Projeto 2 – Projeto de Processador Multiciclo

Arquitetura de Computadores

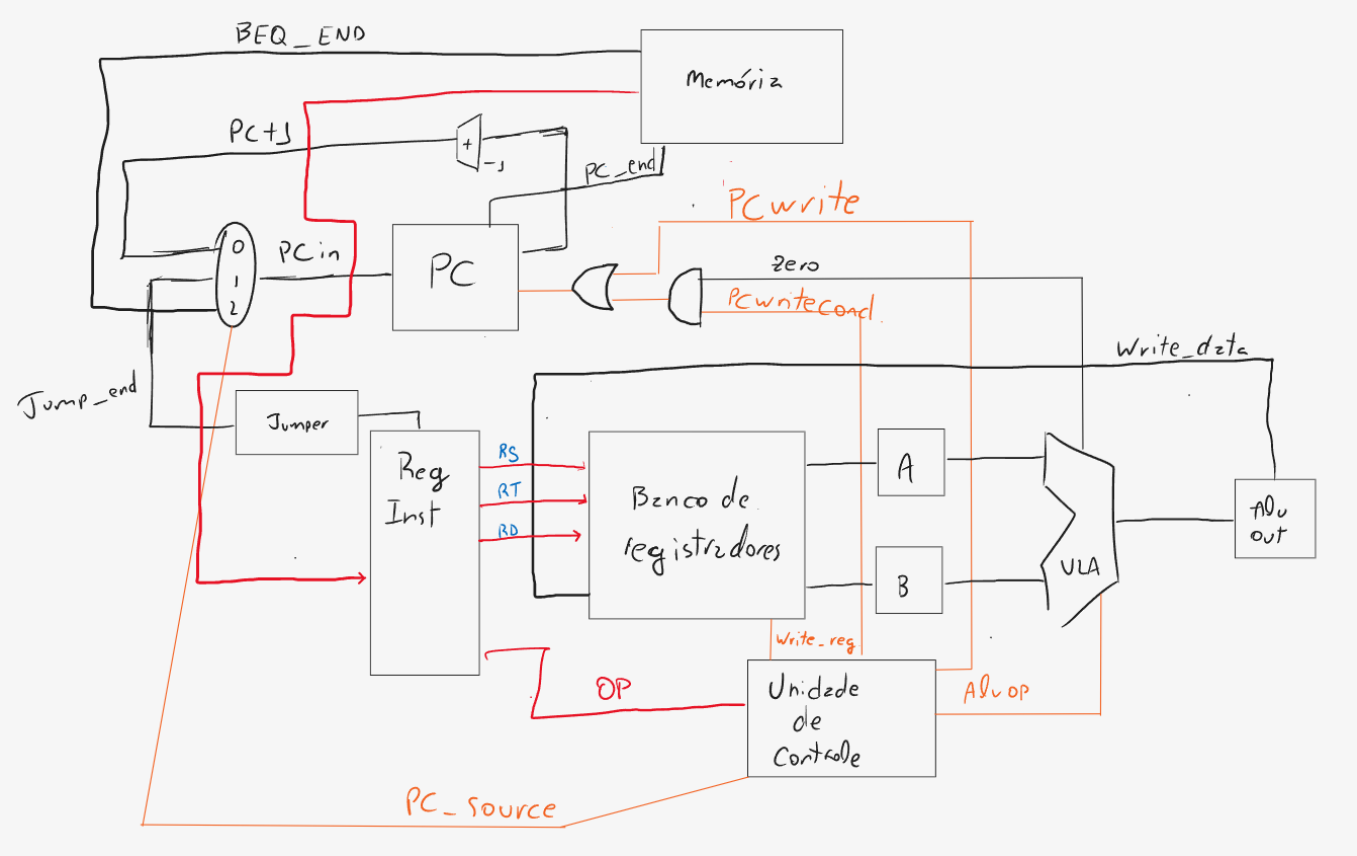
Igor Shinji Itiroko e Rodrigo da Silva Cardoso

Engenharia de Computação – PUC Campinas

Campinas, SP - Brasil

1. Descrição textual

Topologia da CPU (Microsoft Whiteboard)



O projeto foi desenvolvido e testado inteiramente no Quartus 20.1, ele consiste em uma CPU desenvolvida para realizar 4(quatro) operações diferentes. Soma de registradores, subtração de registradores, função Jump condicional (Branch Equal) e Jump incondicional (Jump).

