UNIVERSIDADE DE BRASILIA

ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES

PROJETO DA DISCIPLINA:

MIPS UNICICLO em VHDL

ALUNO: RAPHAEL RODRIGUES

MATRICULA: 11/0039530

ALUNO: ULRICH KOFFI OUFFOUE

MATRICULA: 11/0089561

PROFESSOR: RICARDO PEZZUOL JACOBI

DATA DE ENTREGA: 06/02/2017

RESUMO:

Este relatório descreve a elaboração dos módulos principais da unidade operativa uniciclo MIPS utilizando a plataforma de desenvolvimento Quartus da Altera . De forma que esses módulos sejam capazes de processar conjuntos de instruções binárias de 32 bits compiladas, respeitando todas as condições de implementações necessárias para a execução desses tipos de instruções.

I/- OBJETIVOS:

Os principais objetivos desse laboratório são a familiarização com a plataforma de desenvolvimento Quartus e o desenvolvimento da capacidade de construção dos blocos para a implementação da unidade operativa uniciclo MIPS. Assim, como projeto final da disciplina de OAC (Organização e Arquitetura de Processadores) deverá ser desenvolvida uma versão do processador MIPS Uniciclo, utilizando a linguagem VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language). Através das ferramentas Quartus II e ModelSim-Altera, utilizadas para fazer os testes e simulações necessarias para verificação do reultado esperado. Assim esse laboratório complementa o conteúdo, visto em sala de aula, da disciplina Organização e Arquitetura de Computadores, a partir dele espera-se uma maior capacidade de resolução de problemas dessa categoria.

II/-INTRODUÇÃO TEORICA:

1/- VHDL

Para facilitar a criação dos módulos principais, utilizamos uma linguagem de descrição de hardware (VHDL), que é uma entre as mais utilizadas. Usa-se, geralmente, para descrever sistemas digitais, circuitos complexos de forma padrozinada, entre outros. Deste modo, podemos categorizá-la em diferentes partes:

- **Módulos**

As “caixas” em VHDL são chamadas de modulos, elas são palavras reservadas na qual o programa usa uma referencia com saídas, entradas e lógicas internas. Existem dois tipos de portas, input e output.

- **Operadores**

Os operadores são muito semelhantes aos operadores das linguagens de programação tradicionais, como C ou Java.

- **Estados de controle**

Os comandos de if, else, repeat, while, for, case, em verilog é muito parecido com C. Mas como é uma linguagem de hardware é necessário ter cuidado ao usá-las (caso contrário o desenho não poderá ser implementado em hardware).

- **Atribuição de variável**

Existem três tipos de elementos:

- Elementos Combinacionais que podem ser modelados usando atribuição de declarações
- Elementos Sequenciais podem ser modelados usando apenas declarações

- Declarações iniciais são utilizadas apenas em bancadas de testes.

2/-Somador

O somador é um bloco que tem 2 entradas de 32 bits cada. Seu funcionamento é simples, cujo o nome indica, soma os dois operandos de entradas para gerar uma saída de 32 bits.

3/-Unidade Logico-Aritmética(ULA)

A unidade lógica e aritmética (ALU – Arithmetic Logic Unit) é a base do computador, é o dispositivo que realiza as operações aritméticas ou operações lógicas. Aqui será apresentada a construção de uma ALU a partir de quatro blocos de montagem de hardware (AND, OR, inversores e multiplexadores) e ilustrada como funciona a lógica combinacional. Assim temos :

- **ALU de 1 bit**

Para uma ULA de 1 bit, a unidade lógica de 1 bit para AND, OR e adição pode se representar pela figura seguinte:

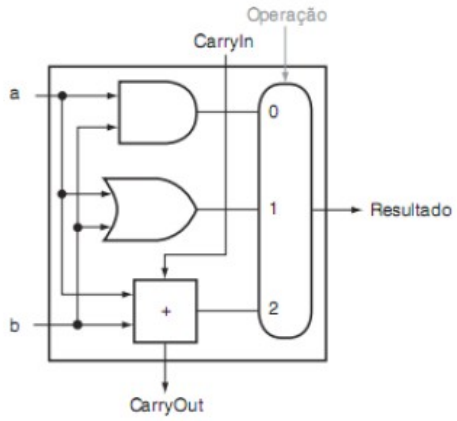


Figura 1: ULA de 1 bit de saída

- **ALU de 32 bits**

Sabendo que uma word possui 32 bits de tamanho, pode-se basear sobre o projeto da ULA de 1 bit, para criar a nossa ULA de 32 bits. Para isso, deve-se replicar e conectar as ULAs, de 1 bit, para formar a de 32 bits, ilustrado pela figura seguinte:

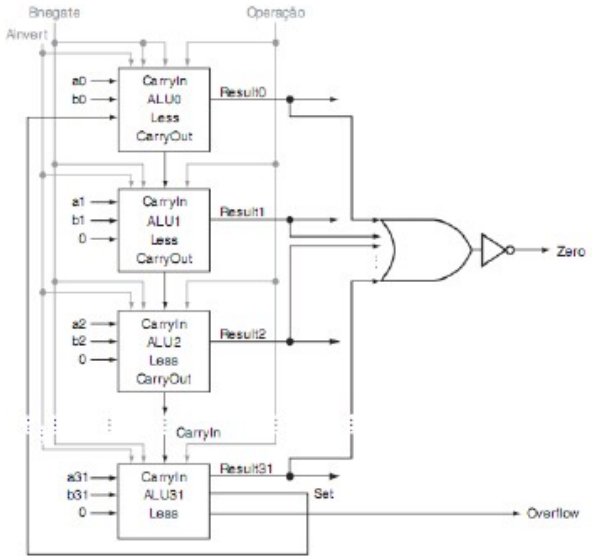


Figura 2: ULA de 32 bit de saída

4/-Multiplexador

É um dispositivo que seleciona as informações de duas ou mais fontes de dados num único canal. Segue abaixo um exemplo de multiplexador de 4 entradas:

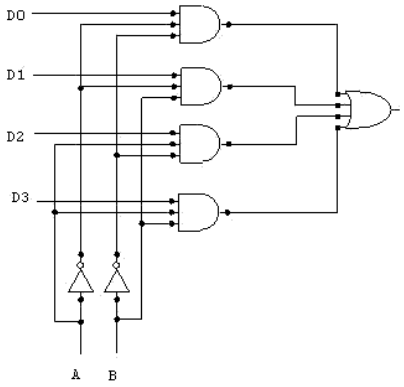


Figura 3: Multiplexador de 4 entradas

5/-Program counter (PC)

O registrador chamado *contador de programa* é usado para contar o endereço da instrução atual. Ele se atualiza a cada subida de relógio para armazenar um novo valor de endereço, podendo, por exemplo, receber PC+4 (valor correspondente apontando para proxima instrução) ou outros valores dependendo da instrução realizada.

6/-Memoria de instruções (MI)

A memória de instruções armazena apenas instruções. Cada instrução no MIPS ocupa exatamente 32 bits, assim, uma instrução ocupa 4 posições de memória. A cada leitura da memória de instruções é buscada não apenas o byte cujo endereço está no barramento de endereço, mas este e os três seguintes. Uma outra limitação imposta para facilitar o acesso à memória de instruções é alinhar instruções em uma fronteira inteira de palavra. Esta expressão significa que cada instrução da MIPS só pode começar a partir de um endereço múltiplo de 4, que correspondem àqueles cujos 2 últimos bits são 00.

7/-Banco de registradores (BREG)

Consiste em um conjunto de registradores que podem ser lidos e escritos fornecendo um número de registrador a ser acessado. Ele pode ser implementado com um decodificador para cada porta de leitura ou escrita e um array de registradores a partir de flip-flops tipo D. Para escrever em um registrador são necessárias 3 entradas (números dos registradores, dados a escrever e um clock que controla a escrita), como ilustrado na figura abaixo:

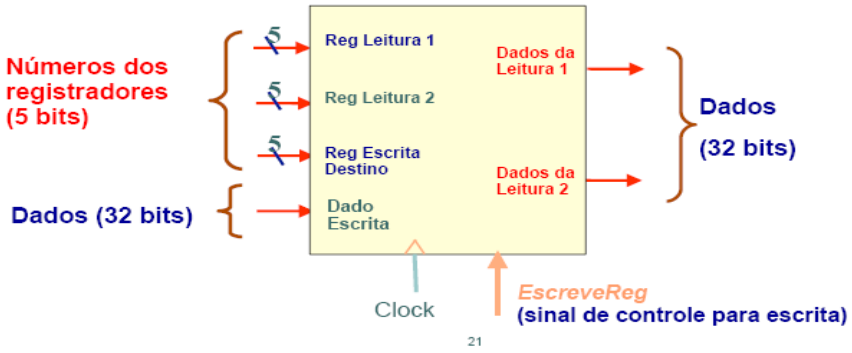


Figura 4: Banco de registradores

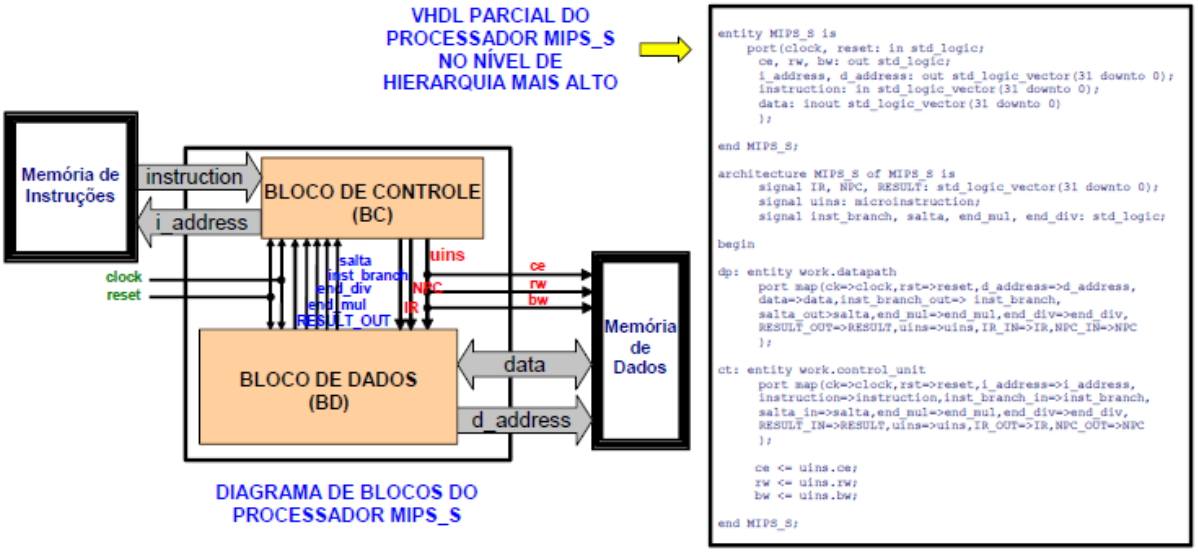
Assim os registradores da Arquitetura MIPS, são ordenados e organizados cada um pelo índice e nomes, como a seguir:

Nome	Número (hexa/binário)	Nome Alternativo	Significado ou Convenção de Utilização
$0	00 / 00000	$zero	constante 0
$1	01 / 00001	$at	reservado para o programa montador
$2	02 / 00010	$v0	resultado de função
$3	03 / 00011	$v1	resultado de função
$4	04 / 00100	$a0	argumento para função
$5	05 / 00101	$a1	argumento para função
$6	06 / 00110	$a2	argumento para função
$7	07 / 00111	$a3	argumento para função
$8	08 / 01000	$t0	temporário
$9	09 / 01001	$t1	temporário
$10	0A / 01010	$t2	temporário
$11	0B / 01011	$t3	temporário
$12	0C / 01100	$t4	temporário
$13	0D / 01101	$t5	temporário
$14	0E / 01110	$t6	temporário
$15	0F / 01111	$t7	temporário
$16	10 / 10000	$s0	temporário (salvo nas chamadas de função/subrotina)
$17	11 / 10001	$s1	temporário (salvo nas chamadas de função/subrotina)
$18	12 / 10010	$s2	temporário (salvo nas chamadas de função/subrotina)
$19	13 / 10011	$s3	temporário (salvo nas chamadas de função/subrotina)
$20	14 / 10100	$s4	temporário (salvo nas chamadas de função/subrotina)
$21	15 / 10101	$s5	temporário (salvo nas chamadas de função/subrotina)
$22	16 / 10110	$s6	temporário (salvo nas chamadas de função/subrotina)
$23	17 / 10111	$s7	temporário (salvo nas chamadas de função/subrotina)
$24	18 / 11000	$t8	temporário
$25	19 / 11001	$t9	temporário
$26	1A / 11010	$k0	reservado para o SO
$27	1B / 11011	$k1	reservado para o SO
$28	1C / 11100	$gp	apontador de área global
$29	1D / 11101	$sp	stack pointer
$30	1E / 11110	$fp	frame pointer
$31	1F / 11111	$ra	armazena endereço de retorno de subrotinas

Tabela 1: Ordem dos registradores do MIPS

8/-Memoria de dados(MD)

A memória de dados possui uma interface um pouco mais complexa do que a da instrução, devido ao fato de poder ser lida ou escrita. Além disto, os dados lidos ou gravados nesta memória podem ter tamanho diversos. Estas necessidades fazem com que o acesso à informação tenha um formato mais flexível. O barramento de dados é fixo, sendo de 32 bits. Lê-se sempre 4 bytes (ou posições) de memória de cada vez. O que é lido pode ser, por exemplo, 1 inteiro ou 4 caracteres ou metade de um número em ponto flutuante de precisão dupla. Como a leitura é alinhada em qualquer fronteira, é até mesmo possível ler metade de um inteiro e dois caracteres nos 32 bits. Isso tudo é possivel através do sinal de escrita e de leitura vindo do controle principal.



(a) Diagrama de blocos processador-memória para a organização MIPS_S.

(b) VHDL parcial do nível mais alto da hierarquia de descrição do processador MIPS_S.

Figura 5: Banco de Registradores

9/-Extensão de sinal

O módulo de extensão de sinal consiste em extender os 16 bits do valor de imediato, contido nas instruções tipo I, para 32 bits, mantendo o sinal. Se for negativo, os 16 bits superiores serão iguais a 1, caso contrário, os 16 bits superiores serão iguais a 0, como ilustrado a figura a seguir:

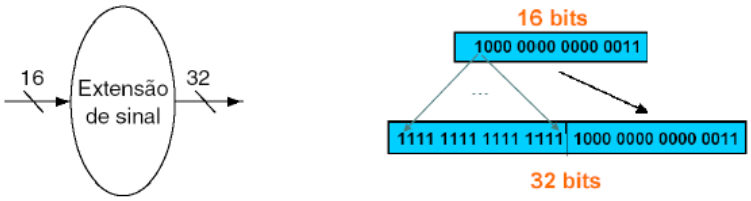


Figura 6: Extensão de sinal

10/- Controles

10-1 CONTROLE PRINCIPAL

A unidade de controle deve, a partir do código da instrução (opcode), fornecer os sinais que realizam as instruções na unidade operativa. Assim, para o início do projeto do controle principal é essencial examinar os formatos das três classes de instrução: tipo R, de desvio e de acesso a memória:

- TIPO R:

0	rs	rt	rd	shamt	funct
31:26	25:21	20:16	15:11	10:6	5:0

Figura 7: Instrução tipo R

- DESVIO:

35ou43	rs	rt	address
31:26	25:21	20:16	15:0

Figura 8: Instrução tipo I

- ACESSO A MEMORIA:

4	rs	rt	address
31:26	25:21	20:16	15:0

Figura 9: Instrução tipo I

Lembrando que:

- O campo op está sempre contido nos bits 31:26, então será chamado Op[5:0].
- Os dois registradores a serem lidos são sempre especificados pelos campos rs e rt, nas posições 25:21 e 20:16, consecutivamente.
- O registrador de base para as instruções load e store está sempre nas posições de bit 25:21 (rs).
- O offset de 16 bits para branch equal, load e store está sempre nas posições 15:0.
- O registrador de destino está em um de dois lugares. Para um load, está em 20:16 (rt), enquanto para uma instrução tipo R, ele está nas posições 15:11 (rd). Portanto, precisa-se incluir um mux para selecionar que campo da instrução será usado para indicar o número de registrador a ser escrito.

São necessárias sete linhas de controle de um únco bit mais o sinal de controle OpALU de 2 bits. Na tabela a seguir têm-se descritos os bits necessários, relacionando para cada tipo de instrução realizada:

Controle	Nome do sinal	formato R	lw	sw	beq
Entradas	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Saídas	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemparaReg	0	1	X	X
	EscreveReg	1	1	0	0
	LeMem	0	1	0	0
	EscreveMem	0	0	1	0
	Branch	0	0	0	1
	OpALU1	1	0	0	0
	OpALU0	0	0	0	1

Tabela 2: Entrada e saída da unidade de controle

10-2 CONTROLE DA ULA

O controle da Ula recebe 6 bits do campo *funct*, sendo 2 bits do opALU e 4 bits de controle das operações que irão ocorrer na ULA. Sendo esses 2 bits de entrada:

- “00” para acesso a memória(lw, sw), fazendo uma soma(add) na ula.
- “01” para desvio(Beq, Bne), fazendo uma subrtração(sub)na ula.
- “10” para operações lógico-aritméticas(add, sub, and, or, slt).

E 4 bits de tipos de operações a serem realizadas pela ULA:

Linhas de Controle ULA	Função
0000	AND
0001	OR
0010	ADD
0110	SUB
0111	SLT
1100	NOR

Tabela 3: Operações de controle

III/-MATERIAIS E MÉTODOS:

Observação: Para facilitar a visualização, criamos um arquivo chamado *Processador-Modulos e fios.png* que se encontra na pasta raiz do trabalho, em que nele está descrito todos os módulos com as entradas e saídas de acordo com as utilizadas no código. Como também a citação de todos os nomes de todos os fios utilizados no projeto.

III-1/Materiais:

Os materiais usados para esse laboratório foram:

- O software: Quartus 10.1 Web edition (64 bits).
- ModelSim-Altera 6.6c (Quartus II 10.1) Starter Edition.

III-2/Requisitos:

Devemos implementar, utilizando o ambiente de desenvolvimento Quartus, por meio da linguagem VHDL, um conjunto de módulos, dentre os quais descritos acima e outros, formando uma unidade operativa da arquitetura Uniciclo MIPS, como mostrado na figura abaixo:

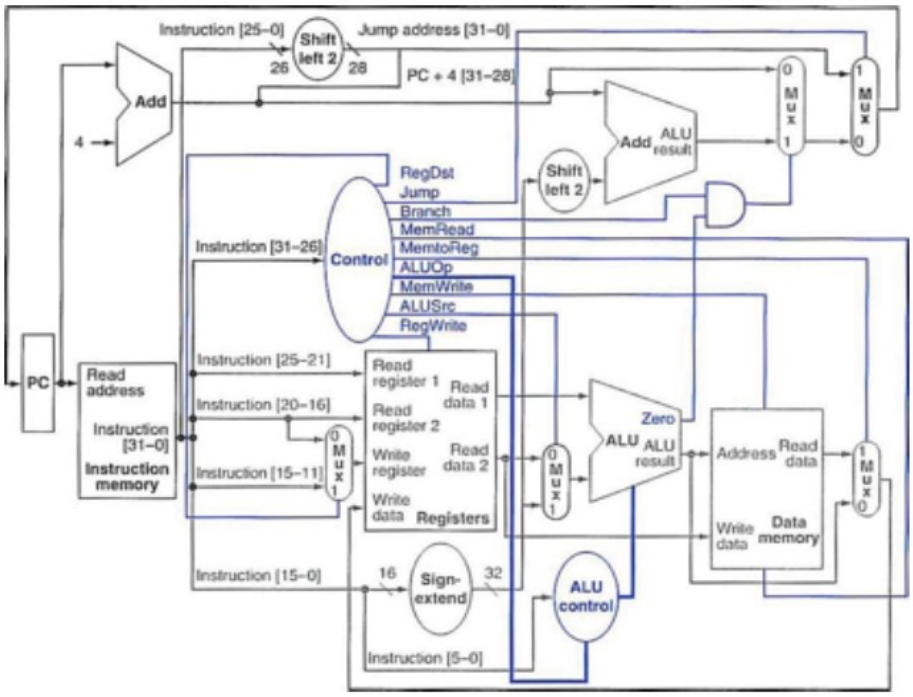


Figura 10: Processador Uniciclo.

