sinais de 8 bits e 2 bits, respectivamente, contendo o valor a ser escrito e o endereço do registrador a ser escrito. Também, para permitir o opcode **LI**, possui como entrada o **enable_li** e um sinal de 8 bits contendo o valor a ser escrito por **LI**.

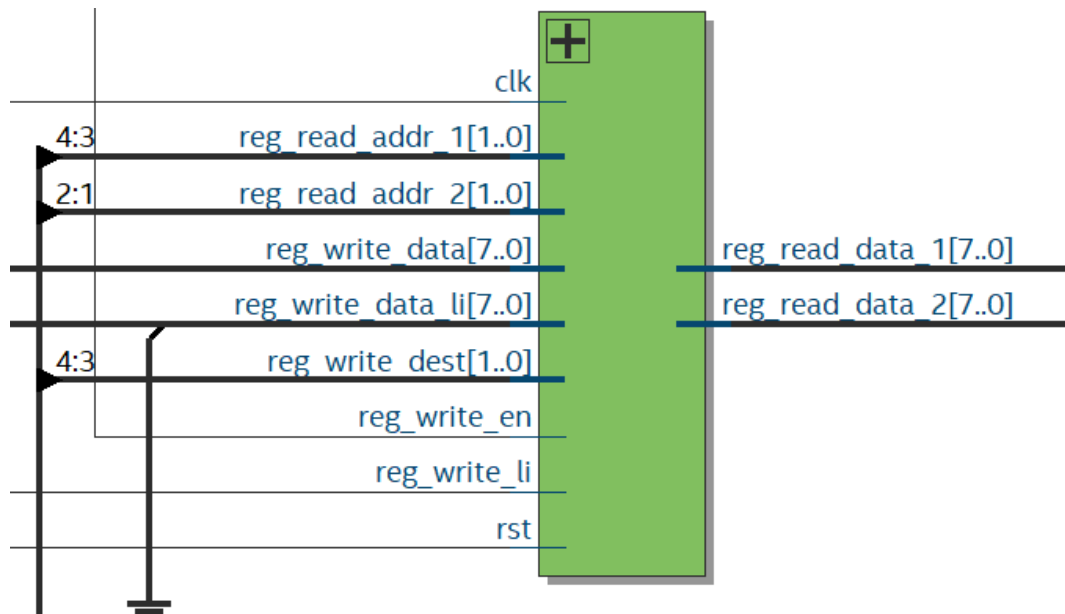


Figura 4 - Bloco simbólico do componente BDRegister gerado pelo Quartus

1.3.4 Control Unit

O componente **Control Unit** (Unidade de Controle) 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:

- **ALU_op**: opcode que será repassado para a controladora da ALU.;
- **Branch**: Ativo em operações de desvios condicionais;
- **Jump**: Usado para sinalizar que a instrução é do tipo J (jump);
- **mem_read**: Sinaliza a memória de dados que um endereço deverá ser lido.
- **mem_write_en**: Sinaliza a memória de dados que um endereço deverá ser escrito.
- **reg_write_en**: sinaliza ao Banco de Registradores que um registrador será reescrito.
- **reg_write_li**: sinaliza que o opcode li está sendo executado, permitindo a escrita imediata no banco de registradores.

- **immediate_en**: sinaliza que um código do tipo I está em execução.