Memória

```
ARCHITECTURE Behavior OF Memoria IS
   TYPE vetor_instrucoes IS ARRAY (0 TO 6) of STD_LOGIC_VECTOR (7 DOWNTO 0);


   SIGNAL int_address: INTEGER RANGE 0 TO 6;

   CONSTANT instrucao : vetor_instrucoes:= ("00011011","01001001","10011100","00000101","11000001","00011011","00000000");

   BEGIN
      int_address <= to_integer(signed(PC_endereco));
      instrucao_out <= instrucao(int_address);
      BEQout <= instrucao(int_address + 1);

END Behavior;
```

O componente ‘Memória’ foi desenvolvido para ser um array de instruções controlado pelos endereços fornecidos pelo PC. Tem como entrada o endereço, e como saídas instrução_out e BEQout. Instrução_out devolve a instrução localizada no endereço fornecido por PC.

BEQout é o endereço seguinte ao endereço lido do PC, para o projeto utilizamos instruções BEQ separadas em duas instruções diferentes, onde a primeira identifica quais registradores serão comparados e a segunda qual o endereço de jump caso a condição seja satisfeita.

Banco de Registradores

```
BEGIN
   --- Instanciamento de registradores.
   R0: registrador PORT MAP (D0, reset, RegWrite, clock, Q0);
   R1: registrador PORT MAP (D1, reset, RegWrite, clock, Q1);
   R2: registrador PORT MAP (D2, reset, RegWrite, clock, Q2);
   R3: registrador PORT MAP (D3, reset, RegWrite, clock, Q3);


   --- Escrita dos registradores.
   -- SetTest só é usado no começo uma vez, só que WriteReg inicia com "11" quando nao é instanciado, por isso que D3 precisa de um AND a mais.
   D0 <= WriteData WHEN RegWrite = '1' AND WriteReg = "00" ELSE
         "00000001" WHEN SetTest = '1' ELSE
          Q0;
   D1 <= WriteData WHEN RegWrite = '1' AND WriteReg = "01" ELSE
         "00000010" WHEN SetTest = '1' ELSE
         Q1;
   D2 <= WriteData WHEN RegWrite = '1' AND WriteReg = "10" ELSE
         "00000011" WHEN SetTest = '1' ELSE
         Q2;
   D3 <= WriteData WHEN RegWrite = '1' AND WriteReg = "11" AND SetTest = '0' ELSE
         "00000100" WHEN SetTest = '1' ELSE
         Q3;

   --- Leitura dos registradores.
   ReadData1 <=   Q0 when ReadReg1 = "00" else
                  Q1 when ReadReg1 = "01" else
                  Q2 when ReadReg1 = "10" else
                  Q3 when ReadReg1 = "11" else
                  ( OTHERS => '0' ) ;

   ReadData2   <= Q0 when ReadReg2 = "00" else
                  Q1 when ReadReg2 = "01" else
                  Q2 when ReadReg2 = "10" else
                  Q3 when ReadReg2 = "11" else
                  ( OTHERS => '0' ) ;

END Structure ;
```

O banco de registradores funciona a partir de duas operações, escrita e leitura, em seu processo de escrita ele coleta um dado de 8 bits como entrada e escreve internamente em um de seus 4 registradores internos. Para a leitura, ele recebe o endereço de dois registradores (RS e RT) e retorna os mesmo registradores em suas saídas RegRead1 e RegRead2.

PC (Program Counter)

```
ARCHITECTURE Behavior OF PC IS
SIGNAL intermediario: STD_LOGIC_VECTOR(N-1 DOWNTO 0) := "00000000";

BEGIN

   PROCESS (clock)
   BEGIN
      IF Clock'EVENT AND Clock = '1' THEN
         IF reset = '1' THEN
            PCout <= ( OTHERS => '0' );
         ELSIF PCload = '1' AND PCSource = "01" THEN
            intermediario <= PCin;
            PCout <= intermediario;
         ELSIF PCload = '1' AND PCSource = "00" THEN
            intermediario <= intermediario + "00000001";
            PCOut <= intermediario;
         ELSIF PCload = '1' AND PCSource = "10" THEN
            intermediario <= BEQin;
            PCout <= intermediario;
         ELSIF PCload = '0' AND PCsource = "10" THEN
            intermediario <= intermediario + "00000001";
            PCOut <= intermediario;
         END IF;
      END IF;
   END PROCESS ;
   PCmsb <= intermediario(7 downto 6);
END Behavior ;
```

PC consiste em um contador de instruções que representa qual o endereço da próxima instrução a ser lida pela memória, por ser controlado por PC <= PC + 1 no final de cada instrução ou por endereços de Jump ou BEQ, a decisão é feita pelo ‘PCSource’, um sinal de controle emitido pela Unidade de Controle.

