UFRR

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

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR AMP - I

ALUNOS:

Bruno Rodrigues Caputo - 2016000041
Philip Mahama Akpanyi - 201514402

Novembro de 2018
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 AMP - I(Advanced Micro Processor versao I)

Novembro de 2018
Boa Vista/Roraima

Resumo

Este trabalho aborda o projeto e implementação de um processador de 16 bits chamado AMP-I(Advanced Micro Processor), desenvolvido como trabalho para disciplina de arquitetura e organização de computadores. Implementado em VHDL(VHSIC *Hardware Description Language*) buscou-se criar uma arquitetura enxuta e coerente que seja capaz de realizar algoritmos inteligíveis. Seus componentes bem como o *datapath* serão aprofundados ao longo deste documento.

Conteúdo

1 Especificação

Nesta seção é apresentado o conjunto de itens para o desenvolvimento do processador AMP-I(Advanced Micro Processor), bem como a descrição detalhada de cada etapa da construção do processador.

1.1 Plataforma de desenvolvimento

Para a implementação do processador AMP foi utilizado a IDE: Quartus (Quartus Prime 18.1) Lite Edition.....

Flow Status	Successful - Mon Aug 27 17:33:50 2012
Quartus II Version	9.0 Build 184 04/29/2009 SP 1 SJ Web Edition
Revision Name	Quantum
Top-level Entity Name	Quantum_Pro
Family	Cyclone III
Met timing requirements	N/A
Total logic elements	3,123 / 15,408 (20 %)
Total combinational functions	2,148 / 15,408 (14 %)
Dedicated logic registers	2,137 / 15,408 (14 %)
Total registers	2137
Total pins	188 / 347 (54 %)
Total virtual pins	0
Total memory bits	0 / 516,096 (0 %)
Embedded Multiplier 9-bit elements	1 / 112 (< 1 %)
Total PLLs	0 / 4 (0 %)
Device	EP3C16F484C6
Timing Models	Final

Figura 1 - Especificações no Quartus

1.2 Conjunto de instruções

O processador AMP-I possui 32 registradores: $0, $alto, $t0 ,$t2, $t3, $t4, $t5, $t6, $t7, $t8, $t9, $t10, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7, $s8, $s9, $s10, $s11, $s12. Assim como 12 formatos de instruções de 16 bits cada,seguem algumas considerações sobre as estruturas contidas nas instruções:

- **Opcode**: a operação básica a ser executada pelo processador;
- **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;
- **Func:** Instruções compartilhadas com o opcode que servem para utilizar funções caso seja necessário

1.2.1 Tipo de Instruções

- Formato do tipo R:

Instruções baseadas em operações aritméticas.

Formato para escrita de código na linguagem Quantum:

Tipo da Instrução Reg1 Reg2

Formato para escrita em código binário:

4 bits	5 bits	5 bits
15-12	11-7	6-2
Opcode	Reg2	Reg1

- Formato do tipo I:

Este formato aborda instruções de Load (exceto *load Immediately*), Store e

Formato para escrita de código na linguagem Quantum:

Tipo da Instrução Reg1 Reg2 Func

Formato para escrita em código binário:

4 bits	5 bits	5 bits	2 bits
15-12	11-7	6-2	1-0
Opcode	Reg2	Reg1	Func

- Formato do tipo J:

Este formato aborda instruções de saltos condicionais ou não de um determinado endereço segue uma instrução única:

Formato para escrita de código na linguagem Quantum:

Tipo da Instrução Endereço

Formato para escrita em código binário:

4 bits	28 bits
15-12	11-0
Opcode	Endereço

1.2.2 Visão geral das instruções do Processador AMD-I:

O número de bits do campo **Opcode** das instruções é igual a quatro, sendo assim obtemos um total ($Bit(0e1)^{NumeroTodaldeBitsdoOpcode} \therefore 2^X = X$) 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 AMP-I.

Opcode	Nome	Formato	Breve Descrição	Exemplo
0000	add	R	Soma	**add** $S0, $S1 ,ou seja, $S0 := $S0+$S1
0001	sub	R	Subtração	**sub** $S0, $S1 ,ou seja, $S0 := $S0 - $S1
0010	mult	R	Multiplicação	**mult** $S0, $S1 ,ou seja, $S0 := $S0 * $S1
0011	addf	R	Soma Float	**add** $S0, $S1 ,ou seja, $S0 := $S0+$S1
0100	subf	R	Subtração Float	**sub** $S0, $S1 ,ou seja, $S0 := $S0 - $S1
0101	multf	R	Multiplicação Float	**mult** $S0, $S1 ,ou seja, $S0 := $S0 * $S1
0110	or	R	OU	**or** $S0, $S1 ,ou seja, $S0 := $S0 || $S1
0111	and	R	E	**and** $S0, $S1 ,ou seja, $S0 := $S0 & $S1
1000	lw	I	Carregar Palavra	**lw** $S0, C($s1) ,ou seja, $s0 = Memoria[$s1 +C]
1001	sw	I	Armazena na memória	**sw** C($s0) , $S1 ,ou seja, Memoria[$s0 +C]:=$s1
1010	beq	I	Se for igual	**beq** $S0, $S1 ,ou seja, if($S0 == $S0)then PC+1*2
1011	j	j	Pula para Endereço	**j** 0x19
1100	not	R	Negação do resultado	**not** $S0, ou seja $S0 := not ($S0)
1101	||			
1110	||			
1111	exit		Encerramento do processador	