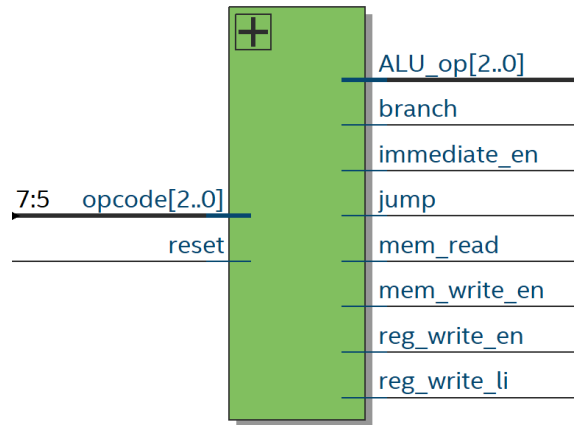


Figura 5 - Bloco simbólico do componente Control Unit gerado pelo Quartus

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

Comando	ALU op	branc h	immedia te en	jum p	me m read	mem write_e n	reg writ e en	reg writ e li
add	000	0	0	0	0	0	1	0
sub	000	0	0	0	0	0	1	0
jump	001	0	0	1	0	0	0	0
and	010	1	0	0	0	0	0	0
beq	011	1	0	0	0	0	0	0
slt	100	1	0	0	0	0	0	0
lw	101	0	1	0	1	0	1	0
sw	110	0	1	0	0	1	0	0
li	111	0	0	0	0	0	0	1
Inicializa ção	000	0	0	0	0	0	0	0

Tabela 6 - Detalhes das flags de controle do processador.

1.3.5 Memória de dados

O componente **Memória de Dados** é responsável por armazenar um grande volume de dados em uma memória secundária. Para permitir um grande armazenamento, acaba sacrificando a velocidade de acesso em relação aos registradores.

Este componente tem como entrada o clock, as flags: `immediate_en`, `mem_read`, `mem_write_en`; os endereços de acesso imediato e por acesso de registrador; um sinal de 8 bits contendo os dados a serem escritos na memória. Já como saída, a memória de dados também possui os dados da memória lida.

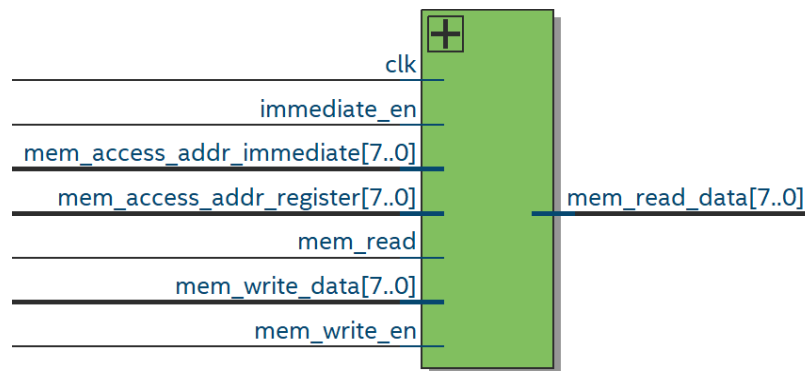


Figura 6 - Bloco simbólico do componente Memória de Dados gerado pelo Quartus

1.3.6 Memória de Instruções

A memória de instruções contém todas as instruções a serem executadas no data path. Este componente recebe o pc, um sinal que contém o endereço de uma instrução, esta instrução selecionada será executada durante o ciclo atual.

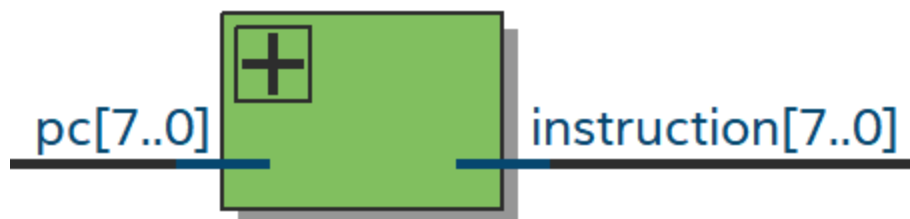


Figura 7 - Bloco simbólico do componente Memória de Instruções gerado pelo Quartus

1.3.7 PC

O PC é um componente que carrega e atualiza o endereço a ser recebido pela memória de instruções. Ele tem como saída o endereço da memória de instruções a ser executada no ciclo atual. Já como entrada contém o próximo endereço a ser lido.