ULA (Unidade Lógica e Aritmética)

```
BEGIN

   Bsig <= std_logic_vector( unsigned(NOT B) + 1) WHEN ALUop = '1' ELSE
           B WHEN Aluop = '0' ELSE
           (OTHERS => '0');

   Zero <= (NOT (Sgen(0) OR Sgen(1) OR Sgen(2) OR Sgen(3) OR Sgen(4) OR Sgen(5) OR Sgen(6) OR Sgen(7)));
   Result <= Sgen;
   SOMADOR1: ripple_carry PORT MAP(A,Bsig,Sgen);
END ARCHITECTURE;
```

A ULA é o componente da CPU responsável pelas operações lógicas e aritméticas envolvendo registradores. Ela instancia um somador ripple_carry para auxiliar na soma. E retorna, além de seu resultado um sinal ‘zero’ responsável pelo controle da instrução BEQ.

UC (Unidade de Controle)

```
Architecture Behavior OF UC IS

SIGNAL state: integer := 0;

BEGIN
   PROCESS (clock)
   BEGIN
      IF clock'EVENT AND clock = '0' THEN

         CASE state IS
            -- Reset nos registradores.
            WHEN 0 => UCSign <= "000000";
                      Reset <= '1';
                      SetTest <= '0';
                      state <= 1;

            -- Set dos valores iniciais do registradores.
            WHEN 1 => UCSign <= "000001";
                      Reset <= '0';
                      SetTest <= '1';
                      state <= 2;

            -- Fetch
            WHEN 2 => UCSign <= "010000";
                      SetTest <= '0';
                      state <= 3;
            -- Decode
            WHEN 3 => UCSign <= "000000";
                      state <= 4;
            -- Execute
            WHEN 4 =>
               IF OPin = "00" THEN
                  UCSign <= "000000";
                  state <= 5;
                  ELSIF OPin = "01" THEN
                     UCSign <= "000010";
                     state <= 5;
                  ELSIF OPin = "10" THEN
                     UCSign <= "101010";
                     state <= 2;
                  ELSE
                     UCSign <= "010100";
                     state <= 2;
               END IF;

            -- Write Back
            WHEN 5 => UCSign <= "000001";
                  state <= 2;

            WHEN OTHERS => state <= 2;
         END CASE;
      END IF;
   END PROCESS;
```

A unidade de controle é responsável por definir os ciclos de cada instrução. Ela recebe como parâmetro o OPcode da instrução e a traduz em um vetor, que é traduzido na CPU nos sinais de controle necessários para a execução da instrução.

Componentes básicos

Além dos componentes citados acima foram usados uma série de componentes menores que normalmente são instanciados como ‘COMPONENT’ no código VHDL.

- **Registradores**

Os registradores são a unidade básica do banco de registradores eles possuem uma entrada Q e uma saída D e respondem por Clock e Load. Não realizam nenhum tipo de alteração no dado, só servem para a sincronização da CPU para torna-la multiciclo.

- **Registrador de instruções**

O registrador de instruções é o componente que recebe as instruções da memória através de um barramento de 8 bits, e traduz essa instrução para ser lida no banco de registradores, na Unidade de Controle ou no Jumper. Para instruções do tipo R e BEQ ele faz a divisão de 2 bits para OP, 2 bits para RS, 2 bits para RT e 2 bits para RD. Para jumps, 2 bits para OP e 6 para endereçamento, e para BEQ 2 bits de OP, 2 para RS e 2 para RT.

- **Jumper**

O Jumper concatena os 6 bits recebidos pelo registrador de instruções e com os 2 bits MSB recebidos pelo PC.

- **CPU**

Utilizando toda a base dos outros componentes descritos acima, a CPU instancia todos eles e através de sinais faz seu funcionamento em conjunto possível.

2. Especificação

2.1 - Registradores

No processador são usados 7 registradores, o Reg A e Reg B usados para operações na ULA, o Aluout (saída da ULA) e 4 registradores internos do banco de registradores, R0, R1, R2 e R3, os quais são endereçados como 00, 01, 10 e 11 respectivamente. Todos possuem 8 bits de tamanho.