1.3 Descrição do Hardware

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

1.3.1 ALU ou ULA

O componente QALU (Q Unidade Lógica Aritmética) tem como principal objetivo efetuar as principais operações aritméticas, dentre elas: soma, subtração, multiplicação (considerando apenas resultados inteiros). Adicionalmente a ALU efetua operações de comparação de valor como maior ou igual, menor ou igual, somente maior, menor ou igual. O componente ALU recebe como entrada três valores: **A** – dado de 16 bits para operação; **B** - dado de 16 bits para operação e **OP** – identificador da operação que será realizada de 4bits. O ALU possui três saídas: **zero** – identificador de resultado (2bit) para comparações (1 se verdade e 0 caso contrário); **overflow** – identificador de overflow caso a operação exceda os 16 bits; e **result** – saída com o resultado das operações aritméticas.

Figura 2 - Bloco simbólico do componente QALU gerado pelo Quartus

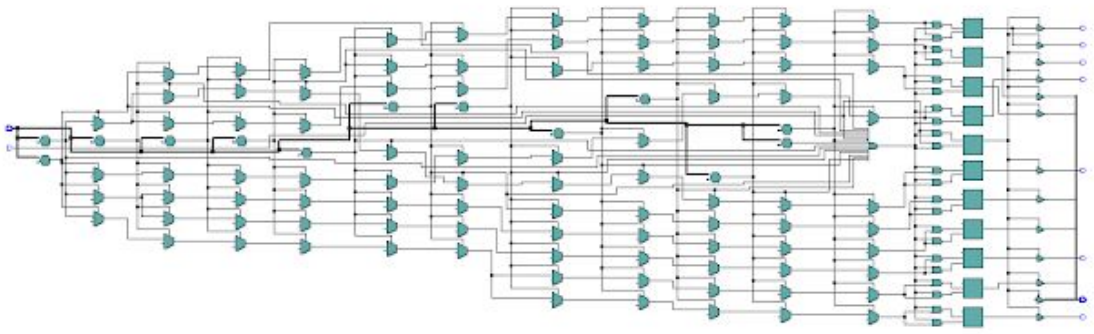
1.3.2 Unidade de Controle

O componente Unidade_Controle_16bits tem como objetivo realizar o controle de todos os componentes do processador de acordo com o opcode e as flags que serão detalhadas a seguir. Sua declaração segue o seguinte formato:

```
entity Unidade_Controle_16bits is
port(
   clk          :  in std_logic;
   opcode       :  in std_logic_vector (3 downto 0);
   regDst       :  out std_logic;
   ulaFont      :  out std_logic;
   memParaReg   :  out std_logic;
   escReg       :  out std_logic;
   lerMem       :  out std_logic;
   escMem       :  out std_logic;
   desvio       :  out std_logic;
   ulaOP        :  out std_logic_vector (7 downto 0)
);
end entity Unidade_Controle_16bits;
```

Nas portas temos:

- clk: clock para fazer checagem de borda alta
- opcode: Qual operação será realizada
- regDst: registrador para determinar especificação do registrador rt, rd
- ulaFont: Flag para o segundo dado lido do banco de registradores, no multiplexador que com a memória de dados
- memParaReg:Flag para sinalizar leitura e escrita na memória de dados do multiplexador
- escReg:Flag para sinalizar escrita no registrador
- lerMem:Flag para sinalizar leitura de memória
- escMem:Flag para sinalizar escrita de memória
- desvio:Flag para salto
- ulaOP:Operação realizada na ULA



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	clk	opcode	regDst	ulaFont	mem ParaReg	esc Reg	ler Mem	esc Mem	desvio	ulaOP
add	1	0000	1	0	0	1	0	0	0	00000000
sub	1	0001	1	0	0	1	0	0	0	00000001
mult	1	0010	1	0	0	1	0	0	0	00000010
addf	1	0011	1	0	0	1	0	0	0	00000011

subf	1	0100	1	0	0	1	0	0	0	00001000
multf	1	0101	1	0	0	1	0	0	0	00010101
or	1	0110	1	0	0	1	0	0	0	00000110
and	1	0111	1	0	0	1	0	0	0	00000111
load	1	1000	0	1	1	1	1	0	0	00001000
beq	1	1010	0	1	1	1	1	0	0	00001010
j	1	1011	0	0	1	0	0	0	1	00001011
not	1	1100	1	0	0	1	0	0	0	00001100