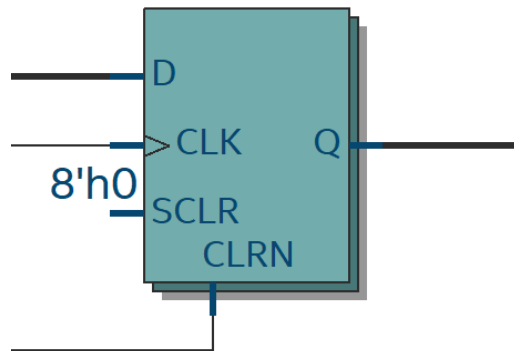


Figura 8 - Bloco simbólico do componente PC gerado pelo Quartus

1.3.8 PC Add

O PC Add é o componente mais simples do processador. A cada ciclo ele incrementa o endereço da instrução sendo executada.

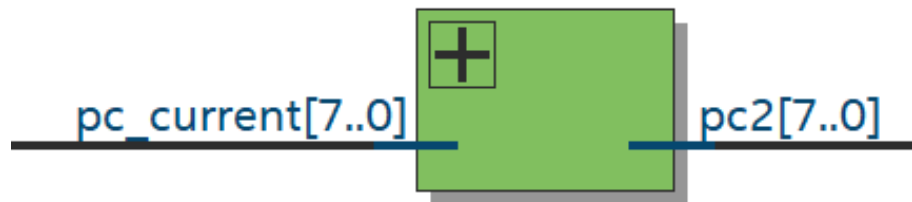


Figura 9 - Bloco simbólico do componente PC Add gerado pelo Quartus

1.3.9 JUMP_MUX

O **Jump Mux** é um componente que funciona como a alavanca de um trilho de trem, ele desvia o endereço a ser executado no próximo ciclo dependendo da instrução atual. Caso a instrução atual seja do tipo J, ele irá receber a flag jump,

selecionando o endereço alvo do jump como saída e, caso não seja uma instrução tipo J, ele selecionará o endereço do clock atual incrementado.

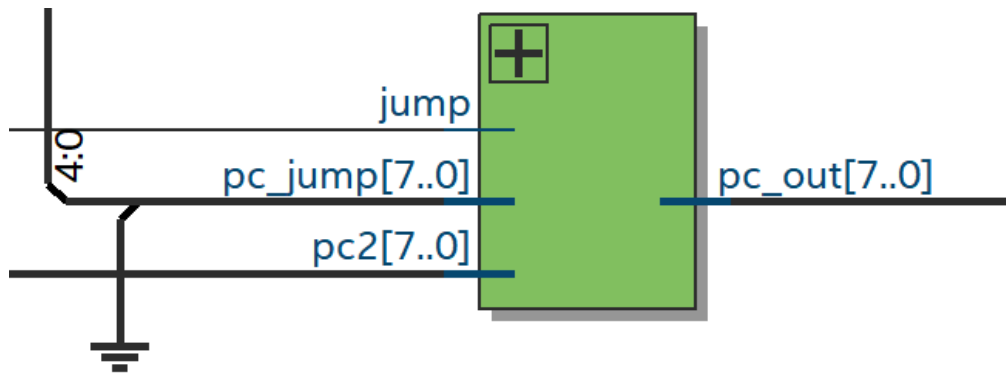


Figura 10 - Bloco simbólico do componente JUMP_MUX gerado pelo Quartus

1.3.10 BRANCH_MUX

O **Branch Mux** é um componente similar ao Jump Mux. Suas principais diferenças é que ele recebe apenas o endereço da instrução atual incrementado e ramifica esse endereço ainda mais incrementado. Caso a instrução atual seja um desvio condicional (branch), receberá a flag enable ativa, tendo como saída o mesmo endereço de entrada, caso a flag esteja inativa, o Branch Mux irá selecionar o novo endereço ramificado e incrementado, isso fará com que a próxima instrução seja pulada caso a flag zero esteja ativa.