2.2 - Formato das instruções

Instrução	OPCODE
Tipo R (soma)	00
Tipo R (sub)	01
BEQ	10
Jump	11

2.3 - Unidade de Controle

Formato das instruções e sinais:

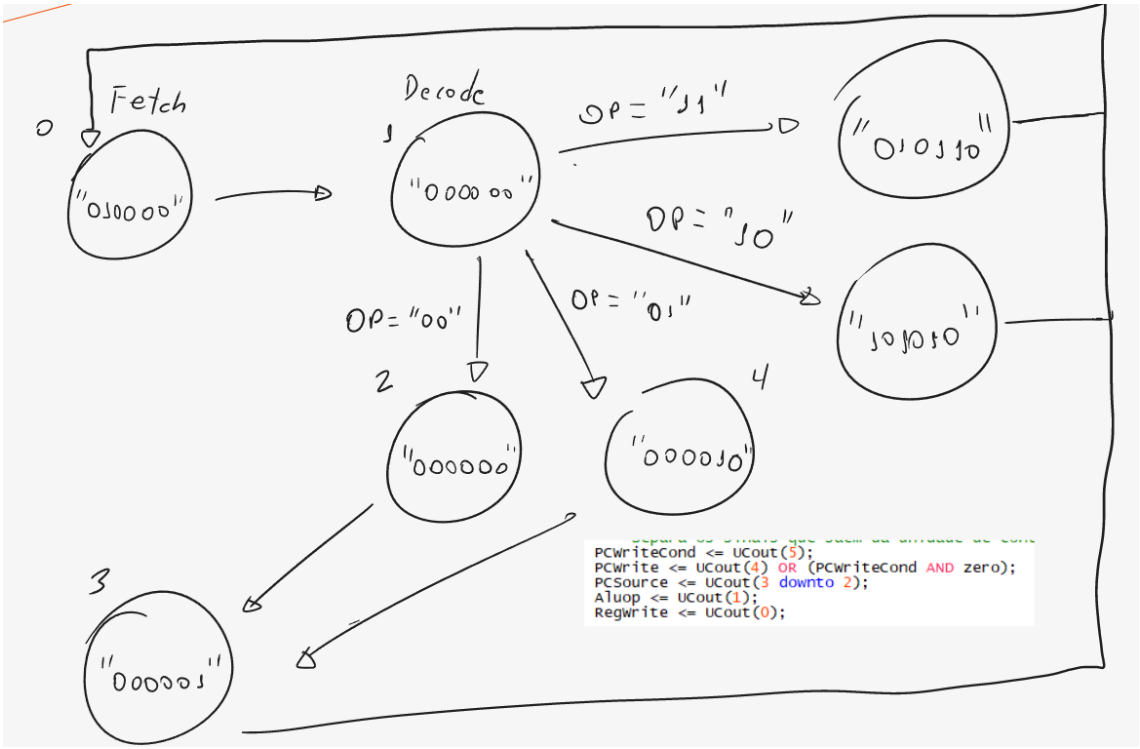
Para operações '+' Aluop = 0, para operações '-' Aluop = 1 no 3o ciclo

Tipo R	1o Ciclo	2o Ciclo	3o Ciclo	4o Ciclo
PCWriteCond	0	0	0	0
PCWrite	1	0	0	0
PCSource	00	00	x	x
Aluop	0	0	0 ou 1	x
RegWrite	0	0	0	1

BEQ	1o Ciclo	2o Ciclo	3o Ciclo
PCWriteCond	0	0	1
PCWrite	1	0	0
PCSource	00	00	10
Aluop	0	0	1
RegWrite	0	0	0

Jump	1o Ciclo	2o Ciclo	3o Ciclo
PCWriteCond	0	0	0
PCWrite	1	0	1
PCSource	00	00	01
Aluop	0	0	0
RegWrite	0	0	0

Diagrama de estados



Para facilitar o entendimento das ondas de simulação realizamos os sinais de controle como um vetor de 6 bits, o qual é dividido na CPU.

PCWriteCond é o sinal de Branch. Recebe 1 apenas quando identifica uma operação de branch.

O **PCWrite** serve para a escrita do PC. Recebe 1 quando está sendo realizado o Fetch da instrução ou quando é realizado algum Jump.

O **PCSource** controla as entradas do PCin, se vai incrementar ou pular para algum endereço. Recebe 00 para quando é uma instrução do tipo R, faz PC+1. Recebe 01 quando é uma instrução de jump, e 10 quando é uma instrução de branch.

O **Aluop** é responsável pelo controle das operações da ULA. Pode ser 0 para uma soma, ou 1 para um subtração.

RegWrite controla a escrita dos registradores. Normalmente é 0, então os regitradores não são alterados, e 1 durante o WriteBack para alterar o valor do registrador destino.

3. Resultados

3.1 - Descrição do teste realizado

Para esse teste, o vetor de intruções foi colocado como visto abaixo:

```
CONSTANT instrucao : vetor_instrucoes:= (
"00011011", "01100110", "10001100", "00000000",
"10001000", "00001000", "00000001", "00000010",
"11000000", "00000011", "00000000", "00000000", "00000000");
```

Nesse caso, espera-se que a CPU faça primeiro uma soma. Opcode = 00, RS = 01, RT = 10 e RD = 11, que seria R1+R2 => R3. Depois, uma subtração R2 - R1 => R2.