1.3.3 Memória de dados

Responsável pela leitura e escrita dos dados, que serão posteriormente armazenados ou não no banco de registradores, sua declaração abaixo:

```
use ieee.numeric_std.ALL;
entity ram16 is
    Port ( I_clk : in  STD_LOGIC;
           I_we : in  STD_LOGIC;
           I_addr : in  STD_LOGIC_VECTOR (15 downto 0);
           I_data : in  STD_LOGIC_VECTOR (15 downto 0);
           O_data : out  STD_LOGIC_VECTOR (15 downto 0));
end ram16;
```

Nas portas temos:

- I_clk: Entrada com o clock para checagem de borda alta
- I_we: Entrada para flag de escrita de dados
- I_addr: Entrada para endereço na memória de dados
- I_data: Dado a ser escrito
- O_data: Saída de dado que irá sair paro o banco de registradores

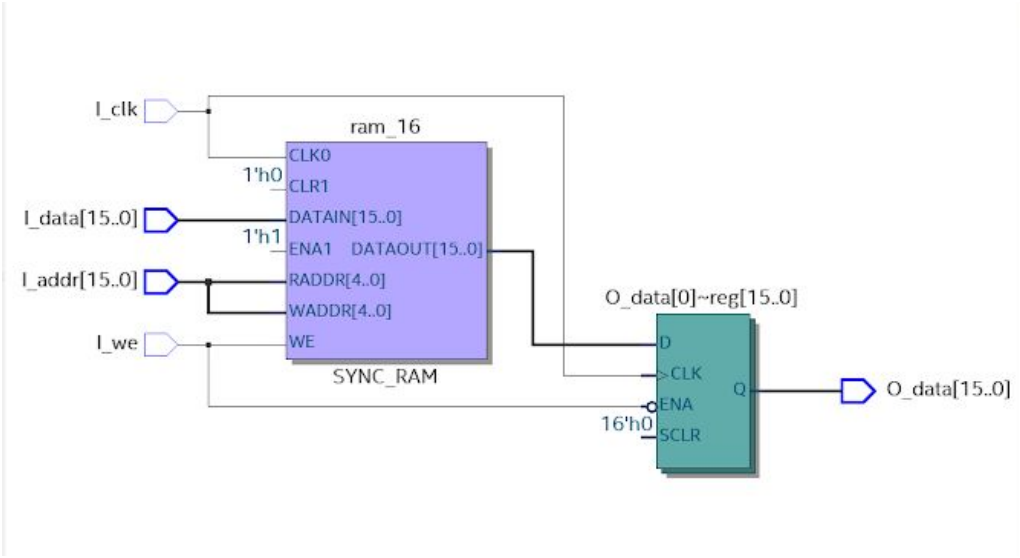


Figura 1 -Imagem *RTL* da memória RAM

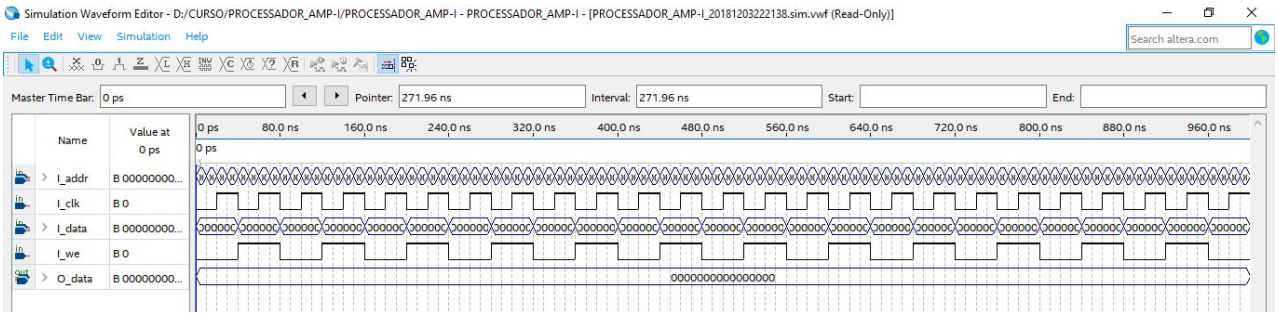


Figura 1 -Waveform da memória RAM

1.3.4 Memória de Instruções

As instruções a serem executadas são armazenadas na memória de instruções que é conhecido como ROM. Cada instrução tem seu próprio endereço

```
entity ROM_16bits is
    port(
    address: in std_logic_vector(15 downto 0);
    data: out std_logic_vector(3 downto 0)
);
end ROM_16bits;
```