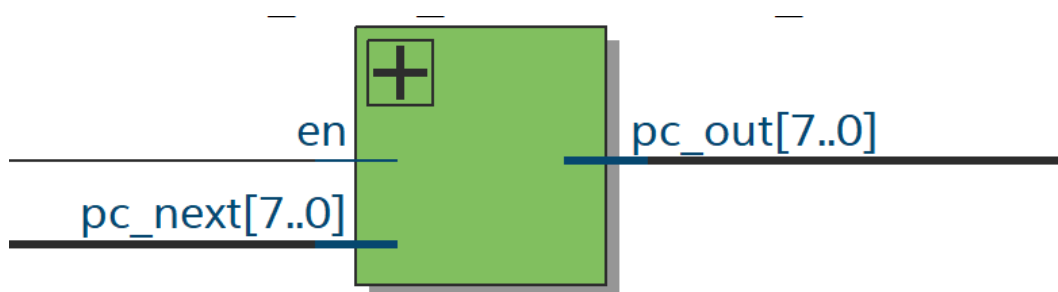


Figura 11 - Bloco simbólico do componente BRANCH_MUX gerado pelo Quartus

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.

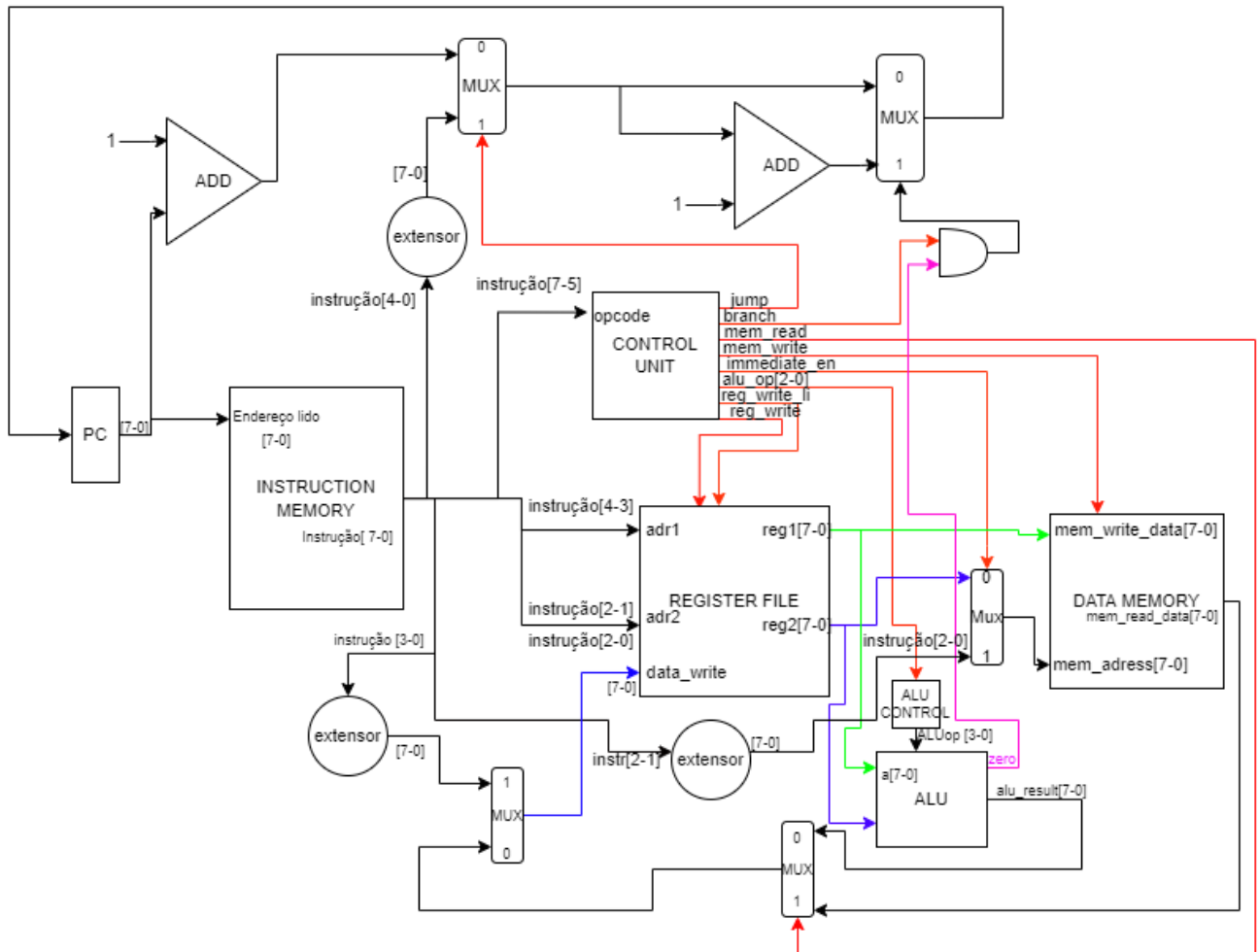


Figura 12- Datapath.

1.5 RTL Viewer