Depois disso vai vir dois branches. O primeiro é “Not Taken”, ou seja, espera-se que o conteudo em R0 seja diferente do conteudo em R3. O segundo é um branch “Taken”, pois a subtração fará com que o valor em R0 seja igual ao valor em R2.

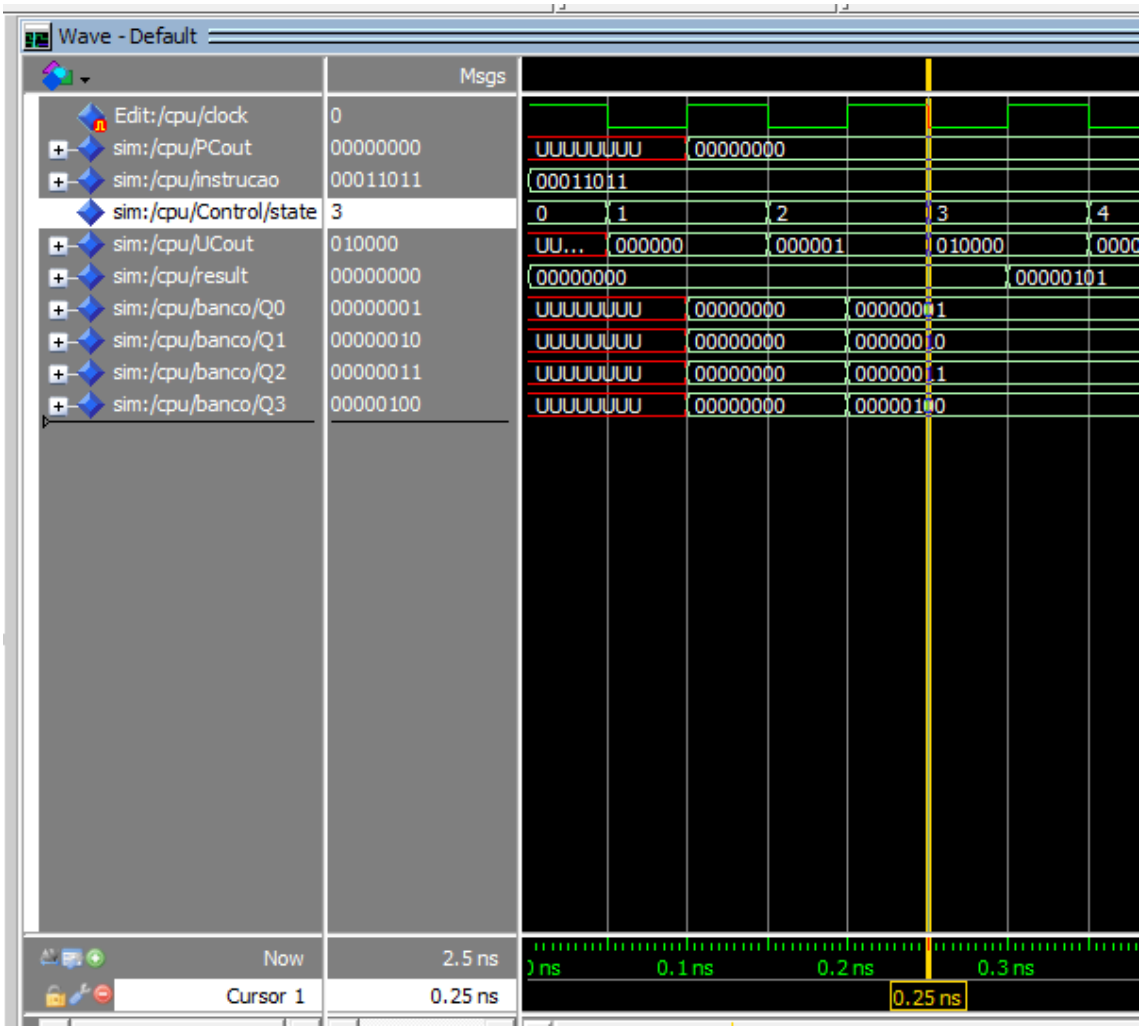
O segundo branch vai pular para a instrução na posição 8 do vetor (como visto em azul, 00001000 = 8), que tem uma instrução de jump (*11000000*), que vai fazer a CPU pular pra posição 0, rodando a primeira instrução de soma novamente.