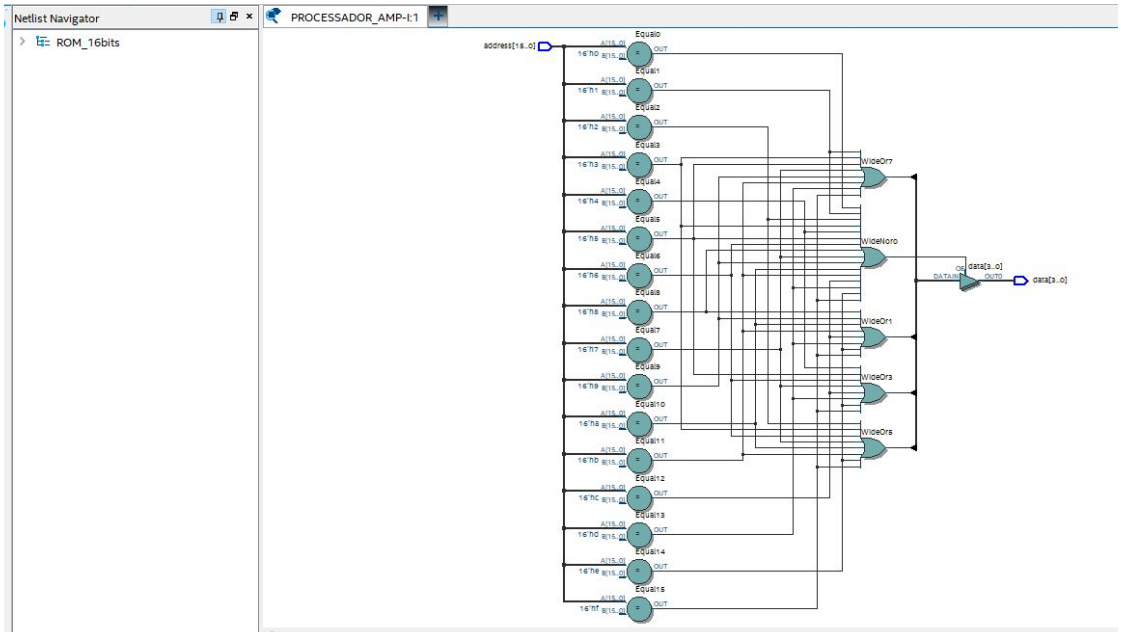


Figura 1 RTL imagem do ROM

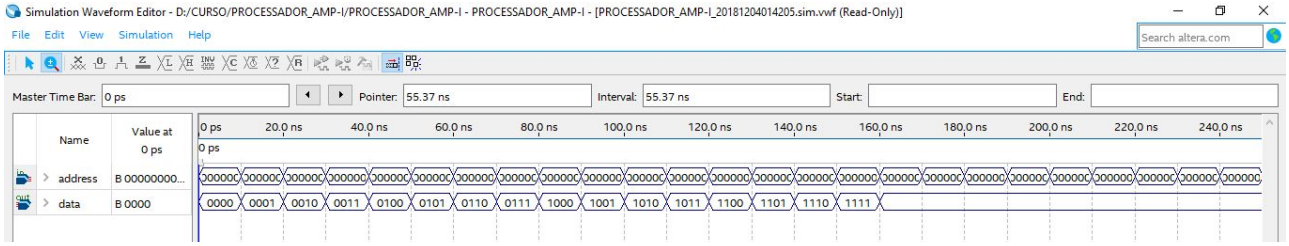


Figura 1 Imagem do Waveform do ROM

1.3.5 Banco de Registradores

Responsável por possuir todos os 32 registradores que podem ser utilizados ao longo do processador, também tem conexão entre memória de instrução e memória de dados.Sua declaração segue abaixo:

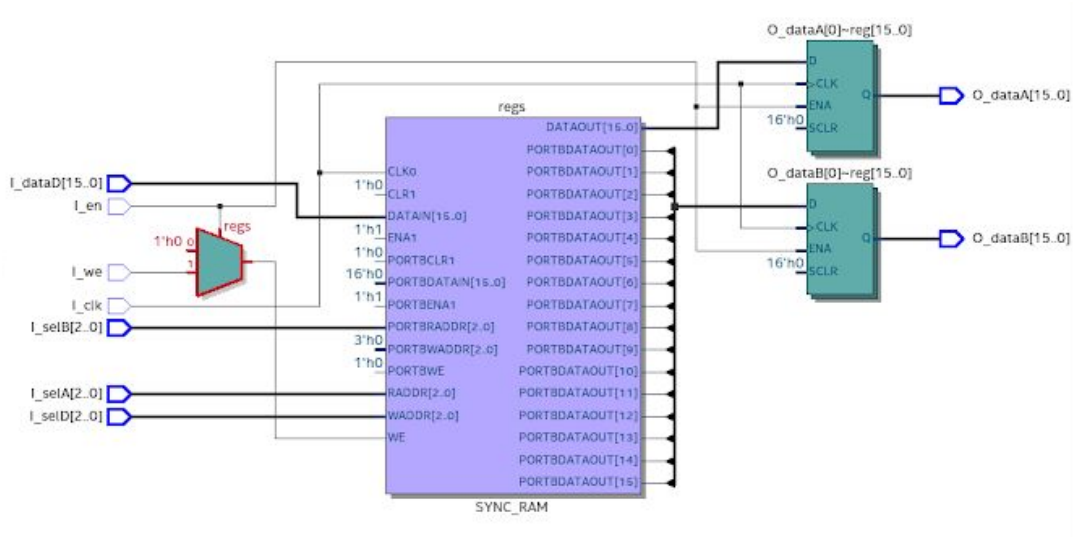
```
entity reg16_8 is
Port ( I_clk : in  STD_LOGIC;
       I_en : in  STD_LOGIC;
       I_dataD : in  STD_LOGIC_VECTOR (15 downto 0);
       O_dataA : out  STD_LOGIC_VECTOR (15 downto 0);
       O_dataB : out  STD_LOGIC_VECTOR (15 downto 0);
       I_selA : in  STD_LOGIC_VECTOR (2 downto 0);
       I_selB : in  STD_LOGIC_VECTOR (2 downto 0);
       I_selD : in  STD_LOGIC_VECTOR (2 downto 0);
       I_we : in  STD_LOGIC);
end reg16_8;
```

Nas portas temos:

- I_clk: clock para fazer atualizar o valor de acordo com as entradas A e B

- ❏ I_en: Flag para saber se ativada para registrar as entradas
- ❏ I_dataD: Entrada com o dado para ser escrito
- ❏ O_dataA: Saída dos dados do registrador A
- ❏ O_dataB:Saída dos dados do registrador B
- ❏ I_selA:Flag para selecionar A
- ❏ I_selB:Flag para selecionar B
- ❏ I_selD:Flag para selecionar D
- ❏ I_we:Flag para verificação de escrita no registrador

Figura 1 -Imagagem *RTL* do banco de registradores



1.3.6 Somador

1.3.7 Realiza a soma aritmética de 16 bits. Nas portas de entradas temos:

- ❏ Cin: Entrada para carry in caso já se tenha realizado uma soma anteriormente
- ❏ a: Entrada do valor que irá ser somado,com cadeia de 16 bits
- ❏ b: Entrada do outro valor que irá ser somado, com cadeia de 16 bits
- ❏ s: Resultado da operação
- ❏ cout: carry out da operação

Figura 1 - Declaração do somador

```
entity somador is
    port(
        cin : in std_logic;
        a,b : in std_logic_vector(15 downto 0);
        cout : out std_logic;
        s : out std_logic_vector(15 downto 0)
    );
end somador;
```

Figura 1 -Imagem *RTL* do somador

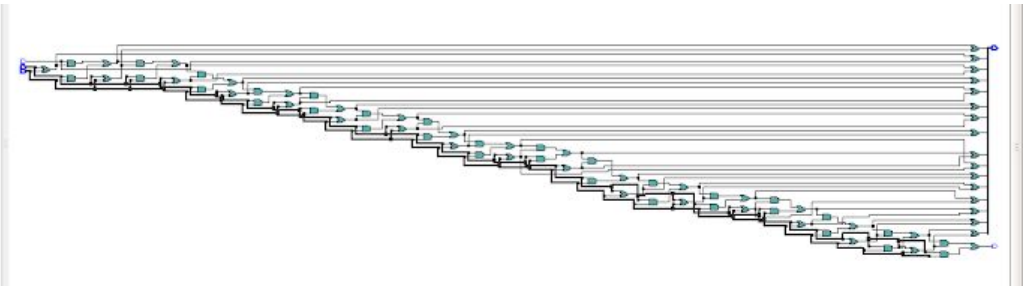
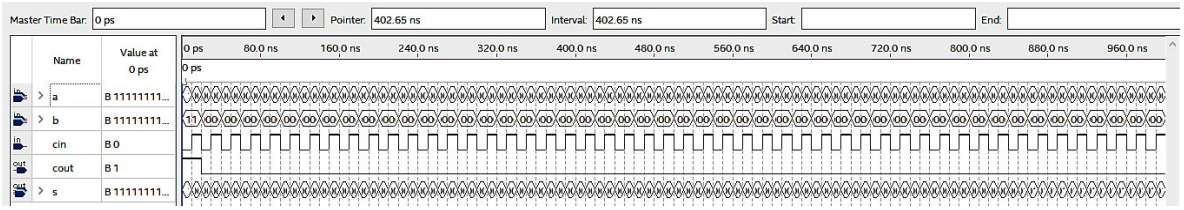
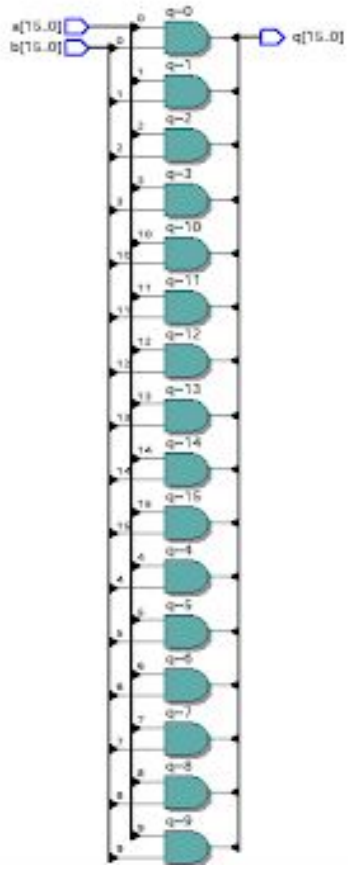