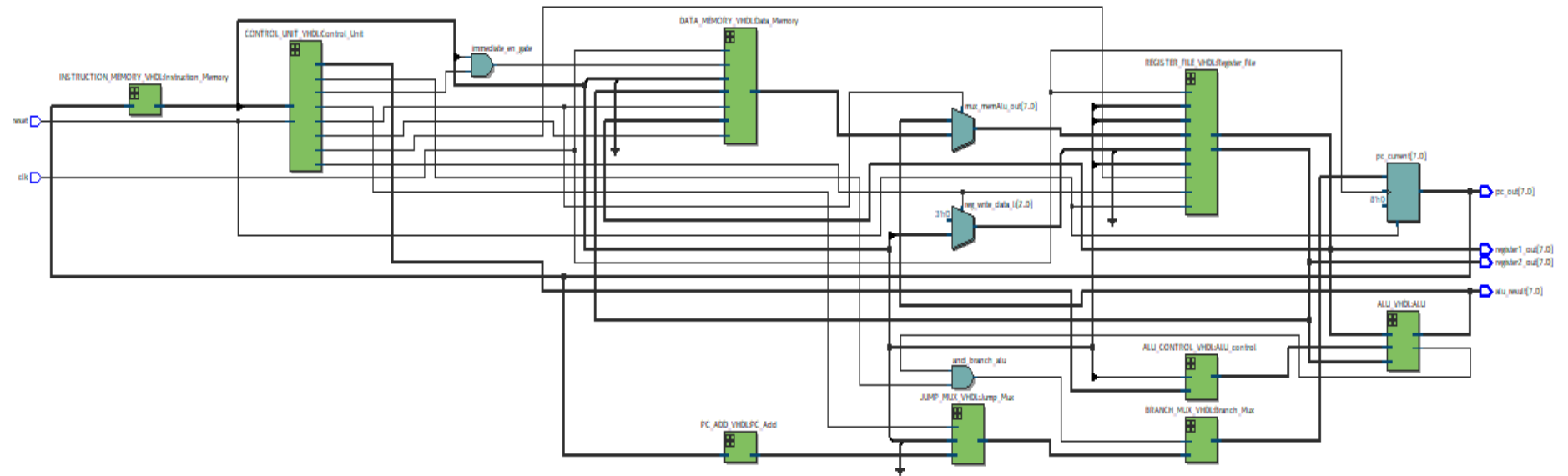


Figura 13- RTL Viewer gerado pelo Quartus

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 **risc8bitProcessor** utilizaremos como exemplo o código para calcular o número da sequência de Fibonacci.