As instruções em cinza teoricamente não serão acessadas e servem para identificar possíveis erros na CPU.

3.2 - Resultados e discussão

Observação importante: O código foi escrito de maneira que, quando a Unidade de Controle está no estado 0, ela atualiza o “state” para o próximo estado, que seria 1. Quando está no estado 1, atualiza o “state” para 2, e assim sucessivamente. Portanto, nos ciclos de clock que o “state” está com 1, na verdade o processador está no estado 0.

Estado 0 e Estado 1

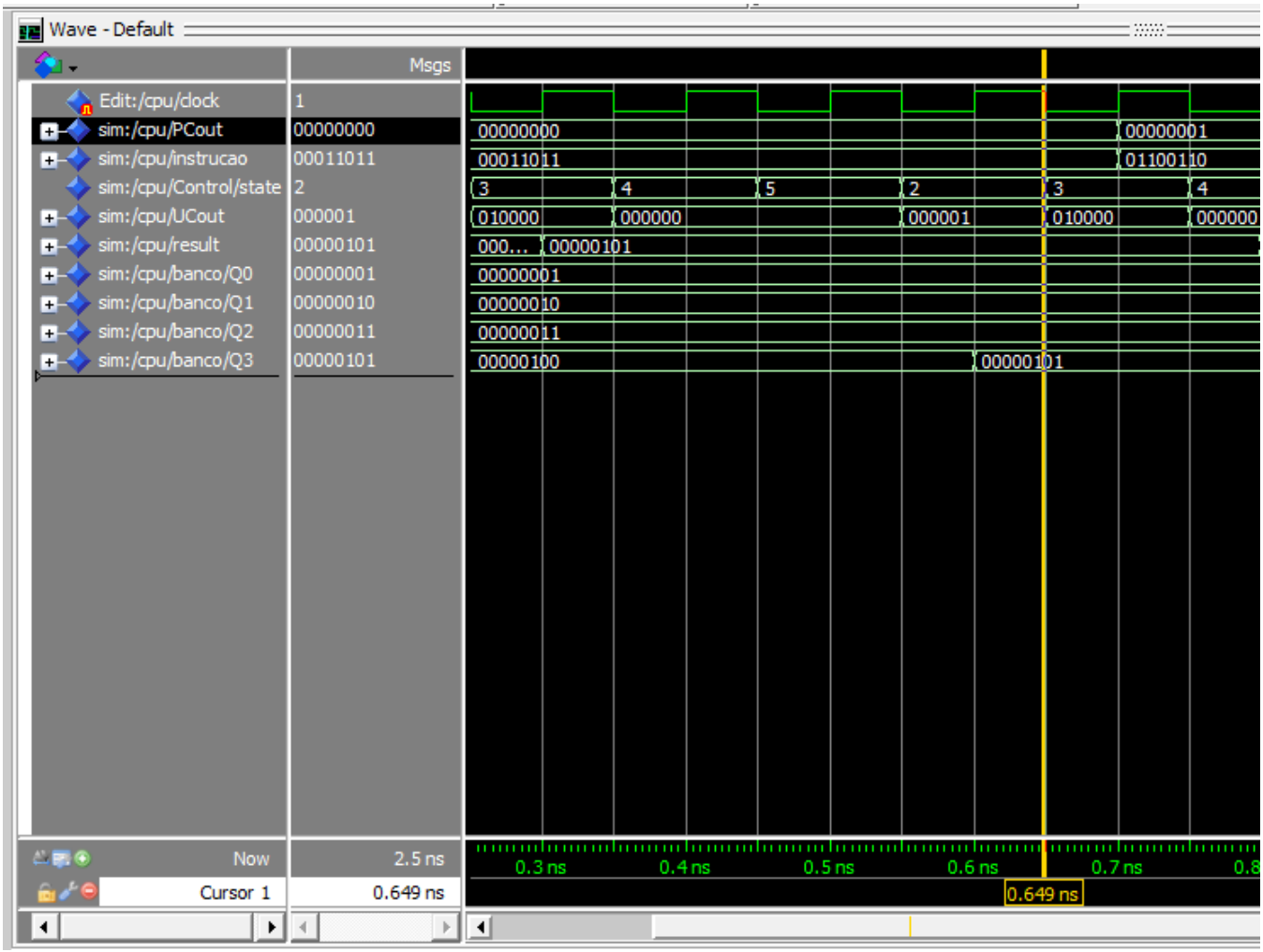


Os primeiros dois estados da unidade de controle da CPU foram feitos para inicializar os registradores.

No **estado 0**, a unidade de controle manda um sinal de “reset” e reseta todos os registradores para 0, inclusive regA, regB e Aluout.