Figura 1 -Waveform do somador



1.3.8 And, Or

Componente que irá dar o resultado das operações lógicas *and,or*. Em ambos os casos temos duas portas de entrada “a”, “b”, sendo a saída “q” respectiva a operação adotada, como mostrado abaixo:



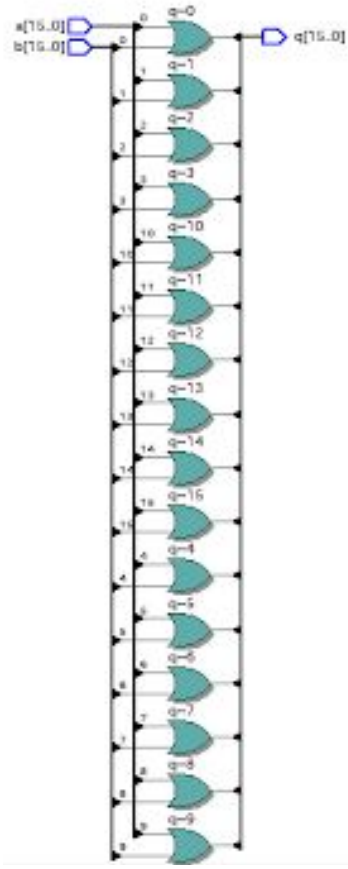


Figura 1 - Declaração do or

Figura 1 -Declaração do Or16

```
entity Or16 is
port(
  a: in std_logic_vector(15 downto 0);
  b: in std_logic_vector(15 downto 0);
  q: out std_logic_vector(15 downto 0));
end Or16;
```

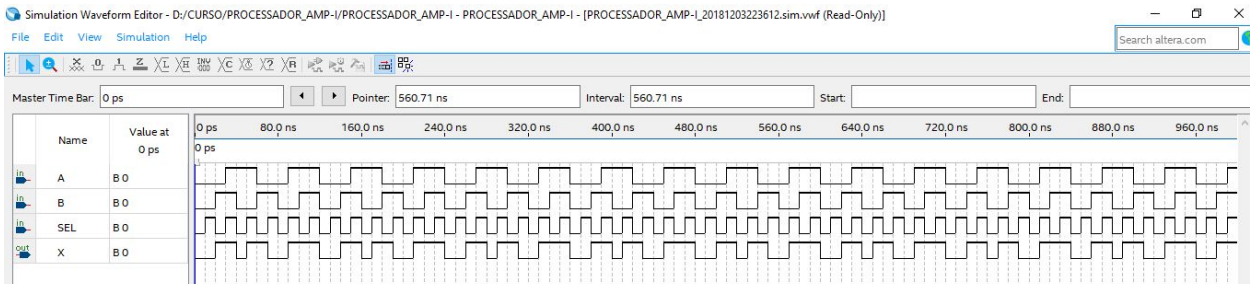
1.3.9 Mux_2x1

Multiplexador simples com duas entradas e uma saída.Neste projeto iremos utilizar quatro multiplexadores semelhantes, sendo suas declarações da seguinte maneira:

```
entity Mux_2x1 is
    Port ( SEL : in  STD_LOGIC;
           A   : in  STD_LOGIC;
           B   : in  STD_LOGIC;
           X   : out STD_LOGIC);
end Mux_2x1;
```

Nas portas temos:

- ❏ SEL: O seletor que irá coordenar a saída da escolha
- ❏ A: Entrada do valor que irá ser recebido
- ❏ B: Entrada do outro valor que irá ser recebido
- ❏ X: Resultado do multiplexador