De forma que a parte operativa do MIPS seja de 32 bits, ou seja, todos os dados armazenados em memória, os registradores, as instruções e as conexões ambas utilizam 32 bits de tamanho. Para se relizar essa implementação, os módulos principais necessarios serão:

- **PC**: é um registrador de 32 bits. Entretanto, pelas restrições de memória adotadas, apenas o número de bits necessário deve ser enviado à memória de instruções, sendo o restante ignorado.

- **Memória de Instruções (MI)**: armazena o código a ser executado. As instruções são de 32 bits. O espaço de endereçamento é reduzido (ex: 8 bits). Cada endereço da MI armazena uma instrução de 32 bits. Idealmente, a MI deve funcionar como um bloco combinacional neste projeto, ou seja, necessita-se apenas do endereço para ler a instrução, sem sinais adicionais de controle. Essa memória não permite o endereçamento a byte. Desta forma, se forem utilizados 8 bits de endereço, deve-se utilizar os bits 2 a 9 do PC como endereço de instrução.

- **Banco de Registradores (BREG)**: é constituído por 32 registradores de 32 bits. O registrador de índice zero, breg[0], é uma constante. Sua leitura retorna sempre zero, e não pode ser escrito. O BREG tem duas entradas de endereços, permitindo a leitura de 2 registradores de forma simultânea. Uma terceira entrada de endereço é utilizada para selecionar um registrador para escrita de dados. A escrita de um dado em registrador ocorre na transição de subida do relógio.

- **Unidade Lógico-Aritmética (ULA)**: opera sobre dados de 32 bits. Provê o resultado em 32 bits, juntamente com o sinal **ZERO**, que indica que o resultado da operação realizada é zero (utilizada no branch).

✴ Operações implementadas na ULA:
ADD, SUB, AND, OR, XOR, SLT, NOR, SLL, SRL, SRA

- **Memória de Dados (MD)**: armazena os dados do programa. Pode ser lida ou escrita. Neste projeto, a MD fornece apenas palavras de 32 bits quando lida. Considerando que a MD é reduzida, deve-se selecionar apenas o número necessário de bits de endereço (8 bits, se a memória comportar 256 palavras). Os bits de endereço selecionados para acessar a MD devem permitir a leitura do segmento de dados conforme o modelo de memória compacto do MARS, onde o endereço base é 0x00002000. A memória é escrita na subida do relógio, quando o sinal de controle EscreveMem estiver acionado. O sinal de leitura LerMem faz com que o conteúdo da posição de memória endereçada seja colocado na saída de dados.

- **Multiplexadores 2 para 1**: são utilizados 4 multiplexadores com 2 entradas de 32 bits e uma saída de 32 bits.

- **Somadores**: são utilizados 2 somadores de 32 bits para operar com endereços.

Obs: para as instruções de deslocamento, deve-se utilizar as funções de deslocamento disponíveis no pacote *numeric_std* : *shift_left* e *shift_right*, com o tipo apropriado de dados para o deslocamento lógico/aritmético.

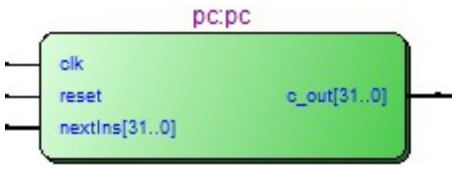
Lembrando que as instruções a serem implementadas são as seguintes: LW, SW, ADD, SUB, AND, OR, NOR, XOR, SLT, ADDI, SLL, SRL, SRA, J,BEQ, BNE

Será usado como teste um programa em assembly disponibilizado pelo professor, para verificação do correto funcionamento e execução do processador implementado.

III-3/Métodos:

Para realização da implementação da unidade Operativa Uniciclo por VHDL, criamos os seguintes blocos:

III-3-1/ PC



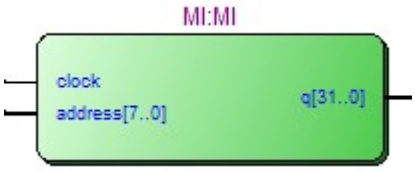
O PC, tem ,na teoria, uma entrada e uma saída, que carrega o endereço das instruções atuais, assim como o endereço da próxima instrução. Para se fazer criamos o módulo da figura acima, mapeando as entradas e saídas setadas a cada subida do Clock, e o reset permitindo atualizar o clock. Assim geramos o as entradas e saídas do bloco através da implementação:

```
Port (
		clk			: in STD_LOGIC;
		nextIns			: in STD_LOGIC_VECTOR (31 downto 0);
		reset			: in std_logic;
		c_out			: out STD_LOGIC_VECTOR (31 downto 0)
	);
end pc;
```

Cujo o codigo VHDL do funcionamento esta a seguir:

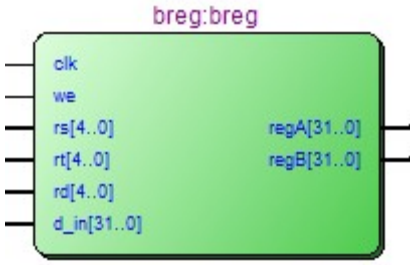
```
architecture Behavioral of pc is
SIGNAL ExitPC: std_logic_vector(31 DOWNTO 0) := (OTHERS => '0');
BEGIN
	c_out <= ExitPC;
	process(clk)
	begin
		if(rising_edge(clk)) then
			if(reset = '1') then
				ExitPC <= nextIns;
			end if;
		end if;
	end process;
end Behavioral;
```

III-3-2/ Memoria de instruções (MI)



A memória de instruções foi feita através do recurso do quartus do "megafunction wizard", que permite criar um bloco com duas entradas e uma saída ativada por um clock, que, no nosso caso, será o clock da memória, o *clock_mem*.

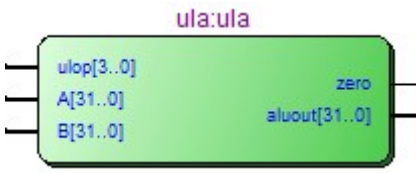
III-3-3/ Banco de registradores(BREG)



O banco de registradores foi criado com 6 entradas, em que a primeira delas, o *clk*, como mostrado na figura acima, que seta os índices dos registradores a serem escolhidos, a entrada *we*, *Write Enable*, como sinal para escrita, as entradas rs, rt e rd como entradas dos registradoes e a entrada *d_in*, para o dado a ser escrito no rt ou rd, dependendo da instrução. Assim como 2 saídas que representam os valores de 32 bits de cada um dos operandos em reg A e reg B. O código VHDL da criação das portas pode ser visto a seguir:

```
	component breg generic ( WSIZE : natural := 32; ISIZE : natural := 5; BREGSIZE : natural := 32 );

		port(

			clk		: in std_logic;

			we		: in std_logic;

			rs		: in std_logic_vector(ISIZE-1 downto 0);

			rt		: in std_logic_vector(ISIZE-1 downto 0);

			rd		: in std_logic_vector(ISIZE-1 downto 0);

			d_in		: in std_logic_vector(WSIZE-1 downto 0);

			regA		: out std_logic_vector(WSIZE-1 downto 0);

			regB		: out std_logic_vector(WSIZE-1 downto 0)

			);

		end component;
```

III-3-4/ Unidade Lógico-Aritmética (ULA)

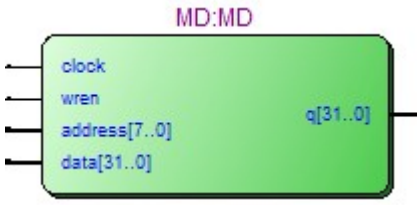


A ULA recebe os operandos do banco de registradores depois de passar por diferentes mux (dependendo da instrução desejada), para realizar as operações

necessárias (“AND”,”OR”,”ADD”,”SUB”,”SLT”,”NOR”,”XOR”,”SLL”,”SRL”,”SRA”) para execução da instrução, dependendo do código de sinal de controle da ULA “ulop”, vindo, no nosso caso, pelo controle principal. Assim descrevemos as entradas e saídas de acordo com a figura acima pelo seguinte código VHDL:

```
component ula
        generic ( WSIZE : natural := 32);
                port (
                ulop            : in std_logic_vector(3 downto 0);
                A, B            : in std_logic_vector(WSIZE-1 downto 0);
                aluout          : out std_logic_vector(WSIZE-1 downto 0);
                zero            : out std_logic
                );
end component;
```

III-3-5/ Memória de Dados (MD):

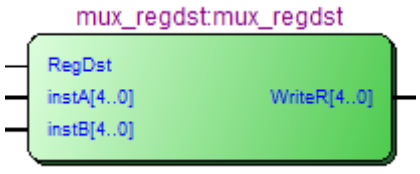


Pelo meio do “megafunction wizard” foi criado uma memória RAM de 4 entradas e uma saída, cujo duas das entradas (*address* e *data*) recebem o endereço e o dado a ser lido ou escrito na memória (dependendo da instrução), uma para o sinal de controle e outra para o clock. Tendo a saída como o valor lido.