Endereço	Linguagem de Alto Nível	Binário			
		3-bits	2-bits	2-bits	1-bit
		Opcode	R1	R2	F
			R1	R2 / I	I_E
			Endereço		
Dado					
0	li \$0, 7	111	00	111	
1	add \$0, \$0	000	00	00	0
2	add \$0, \$0	000	00	00	00
3	add \$0, \$0	000	00	00	00
4	add \$0, \$0	000	00	00	00
5	li \$1, 3	111	01	011	
6	sw \$0, 3	110	00	11	1
7	li \$1, 1	111	01	001	
8	li \$2, 0	111	10	000	
9	li \$3 5	111	11	101	
10	sw \$1, 0	110	01	00	1
11	lw \$0, 0	101	00	00	1
12	sw \$1, \$0	110	01	00	0
13	sw \$1, \$3	110	01	11	0
14	lw \$1, 0	101	01	00	1
15	add \$3, \$0	000	11	00	0
16	lw \$1, 1	101	01	01	1
17	add \$1, \$2	000	01	10	0
18	lw \$2, 1	101	10	01	1
19	lw \$0, 3	101	00	11	1
20	slt \$1, \$0	100	01	00	0
21	j 12	001	01011		
22	li \$0, 0	111	00	000	
23	li \$1, 5	111	01	101	
24	sw \$0, \$1	110	00	01	0

Tabela 7- Código Fibonacci para o processador **risc8bitProcessor**.

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

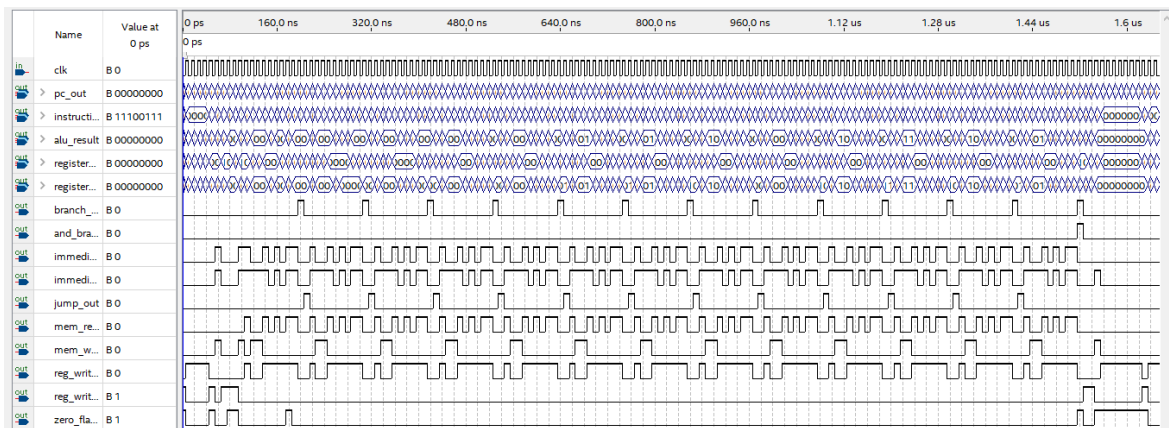
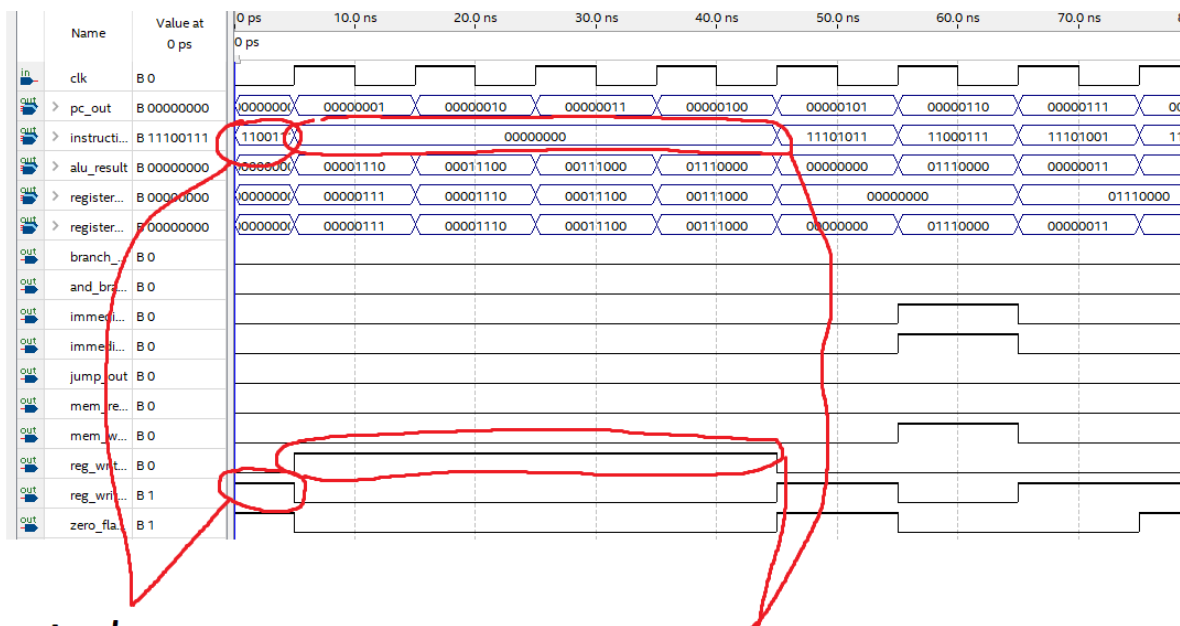