1.3.10 Multiplicador usando algoritmo de booth

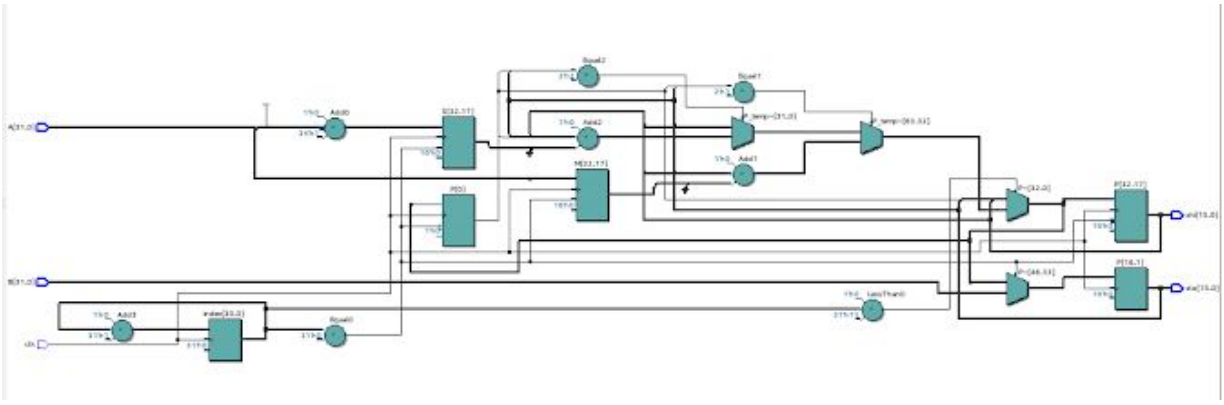
Para realizar a multiplicação dos valores na ULA, utilizamos o algoritmo de booth Sua declaração está como a seguir:

```
entity Multbooth is
    generic(
        nr_of_bits : positive := 16
        );
    port(
        clk : in  std_logic;
        A   : in  integer;
        B   : in  integer;
        zhi : out std_logic_vector(nr_of_bits-1 downto 0);
        zlo : out std_logic_vector(nr_of_bits-1 downto 0)
        );
end entity Multbooth;
```

Nas portas temos:

- clk: O clock para ajudar na checagem dos dois últimos bits para realizar o *shift* e a operação adequada
- A: Entrada do valor que irá ser multiplicado
- B: Entrada do outro valor que irá ser multiplicado
- zhi: É o resultado dos bits mais significativos
- zlo: É o resultado dos bits menos significativos

Figura 1 imagem RTL do componente Multbooth

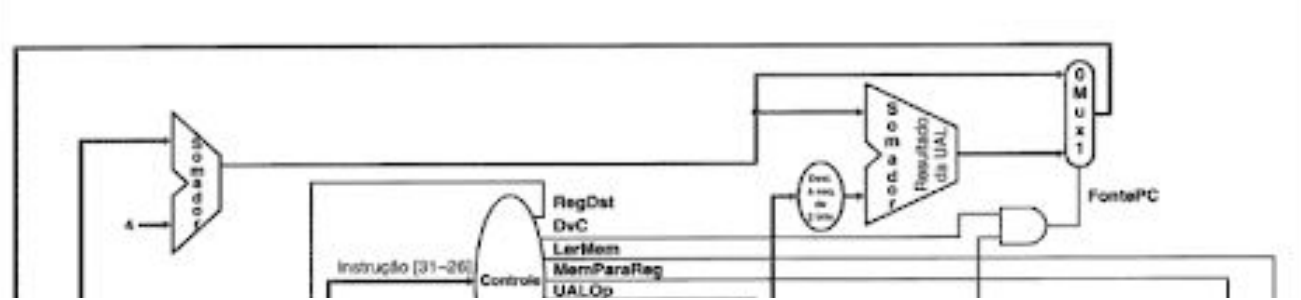


Waveform

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

Nosso objetivo muito se assemelha ao datapath do Mips e se apresente abaixo:



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 AMD-I utilizaremos como exemplo o código para calcular o número da sequência de Fibonacci. OBS: Apenas como exemplo de como seriam realizados os testes.

Tabela 3 - Código Fibonacci para o processador Quantum/EXEMPLO.

Endereço	Linguagem de Alto Nível	Binário		
		Opcode	Reg2	Reg1
			Endereço	
			Dado	
0	**LI** $S0, 0	1111	00	00
1			00000000	
2	**LI** $S3, 6	1111	00	11
3			00000110	
4	**SW** $S3, $S0	0111	00	11
5	**LI** $S1, 1	1111	00	01
6			00000001	
7	**LRT** $S2, $S1	0110	01	10
8	**LI** $S3, 3	1111	00	11
9			00000011	
10	**LW** $S0, $S0	0101	00	00
11	**CMPG** $S3,$S0	1010	00	11
12	**JMP fim**	1101	0000	
13			00011010	
14	**loop_fib: LI** $S0, 1	1111	00	00
15			00000001	
16	**ADD** $S3, $S0	0010	00	11
17	**LRT** $S0, $S2	0110	10	00
18	**ADD** $S2, $S1	0010	01	10
19	**LRT** $S1, $S0	0110	00	01
20	**LI** $S0, 0	1111	00	00
21			00000000	
22	**LW** $S0, $S0	0101	00	00
23	**CMPLE** $S3,$S0	1001	00	11
24	**JMP loop_fib**	1101	0000	
25			00001110	
26	**Fim: DEBUG** $S2, $S2	0001	10	10