III-3-6/ Multiplexadores:

Nesse projeto uniciclo foram criados vários multiplexadores , cujo alguns serão descritos como adicionais. Aqui serão mostrados os multiplexadores principais. Temos portanto:

- Multiplexador de registrador destino

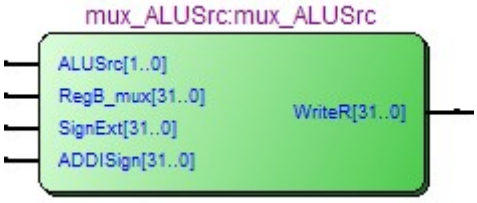


Esse multiplexador que nomeamos de *mux_regdest* é de 2 entradas de dados e uma saída, para escolher o tipo de operação e o registador para escrita necessário,

podendo ser do tipo R (caso for setado para ‘1’) ou tipo I (quando setado para ‘0’). Temos o seguinte código :

```
entity mux_regdst is

	Port (
		RegDst  : in  STD_LOGIC;
		instA   : in  STD_LOGIC_VECTOR (4 downto 0);
		instB   : in  STD_LOGIC_VECTOR (4 downto 0);
		WriteR  : out STD_LOGIC_VECTOR (4 downto 0)
	);

end mux_regdst;

architecture Behavioral of mux_regdst is

begin
    WriteR <= instB when (RegDst = '1') else InstA;

end Behavioral;
```

- Multiplexador de operação na ULA

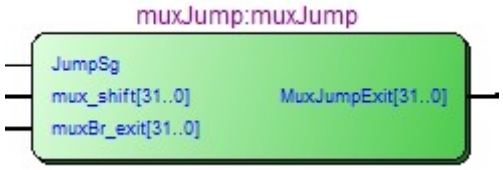


Chamado de *mux_Alusrc*, ele permite escolher umas das suas 3 entradas para poder escolher que tipo de operação será feita e qual sera o segundo operando a ser recebido na ULA , para eventual operação. Quando ativado em ”00” ele recebe os 32 bits da saida do banco registrador (Tipo R ou beq, bne), em “01”, faz escolhe o sinal estendido para instrução do tipo I, e enfim em “10” para instrução logica aritmética com immediato(addi, subi,...). Assim implementação da seguinte maneira:

```
entity mux_ALUSrc is

	Port (
		ALUSrc		: in  STD_LOGIC_VECTOR (1 downto 0);
		RegB_mux	: in  STD_LOGIC_VECTOR (31 downto 0);
		SignExt		: in  STD_LOGIC_VECTOR (31 downto 0);
		ADDISign	: in  STD_LOGIC_VECTOR (31 downto 0);
		WriteR		: out STD_LOGIC_VECTOR (31 downto 0)
	);

end mux_ALUSrc;

architecture Behavioral of mux_ALUSrc is

begin

	process (ALUSrc, SignExt, RegB_mux, ADDISign)
	begin
		if (ALUSrc = "01") then
			WriteR <= SignExt;
		elsif (ALUSrc = "00") then
			WriteR <= RegB_mux;
```

```
                elsif (ALUSrc = "10") then
                        WriteR <=  ADDISign;
                end if;
         end process;

end Behavioral;
```

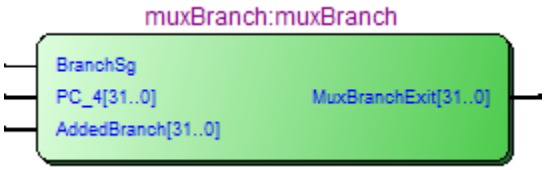
- Multiplexador do jump



O *muxJump* tem 3 entradas recebendo o valor do PC+4(endereço da proxima instrução) e o valor do endereço de salto incondiconal “jump” e o sinal de controle”jumpsg” para ativação do mux. Assim ao ativar o sinal de controle respectivo em “0” teremos como saida o PC +4, em “1” teremos como saida o endereço de salto atribuido ao PC. Como descrito a seguir:

```
entity muxJump is

        Port (
                JumpSg                  : in  STD_LOGIC;
                mux_shift               : in  STD_LOGIC_VECTOR (31 downto 0);
                muxBr_exit              : in  STD_LOGIC_VECTOR (31 downto 0);
                MuxJumpExit             : out STD_LOGIC_VECTOR (31 downto 0)
        );

end muxJump;

architecture Behavioral of muxJump is

begin
    MuxJumpExit <= muxBr_exit when (JumpSg = '0') else mux_shift;

end Behavioral;
```

- Multiplexador do branch

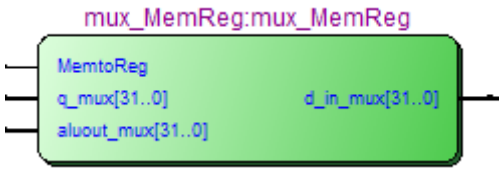


É um multiplexador de 3 entradas , cujo uma para PC+4, outra para endereço do desvio condicional e uma ultima para o sinal de controle do branch(Branchsg. Assim podemos ter:

```
entity muxBranch is

        Port (
                BranchSg        : in  STD_LOGIC;
                PC_4            : in  STD_LOGIC_VECTOR (31 downto 0);
                AddedBranch     : in  STD_LOGIC_VECTOR (31 downto 0);
```

```
                MuxBranchExit  : out STD_LOGIC_VECTOR (31 downto 0)
        );

end muxBranch;

architecture Behavioral of muxBranch is

begin
    MuxBranchExit <= AddedBranch when (BranchSg = '1') else PC_4;
end Behavioral;
```

- Multiplexador da memória para o banco de registradores



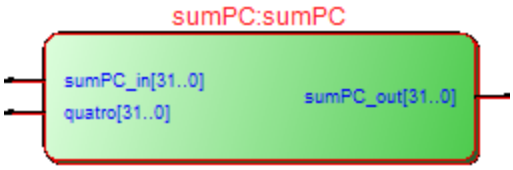
Este Multiplexador escolhe dentro das duas entradas de dados se irá escrever o dado vindo da memória (ativando “1”) ou da saída da ULA (ativando ”0”) no banco de registradores:

```
entity mux_MemReg is

        Port (
                MemtoReg              : in  STD_LOGIC;
                q_mux                 : in  STD_LOGIC_VECTOR (31 downto 0);
                aluout_mux            : in  STD_LOGIC_VECTOR (31 downto 0);
                d_in_mux              : out STD_LOGIC_VECTOR (31 downto 0)
        );

end mux_MemReg;

architecture Behavioral of mux_MemReg is

begin
    d_in_mux <= aluout_mux when (MemtoReg = '0') else q_mux;

end Behavioral;
```

III-3-7/ Somadores:

Foram criados dois somadores de acordo com o desejado, um nomeado sumPC, cujo a função é de receber o dado do PC somando a 4, para endereçar a proxima instrução caso for uma operação sem nenhum tipo de desvio. O componente foi criado a partir do seguinte código :

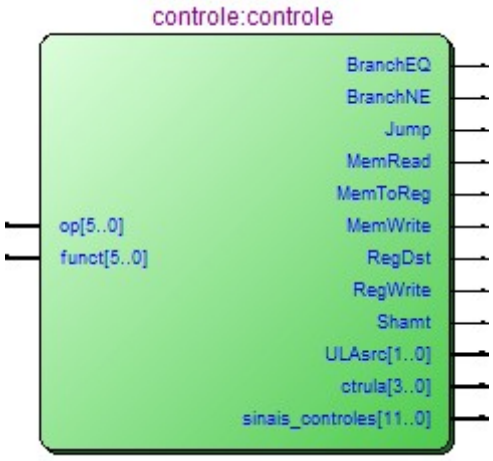
```
        component sumPC

port (

        sumPC_in              : in std_logic_vector(31 downto 0);

        quatro                : in std_logic_vector(31 downto 0);

        sumPC_out             : out std_logic_vector(31 downto 0)

);

end component;
```



O segundo Somador nomeado sumBranch, cujo a função é de calcular o endereço do salto sendo PC +4 + offset*2(bne ou beq) quando for uma instrução que executa um desvio condicional , ou sendo concatenação dos bits mais significativo do PC +4 (previamente calculado) com os bits [25-0] deslocado de 2 bits da instrução, vindo da memoria de instrução , no caso de um desvio incondicional(Jump). Temos a seguinte declaração em VHDL:

```
component sumBranch
        port (
                sumBranch_inA          : in std_logic_vector(31 downto 0);
                sumBranch_inB          : in std_logic_vector(31 downto 0);
                sumBranch_out          : out std_logic_vector(31 downto 0)
        );
        end component;
```

III-3-8/ Sinais de controles:



Os sinais de controle definem quais blocos ou multiplxadores devem ser ativados, ou não, dependendo do campo do opcode[31-26] e funct[5-0]. Assim, cada instrução tipo de instrução poderá ativar módulos específicos . No nosso caso tivemos 11 bits de controles, ordenados como um array de bits controles. Assim temos o seguinte componente descrito em VHDL:

```
entity controle is

	port(

		op			: in std_logic_vector(5 downto 0);

		funct			: in std_logic_vector(5 downto 0);

		RegDst			: out std_logic;

		ULAsrc			: out std_logic_vector(1 downto 0);

		MemToReg		: out std_logic;

		RegWrite		: out std_logic;

		MemRead			: out std_logic;

		MemWrite		: out std_logic;

		Jump			: out std_logic;

		BranchEQ		: out std_logic;

		BranchNE		: out std_logic;

		Shamt			: out std_logic;

		ctrula			: out std_logic_vector (3 downto 0);

		sinais_controles	: out std_logic_vector(11 downto 0)

	);

end controle;
```