Figura 14 - Waveform gerada pelo Quartus.

Chegando mais perto, nessa waveform, poderemos analisar melhor:



li \$0 7

multiplos add \$0, \$0
para dobrar o \$0

Figura 15 - Parte inicial da Waveform gerada pelo Quartus.

É possível analisar as instruções iniciais do algoritmo fibonacci, onde lê-se um número com o maior valor de li, e o dobra múltiplas vezes com adições. Este número servirá como um limite para a sequência, definindo até onde ela vai.

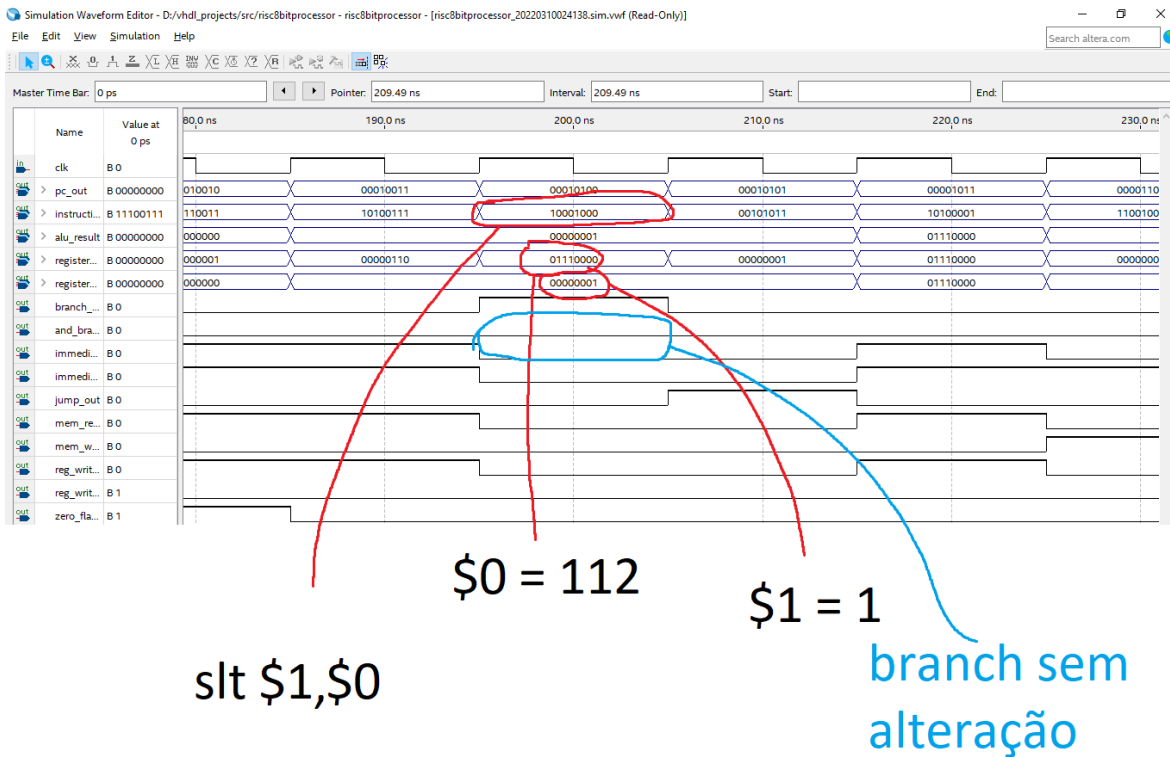
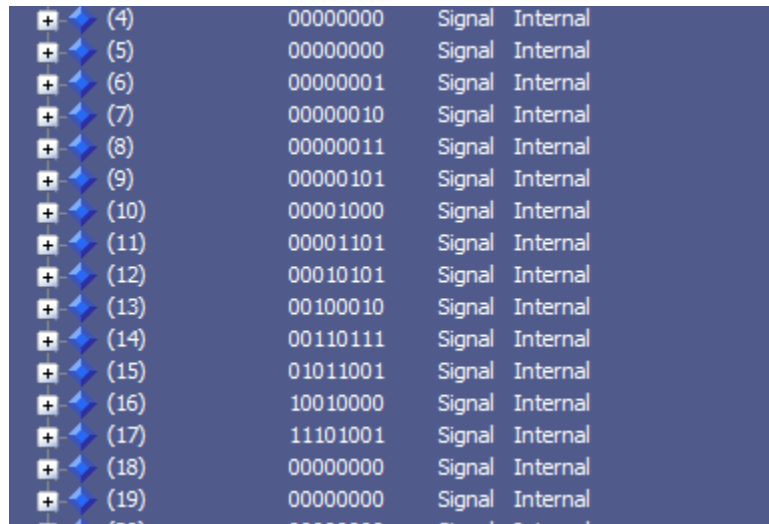