No **estado 1**, a unidade de controle manda um sinal de “SetTest” que seta os valores dos registradores do Banco de Registradores para valores pré determinados no código do componente:
R0 recebe 00000001
R1 recebe 00000010
R2 recebe 00000011
R3 recebe 00000100

Soma



Estado 2 - Fetch
UCout = 010000
State = 3

Estado 3 - Decode
UCout = 000000
State = 4

Estado 4 - Execute
UCout = 000000 (soma)
State = 5

Estado 5 - Write Back
UCout = 000001
State = 2

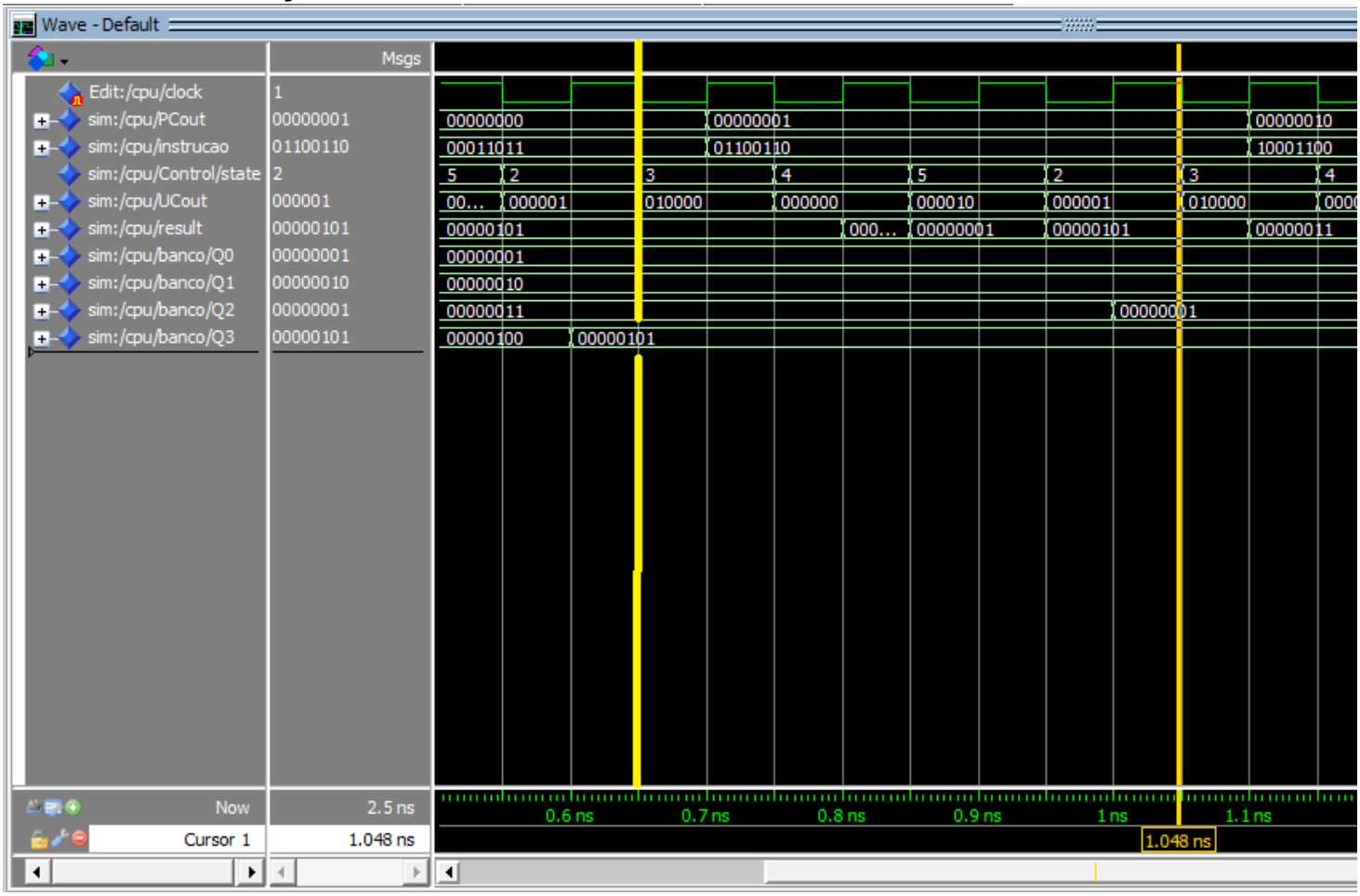
PCout = 0 | Instrução 00011011

Soma R1 com R2 e coloca o resultado em R3.