Com o seguinte código VHDL descrevendo o funcionamento do controle:

```
architecture Behavioral of controle is

	signal controles	: std_logic_vector(11 downto 0);

	signal ctrulaS		: std_logic_vector (3 downto 0);

	signal ULAop		: std_logic_vector (1 downto 0);

	signal clk		: std_logic;

	signal ulaOP1		: std_logic;

	signal ulaOP2		: std_logic;

	signal ulasrc1		: std_logic;

	signal ulasrc2		: std_logic;

	-- controles(Shamt, branchNE, regDst, ulasrc1, memToReg, regWrite, memread,
memWrite, branchEQ, ulactrl(2 e 1), jump)


	begin
```

```
		process (funct, controles, op)
		begin
			if ((funct = "000000") or (funct = "000010") or (funct = "000011")) then -- SLL, SRL, SRA
				if op = "000000" then -- TIPO R
					controles <="101001000100";
				elsif op = "001000" then -- ADDI
					controles <="000101000110";
				elsif op = "100011" then -- LW
					controles <="000111100000";
				elsif op = "101011" then -- SW
					controles <="000100010000";
				elsif op = "000100" then -- BEQ
					controles <="000000001010";
				elsif op = "000101" then -- BNE
					controles <="010000000010";
				elsif op = "000010" then -- JUMP
					controles <="000000000001";
				end if;
			else
				if op = "000000" then -- TIPO R
					controles <="001001000100";
				elsif op = "001000" then  -- ADDI
					controles <="000101000110";
				elsif op = "100011" then -- LW
					controles <="000111100000";
				elsif op = "101011" then -- SW
					controles <="000100010000";
				elsif op = "000100" then -- BEQ
					controles <="000000001010";
				elsif op = "000101" then -- BNE
					controles <="010000000010";
				elsif op = "000010" then --JUMP
					controles <="000000000001";
				end if;
			end if;
```

```
end process;

ulaOP1  <= controles(2);
ulaOP2  <= controles(1);
ULAop   <= ulaOP1 & ulaOP2;

ctrulaS <=
        "0000" when ULAop = "00" else -- lw sw
        "0000" when ULAop = "11" else -- addi
        "0001" when ULAop = "01" else -- beq bne
        "0000" when ULAop = "10" and funct = "100000" else -- add
        "0001" when ULAop = "10" and funct = "100010" else -- sub
        "0010" when ULAop = "10" and funct = "100100" else -- and
        "0011" when ULAop = "10" and funct = "100101" else -- or
        "1001" when ULAop = "10" and funct = "100111" else -- nor
        "0101" when ULAop = "10" and funct = "100110" else -- xor
        "0110" when ULAop = "10" and funct = "000000" else -- sll
        "0111" when ULAop = "10" and funct = "000010" else -- srl
        "1110" when ULAop = "10" and funct = "000011" else -- sra
        "1111" when ULAop = "10" and funct = "101010" else -- slt
         "----";

        process (ULAop)
        begin
                if  ULAop = "01" or ULAop = "10" then
                        ULAsrc <= "00";
                        elsif ULAop = "00" then
                        ULAsrc <= "01";
                        elsIF ULAop = "11" then
                        ULAsrc <= "10";

                end if;
        end process;

Shamt <= controles(11);
BranchNE <= controles(10);
```

```
	RegDst  <= controles(9);

	ulasrc1  <= controles(8);

	MemToReg <= controles(7);

	RegWrite <= controles(6);

	MemRead  <= controles(5);

	MemWrite <= controles(4);

	BranchEQ <= controles(3);

	Jump     <= controles(0);


	ctrula  <= ctrulaS;


	sinais_controles <= controles(0) & controles(1) & controles(2) & controles(3) & controles(4) & controles(5) & controles(6) & controles(7) & controles(8) & controles(9) & controles(10) & controles (11);

end Behavioral;
```

III-3-9/ Blocos adicionais:

Os blocos adicionais representam todos os blocos que foram criados alem das especificações de projeto para uma melhor avaliação dos resultados e para implementar instruções adicionais. Temos portanto:

- Sign Extend

O sinal extendido extende o bit de sinal dos 16 bits de entrada para 32 bits, como descrito no seguinte código VHDL:

```
component signExtd
	port(
		IMM16			: in std_logic_vector(15 downto 0);
		IMM32			: out std_logic_vector(31 downto 0)
	);
	end component;
architecture Behavioral of  signExtd is

begin
	process (IMM16)
	begin
		if IMM16(15) = '0' then
			IMM32 <= "0000000000000000" & IMM16;
		else
			IMM32 <= "1111111111111111" & IMM16;
		end if;
	end process;
end architecture;
```

- Portas and(BEQ)

Aqui descrevemos a porta AND necessária para ativar o mux para o calculo do desvio. Tendo como entradas a saida da ULA"Zero"(beq ou bne) e o sinal de controle do beq(branchControl), como descrito no componente criado em VHDL:

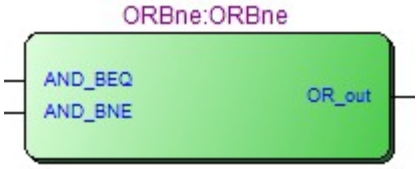
```
component andPort
	port (
		branchControl		: in  STD_LOGIC;
		zeroULA			: in  STD_LOGIC;
		and_out			: out STD_LOGIC
	);
	end component;
architecture Behavioral of andPort is

begin
    and_out <= branchControl and zeroULA;

end Behavioral;
```

- Portas and(BNE)

A porta AND do bne fazendo a mesma função do que a porta and descrita anteriormente, com um diferencial de ter como entrada um sinal de controle para o bne(bneControl):

```
entity ANDBne is

	Port (

		bneControl		: in  STD_LOGIC;

		zeroULABne		: in  STD_LOGIC;

		andBne_out		: out STD_LOGIC

	);

	end ANDBne;


architecture Behavioral of ANDBne is

begin

    andBne_out <= (bneControl and (not zeroULABne));

end Behavioral;
```

- Portas Or(BNE)



Tivemos que criar uma porta Or para os dois sinais de controle bnecontrol e beq controle para setar a saida do mux com a entrada do endereço de desvio, caso for uma operação do tipo branch:

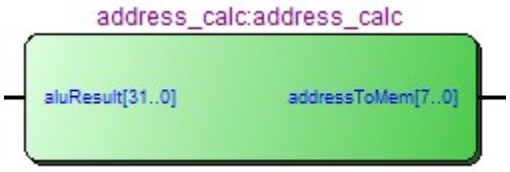
```
entity ORBne is
	Port (
		AND_BEQ		: in  STD_LOGIC;
		AND_BNE		: in  STD_LOGIC;
		OR_out		: out STD_LOGIC
	);

end ORBne;

architecture Behavioral of ORBne is

begin
    OR_out <= (AND_BEQ or AND_BNE);

end Behavioral;
```

- Modulo de concatenação do shamt

Esse módulo concatena os bits ‘0’ a esquerda dos 5 bits do shamt para dar 32 bits e assim poder ser usado pela ULA para executar uma instrução sll, srl ou ainda sra.

```
entity concatShamt is
	port(
		inShamt: in std_logic_vector(4 downto 0);
		exitShamt: out std_logic_vector(31 downto 0)
	);
end concatShamt;

architecture Behavioral of concatShamt is

begin

	process (inShamt)
	begin
			exitShamt <= "00000000000000000000000000000" & inShamt;
	end process;

end architecture;
```

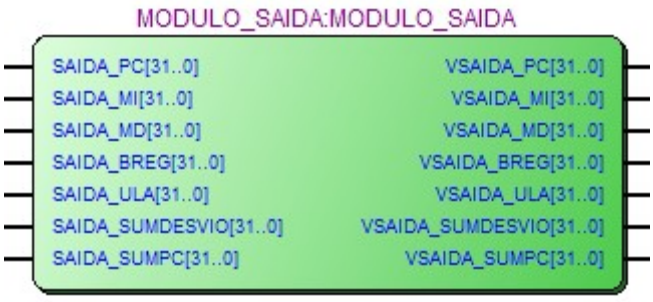
- Calculador de endereço(LW/SW)



O calculo do endereço a ser lido ou escrito pela instrução de acesso a memoria devem ser feito nesse seguinte módulo. Ele recebe o calculo do endereço feito na ULA e o aloca como enderço válido a ser usado pelo indice do dado gerado no arquivo de dado mif do MARS. Assim temos :

```
ENTITY address_calc IS
	port (
		aluResult		: IN std_logic_vector(31 DOWNTO 0);
		addressToMem		: OUT std_logic_vector(7 DOWNTO 0)
	);
END address_calc;
ARCHITECTURE Behavioral OF address_calc IS
	SIGNAL shift2		: std_logic_vector(31 DOWNTO 0);
	SIGNAL calcMem		: unsigned(31 downto 0);
BEGIN
	calcMem <= unsigned(aluResult) - X"2000";
	shift2 <= std_logic_vector(shift_right (calcMem,2));
	addressToMem <= std_logic_vector(shift2(7 DOWNTO 0));
END ARCHITECTURE Behavioral;
```

- Modulo de saida:



Esse modulo de saida, foi criado para poder verificar cada saáda de cada módulo principal, de forma a verificar o correto funcionamento de cada um deles e deles como um todo, assim mostra o codigo VHDL abaixo:

```
entity MODULO_SAIDA is

	Port (
		SAIDA_PC	: in  STD_LOGIC_VECTOR (31 DOWNTO 0);
		SAIDA_MI	: in  STD_LOGIC_VECTOR (31 DOWNTO 0);
		SAIDA_MD	: in  STD_LOGIC_VECTOR (31 DOWNTO 0);
		SAIDA_BREG	: in  STD_LOGIC_VECTOR (31 DOWNTO 0);
		SAIDA_ULA	: in  STD_LOGIC_VECTOR (31 DOWNTO 0);
		SAIDA_SUMDESVIO: in  STD_LOGIC_VECTOR (31 DOWNTO 0);
		SAIDA_SUMPC	: in  STD_LOGIC_VECTOR (31 DOWNTO 0);

		-- V = VALOR DE SAIDA
		VSAIDA_PC	: OUT  STD_LOGIC_VECTOR (31 DOWNTO 0);
		VSAIDA_MI	: OUT  STD_LOGIC_VECTOR (31 DOWNTO 0);
		VSAIDA_MD	: OUT  STD_LOGIC_VECTOR (31 DOWNTO 0);
		VSAIDA_BREG	: OUT  STD_LOGIC_VECTOR (31 DOWNTO 0);
		VSAIDA_ULA	: OUT  STD_LOGIC_VECTOR (31 DOWNTO 0);
		VSAIDA_SUMDESVIO: OUT  STD_LOGIC_VECTOR (31 DOWNTO 0);
		VSAIDA_SUMPC	: OUT  STD_LOGIC_VECTOR (31 DOWNTO 0)
	);

end MODULO_SAIDA;

architecture Behavioral of MODULO_SAIDA is

begin

	VSAIDA_PC		<= SAIDA_PC;
	VSAIDA_MI		<= SAIDA_MI;
	VSAIDA_MD		<= SAIDA_MD;
	VSAIDA_BREG		<= SAIDA_BREG;
	VSAIDA_ULA		<= SAIDA_ULA;
	VSAIDA_SUMDESVIO	<= SAIDA_SUMDESVIO;
	VSAIDA_SUMPC		<= SAIDA_SUMPC;

end Behavioral;
```