OBS:Apenas exemplo, o processador não fora simulado deste modo

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

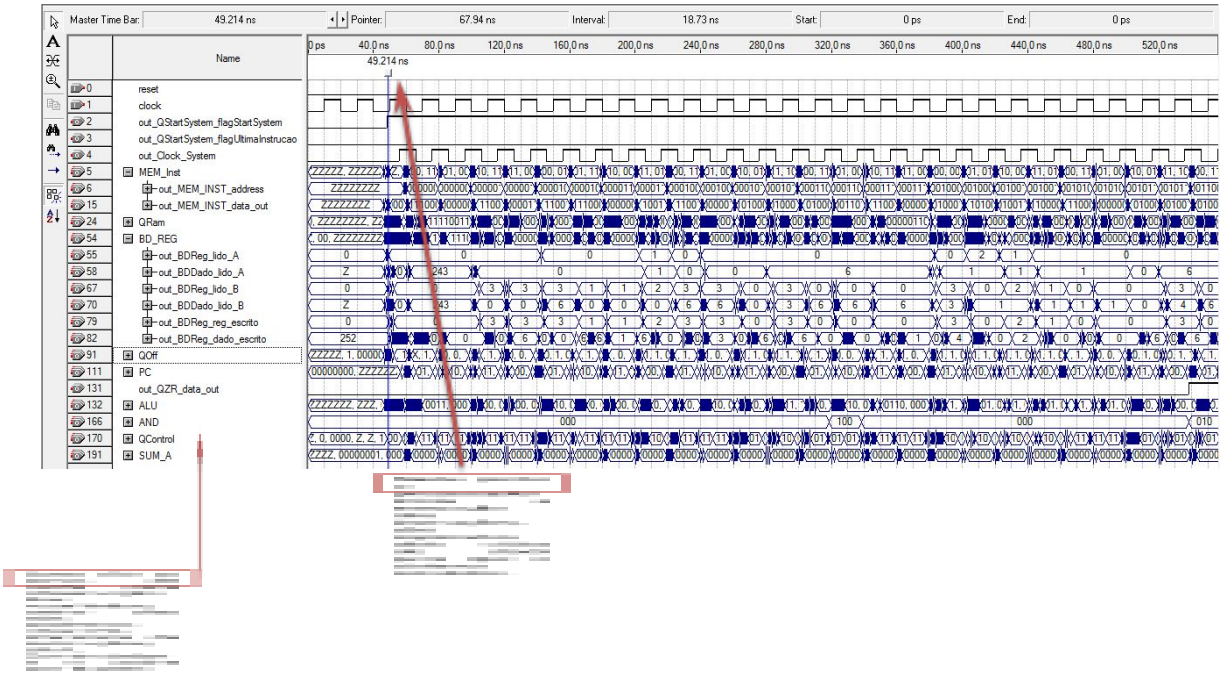


Figura 3 - Resultado na waveform.

3 Considerações finais

Este trabalho apresentou o projeto e implementação do processador de 16 bits denominado de AMP-i(Advanced Micro Processor) versão 1, apesar das dificuldades encontradas os componentes deste processador podem ser utilizados para realizar operações básicas como add,mult e and. Devido a limitação de 16 bits, decidiu-se diminuir um dos registradores mas basear sua arquitetura no processador do MIPS, sendo assim, optou-se por utilizar a arquitetura RISC onde as instruções possuem todas os mesmo tamanho. Separação entre memória de de instrução, dados e banco de registradores, contendo logicamente uma ULA(Unidade Lógica Aritmética) e uma unidade de controle responsável por organizar e definir as FLAGS utilizadas e sinalizadas de acordo com a operação necessária.

4 Bibliografia

Somador retirado do sítio
“http://www.ic.unicamp.br/~cortes/mc602/slides/VHDL/VHDL_3_MC_Circ_Arit_v1.pdf”

Multiplicador de booth retirado do sítio

“https://stackoverflow.com/questions/42630068/booth-multiplier-puts-a-1-in-the-high-32-bits-for-a-64-bit-register”

Banco de registradores

“http://labs.domipheus.com/blog/designing-a-cpu-in-vhdl-part-2-xilinx-ise-suite-register-file-testing”

Memória RAM

“http://labs.domipheus.com/blog/designing-a-cpu-in-vhdl-part-3-instruction-set-decoder-ram/”