Figura 16 - SLT da Waveform gerada pelo Quartus.

Aqui podemos perceber que \$0, o número dobrado n vezes é $112(7 \cdot 2^4)$, está sendo comparado com o número atual da sequência fibonacci.

if (\$s1 < \$s0).

Ao final da execução do algoritmo, a memória de dados estará desta forma, com a sequência fibonacci escrita em sequência:



+	+	(4)	00000000	Signal	Internal
+	+	(5)	00000000	Signal	Internal
+	+	(6)	00000001	Signal	Internal
+	+	(7)	00000010	Signal	Internal
+	+	(8)	00000011	Signal	Internal
+	+	(9)	00000101	Signal	Internal
+	+	(10)	00001000	Signal	Internal
+	+	(11)	00001101	Signal	Internal
+	+	(12)	00010101	Signal	Internal
+	+	(13)	00100010	Signal	Internal
+	+	(14)	00110111	Signal	Internal
+	+	(15)	01011001	Signal	Internal
+	+	(16)	10010000	Signal	Internal
+	+	(17)	11101001	Signal	Internal
+	+	(18)	00000000	Signal	Internal
+	+	(19)	00000000	Signal	Internal

Figura 16 - SLT da Waveform gerada pelo Quartus.

3 Considerações finais

Este trabalho apresentou o projeto e implementação do processador de 8 bits denominado de **risc8bitProcessor**. Todo desenvolvimento do projeto está disponível no github, diferentes etapas no desenvolvimento estão separadas em branch (versionamento).

Primeiramente iniciado com o desenvolvimento da ALU, posteriormente a instrução de memória e outros componentes da arquitetura RISC. Desenvolvido até o presente momento, mesmo com algumas limitações, o processador é capaz de executar algoritmos complexos como gravar a sequência fibonacci em diferentes bytes de sua memória de dados.

Portanto, concluo que o processador alcançou minhas expectativas e objetivos.