- Modulo de saida secundario:

Esse modulo foi criado para apresentar as saidas dos multiplexadores principais:

```
entity MODULO_SAIDA_SECUNDARIO is

	Port (
		SAIDA_MUXRD	: in  STD_LOGIC_VECTOR (4 DOWNTO 0);
		SAIDA_MUXALU	: in  STD_LOGIC_VECTOR (31 DOWNTO 0);
		SAIDA_MUXAB	: in  STD_LOGIC_VECTOR (31 DOWNTO 0);
		SAIDA_MUXB	: in  STD_LOGIC_VECTOR (31 DOWNTO 0);

		-- V = VALOR DE SAIDA
		VSAIDA_MUXRD	: OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
		VSAIDA_MUXALU	: OUT  STD_LOGIC_VECTOR (31 DOWNTO 0);
		VSAIDA_MUXAB	: OUT  STD_LOGIC_VECTOR (31 DOWNTO 0);
		VSAIDA_MUXB	: OUT  STD_LOGIC_VECTOR (31 DOWNTO 0)
	);
```

```
end MODULO_SAIDA_SECUNDARIO;

architecture Behavioral of MODULO_SAIDA_SECUNDARIO is

begin

	VSAIDA_MUXRD		<= SAIDA_MUXRD;
	VSAIDA_MUXALU		<= SAIDA_MUXALU;
	VSAIDA_MUXAB		<= SAIDA_MUXAB;
	VSAIDA_MUXB		<= SAIDA_MUXB;

end Behavioral;
```

- Modulo de saida das instrução

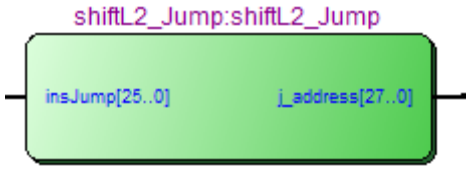
Como o nome o indica ele foi criado para apresentar as saidas dos diferentes campos , resutados da busca de instrução na memoria de instrução, para verificar o tipo de operação assim como os indices dos registradores usados. Isso foi realizado como a seguir:

```
entity MODULO_SAIDA_INSTRUCTION is

	Port (
		SAIDA_INS_RS		: in STD_LOGIC_VECTOR (4 DOWNTO 0);
		SAIDA_INS_RT		: in STD_LOGIC_VECTOR (4 DOWNTO 0);
		SAIDA_INS_RD		: in STD_LOGIC_VECTOR (4 DOWNTO 0);
		SAIDA_INS_SHAMT		: in STD_LOGIC_VECTOR (4 DOWNTO 0);
		SAIDA_INS_FUNCT		: in STD_LOGIC_VECTOR (5 DOWNTO 0);
		SAIDA_INS_OPCODE	: in STD_LOGIC_VECTOR (5 DOWNTO 0);
		SAIDA_INS_IMM16		: in STD_LOGIC_VECTOR (15 DOWNTO 0);
		SAIDA_INS_IMM26		: in STD_LOGIC_VECTOR (25 DOWNTO 0);

		-- V = VALOR DE SAIDA
		VSAIDA_INS_RS		: out STD_LOGIC_VECTOR (4 DOWNTO 0);
		VSAIDA_INS_RT		: out STD_LOGIC_VECTOR (4 DOWNTO 0);
		VSAIDA_INS_RD		: out STD_LOGIC_VECTOR (4 DOWNTO 0);
		VSAIDA_INS_SHAMT	: out STD_LOGIC_VECTOR (4 DOWNTO 0);
		VSAIDA_INS_FUNCT	: out STD_LOGIC_VECTOR (5 DOWNTO 0);
		VSAIDA_INS_OPCODE	: out STD_LOGIC_VECTOR (5 DOWNTO 0);
		VSAIDA_INS_IMM16	: out STD_LOGIC_VECTOR (15 DOWNTO 0);
		VSAIDA_INS_IMM26	: out STD_LOGIC_VECTOR (25 DOWNTO 0)
	);

end MODULO_SAIDA_INSTRUCTION;

architecture Behavioral of MODULO_SAIDA_INSTRUCTION is

begin
```

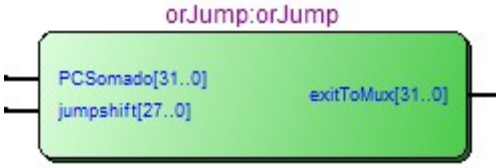
```
        VSAIDA_INS_RS               <= SAIDA_INS_RS;
        VSAIDA_INS_RT               <= SAIDA_INS_RT;
        VSAIDA_INS_RD               <= SAIDA_INS_RD;
        VSAIDA_INS_SHAMT            <= SAIDA_INS_SHAMT;
        VSAIDA_INS_FUNCT            <= SAIDA_INS_FUNCT;
        VSAIDA_INS_OPCODE           <= SAIDA_INS_OPCODE;
        VSAIDA_INS_IMM16            <= SAIDA_INS_IMM16;
        VSAIDA_INS_IMM26            <= SAIDA_INS_IMM26;

end Behavioral;
```

III-3-10/Instruções adicionais JUMP/ADDI/Branch if not equal /Shift left logical/ Shift right logical/SRA

- IMPLEMENTAÇÃO JUMP

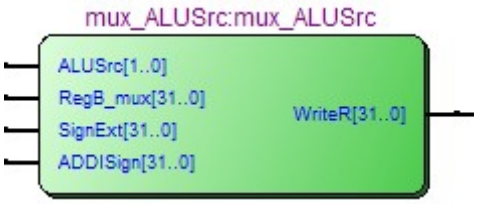


Foi usado para realizar o cálculo do endereço de desvio condicional, um modulo *shiftL2_Jump* para fazer o deslocamento de 2 bits dos bits[25-0] da instrução vindo daJUMP memoria de instrução:



E para concatenação desses bits agora [27-0] com os 4 bits mais significativos de PC+4 usamos o módulo que nomeamos de “orJump” a seguir:
Como explicado anteriormente, esse resultado de desvio será ativado pelo multiplexador”muxJump” responsável de enviar PC+4 ou o endereço de desvio.

- IMPLEMENTAÇÃO ADDI



Para implementação Addi, tivemos que adicionar mais uma entrada ao multiplexador “mux_Alusrc”, tornando o de 3 entradas, responsável para escolher o conteudo dos 32 bits de imediato. Como a seguir:

Esses 32 bits foram gerados a partir de um módulo ”addiComplete” que concatena os 16 bits do imediato com 16 bits a esquerda zerado. Assim tivemos:

- IMPLEMENTAÇÃO BNE

  As diferentes portas AND e OR implementados para o bne foram exibidos já nos item anteriores, assim tivemos que para controlar o muxbranch que é destinado para operação do tipo branch criar um novo sinal de controle vindo do controle pricipal”branchNE” para realizar lo.

- IMPLEMENTAÇÃO SLL, SRL e SRA

  Para a implementação das instruções de SLL, SRL e SRA tivemos que criar um multiplexador de duas entradas, para selecionar o valor de entrada do regA (primeira saída do breg) e o valor do shamt, sendo controlado pelo valor de controle ShamtSg.

Lembrando que o código VHDL que será enviado junto com o relatório descreve melhor o funcionamento da implementação uniciclo desejada.

IV/-<u>RESULTADOS E SIMULAÇÃO</u>:

Como resultado, tivemos o que se era esperado através da execução do programa Uniciclo.vhd, utilizando o arquivo text.mif, que contém o código em hexadecimal gerado pelo MARS, e o arquivo data.mif, que contém os valores em hexadecimal dos dados de memória, também gerado pelo MARS.

Com isso, obtivemos:

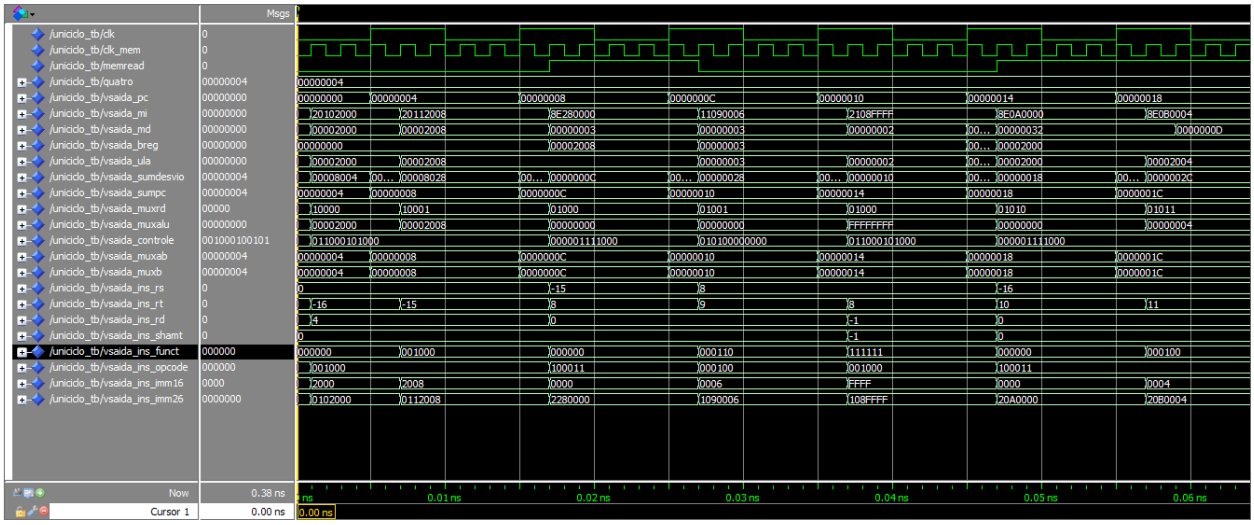


Figura 11: Execução de 0 a 60 ps.

Em que foram feitas as operações de ADDI, opcode = 001000, ADDI novamente, LW, opcode = 100011, carregando 3 da memória, BEQ, opcode = 000100, comparando 3 com 0 para o pulo, ADDI subtraindo 3 com 2, LW, carregando 50 da memória e LW, carregando 13 da memória.

Após essas instruções, tivemos:

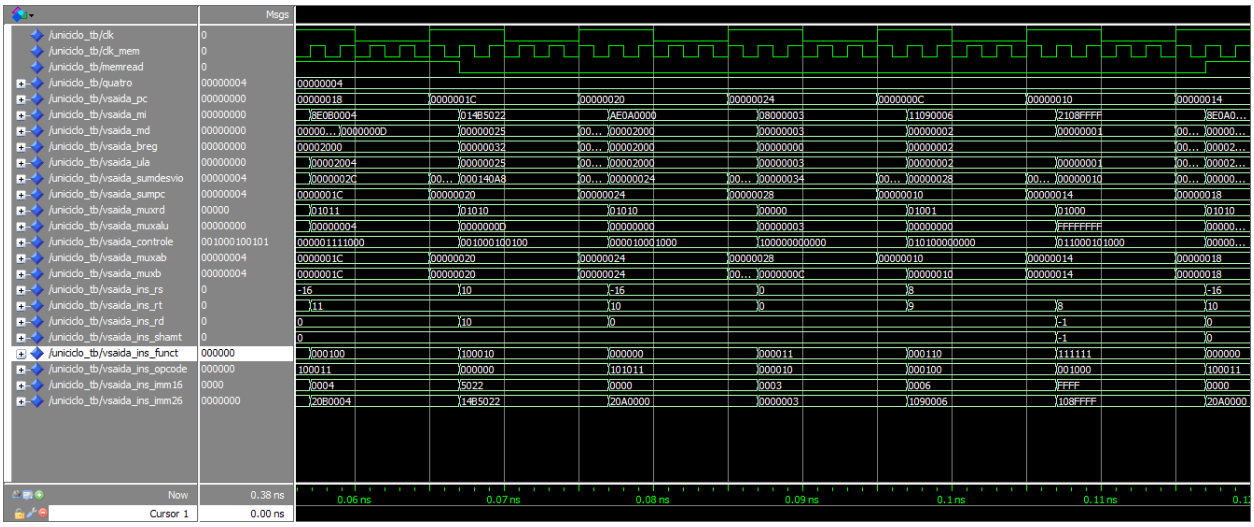


Figura 12: Execução de 60 a 110 ps.

Realizando as instruções:

SUB, com o opcode = 0 e funct = 100010, realizando a subtração de 0x32 – 0x0D = 0x25, SW, com o opcode = 101011, colocando 0x25 no endereço 0 (2000) da memória. Após, foi feito um jump para o beq desse mesmo label.

Nas 3 imagens seguintes foram realizadas operações de loop:

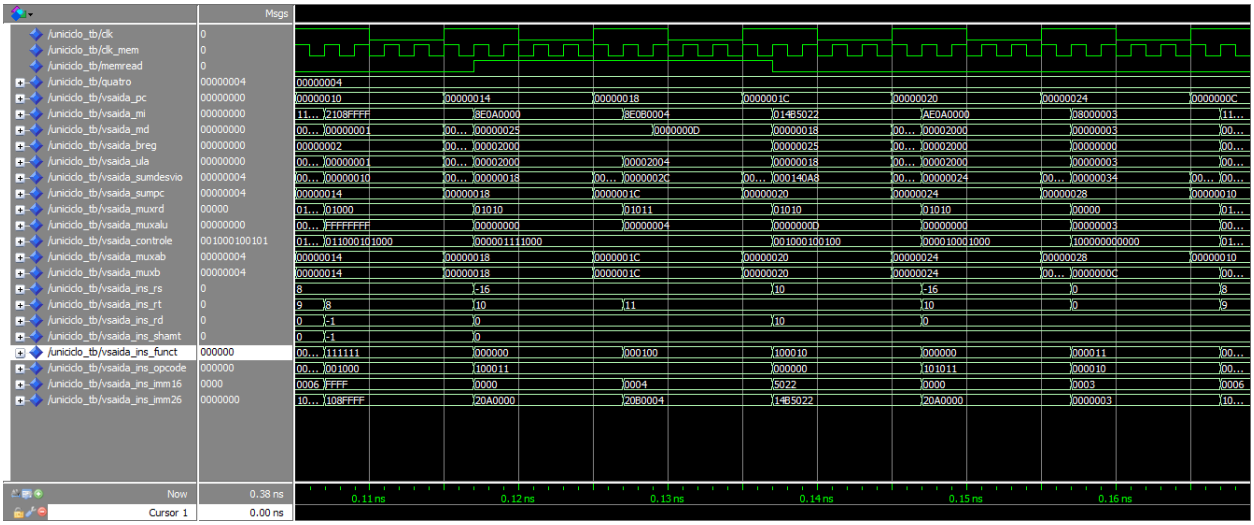


Figura 13: Execução de 110 a 160 ps.

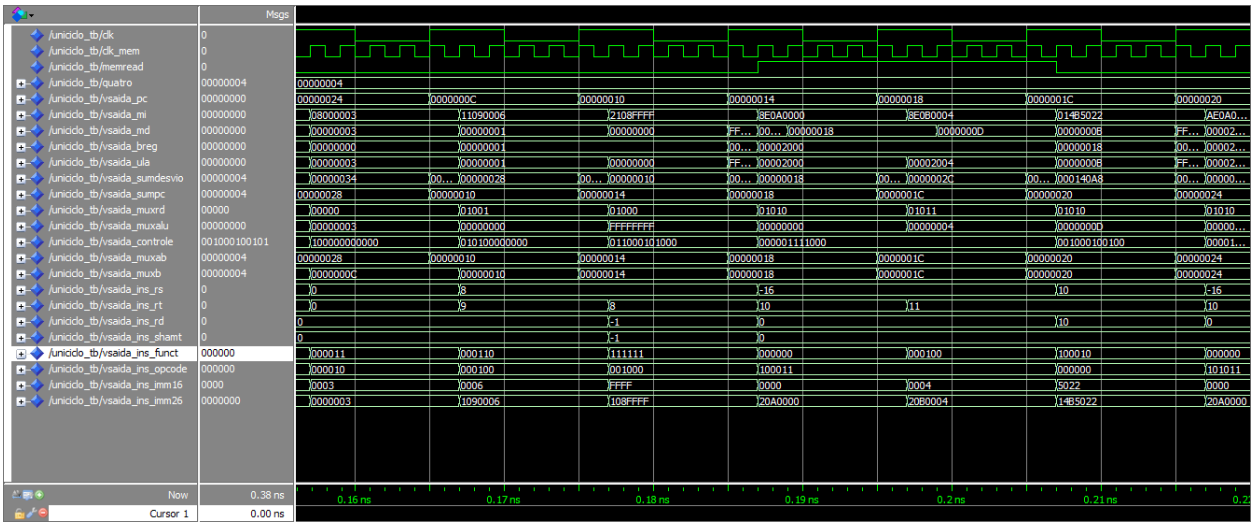


Figura 14: Execução de 160 a 210 ps.

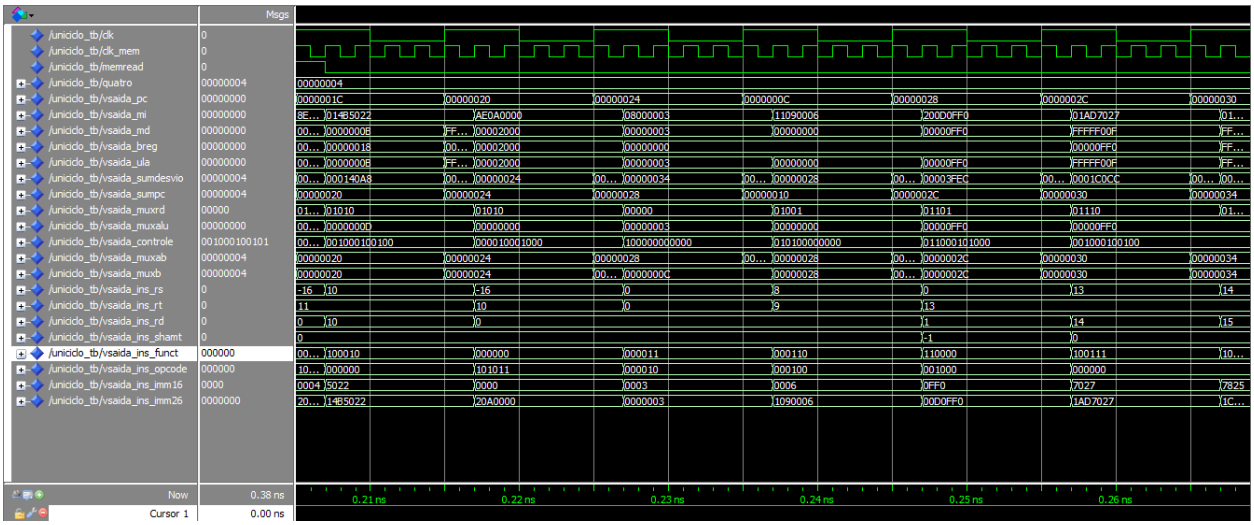


Figura 15: Execução de 210 a 260 ps.

Nesta última imagem é encerrado o loop e segue-se o código fazendo a operação ADDI e depois NOR, opcode = 000000 e funct = 100111, resultando em FFFFF00F.

Na próxima imagem, foi-se realizada a operação OR, opcode = 000000 e funct = 100101, resultado em FFFFFFFF.

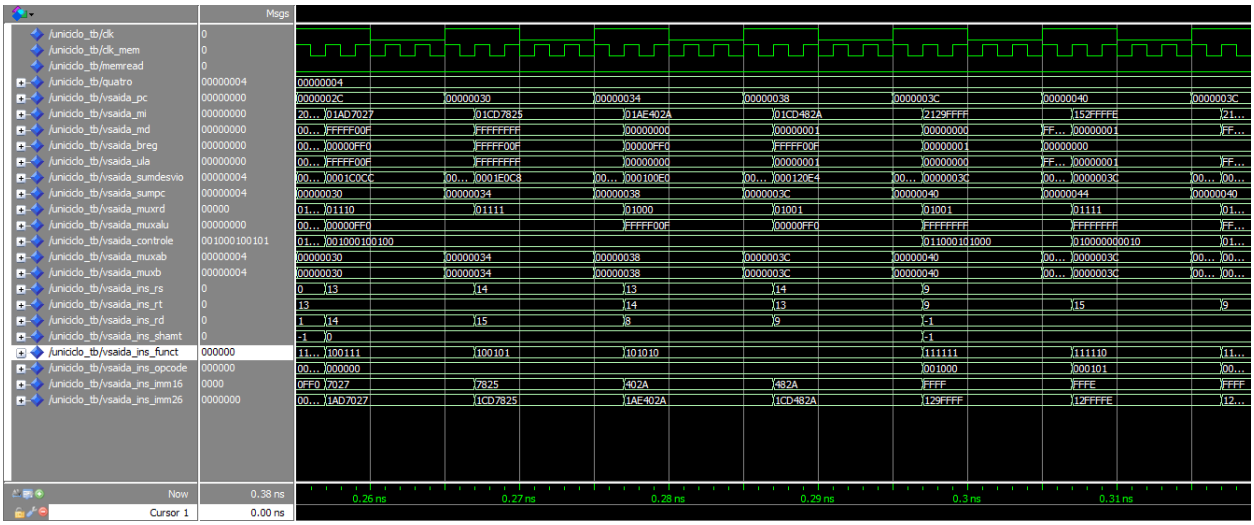


Figura 16: Execução de 260 a 310 ps.

Após a operação OR, foram realizadas duas operações slt, opcode = 000000 e funct = 101010, a primeira resultando em 00000000 e a segunda em 00000001. Após foi realizada mais uma operação de ADDI, seguindo de um BEQ, entrando em um pequeno loop, como mostrado na imagem abaixo:

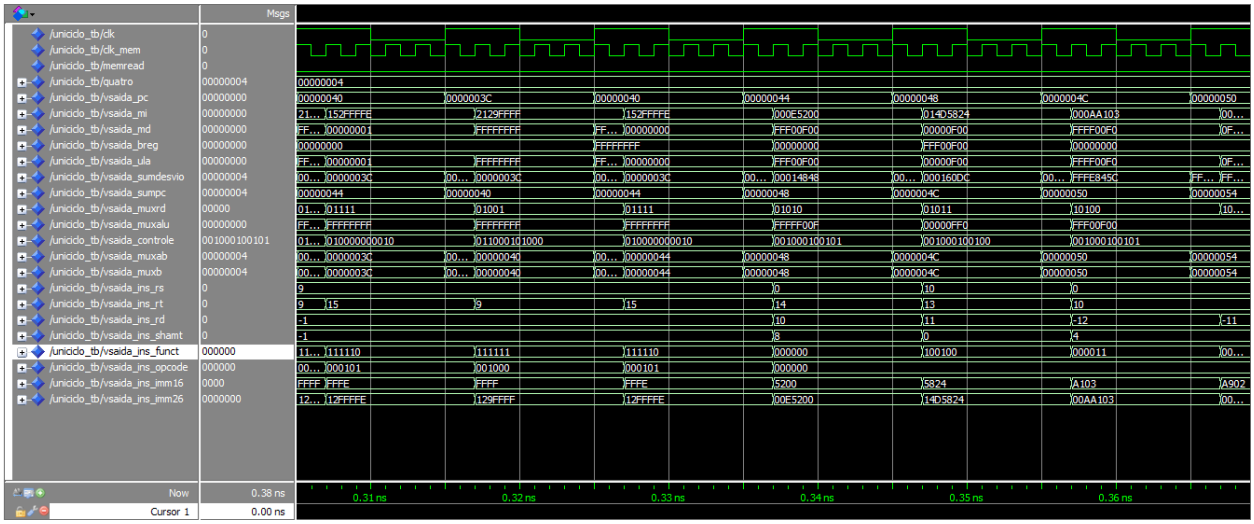


Figura 17: Execução de 310 a 360 ps.

Entra-se em um pequeno loop de uma chamada e realiza a operação SLL, opcode = 000000 e funct = 000000, resultando em FFF00F00. Após é realizado um AND, opcode = 000000 e funct = 100100, resultando em 00000F00. Após o AND, é feito um SRA, opcode = 000000 e funct = 000011, resultando em FFFF00F0.

Em seguida, quase terminando a execução do programa, temos a última instrução:

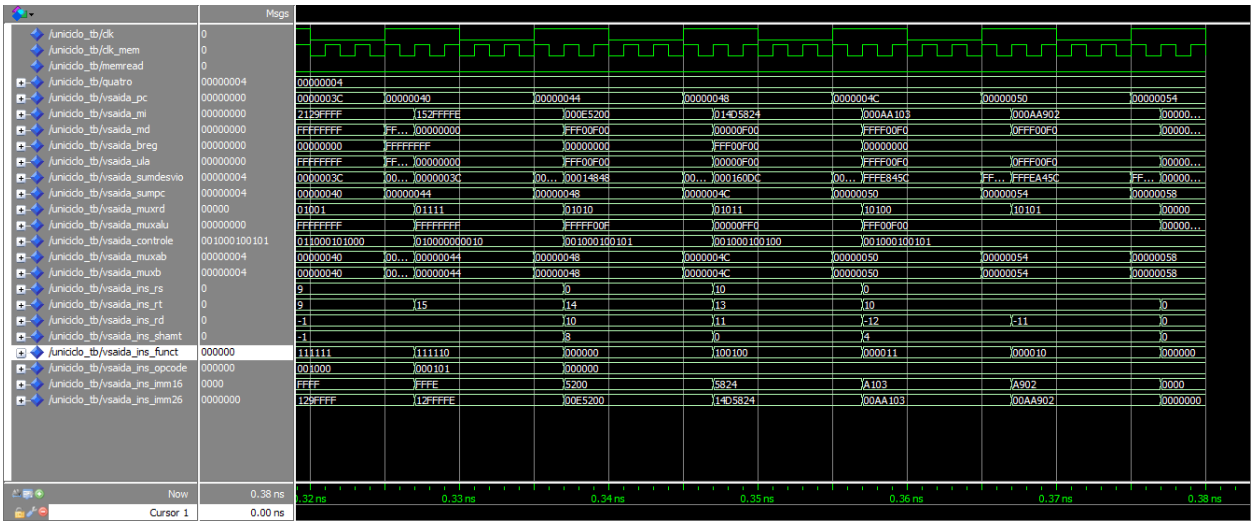
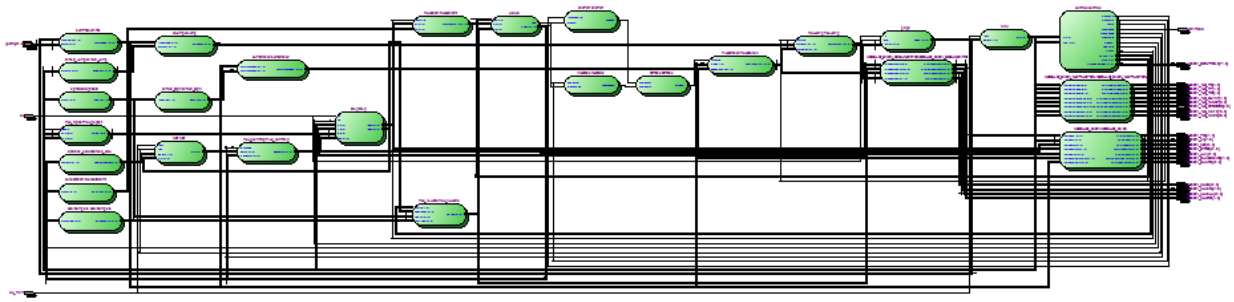


Figura 18: Execução de 360 a 380 ps.

Um SRL, resultando em 0FFF00F0, como confirmado através dos comentários do código do professor, *testeMIPS.asm*, e da execução deste mesmo arquivo no MARS, passo a passo.

Como resultado da simulação RTL, do processador Uniciclo, tivemos o seguinte diagrama:



IV/-<u>CONCLUSÃO</u>:

Como estipulado com os objetivos do trabalho, foi-se possível aprimorar nossos conhecimentos quanto ao assembly MIPS, quanto ao funcionamento do Uniciclo, quanto ao funcionamento do programa Quartus e o aprendizado da linguagem VHDL. Como resultados, obtivemos os esperados para o término no trabalho, realizando corretamente a simulação, a execução e projeção do processador pedido. Além disso, foi possível também, relembrar alguns conceitos e fixar melhor os funcionamentos de cada módulo implementado.