Como mostra na simulação, Q1 = 10, Q2 = 11, então R1+R2 = 101 que é gravado no quarto ciclo de clock em Q3.

O sinal “result” mostra que a ULA somou corretamente os valores.

Subtração



Pcout = 1 | Instrução 01100110

Subtração de R2 com R1 e coloca o resultado em R2

Com Q2 = 11 e Q1 = 10, 11 - 10 = 01, que é gravado corretamente no oitavo ciclo de clock da imagem (quarto ciclo depois da primeira linha amarela) em Q2.

Para subtração, o UCout no estado 4 de execução passa a ser 000001, diferente do UCout da soma.

O resultado 00000001 não se perde no “write back” pois está gravado no Aluout, mesmo que no ciclo o result mude para 101.

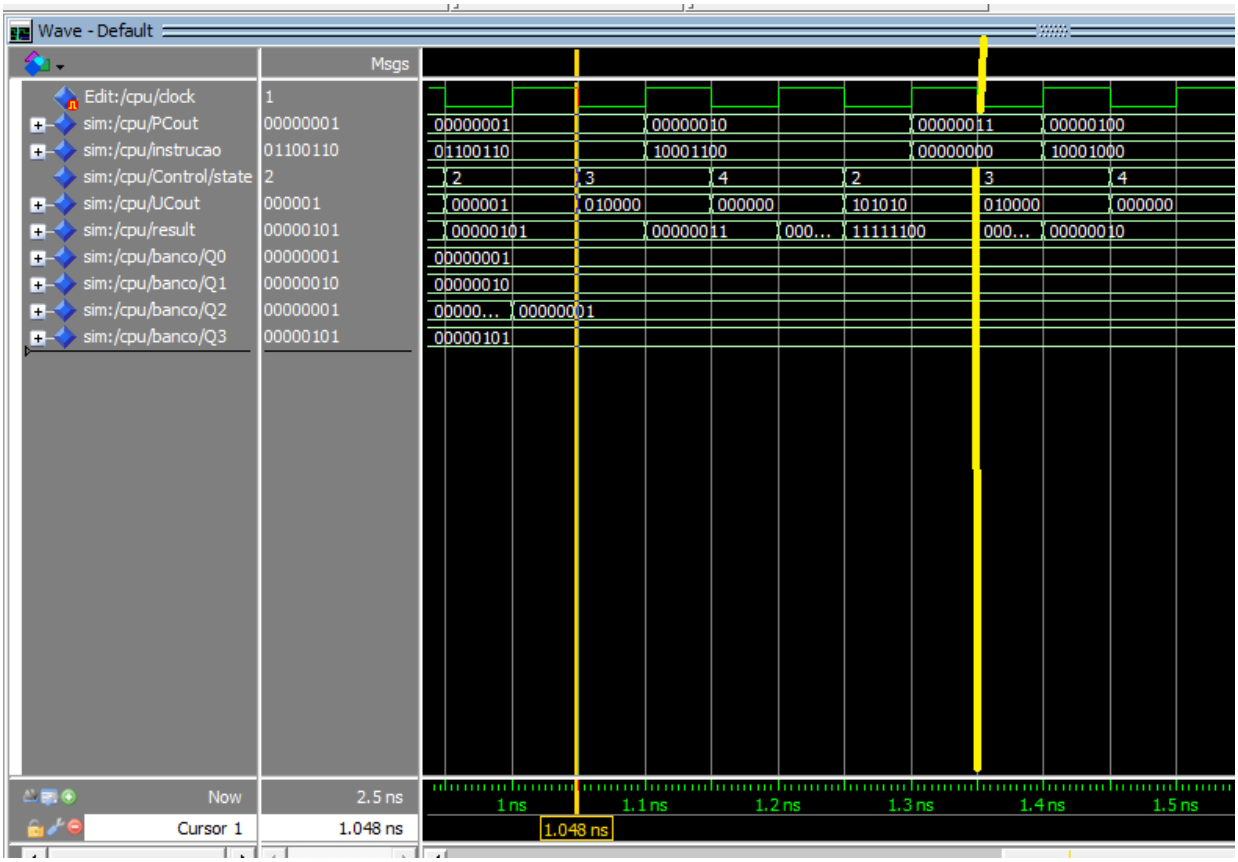
Branch Equal 1 - Not Taken

As instruções BEQ e JUMP rodam em 3 ciclos.

Estado 2 - Fetch
UCout = 010000
State = 3

Estado 3 - Decode
UCout = 000000
State = 4

Estado 4 - Execute
UCout = 101010 (branch)
= 010100 (jump)
State = 2

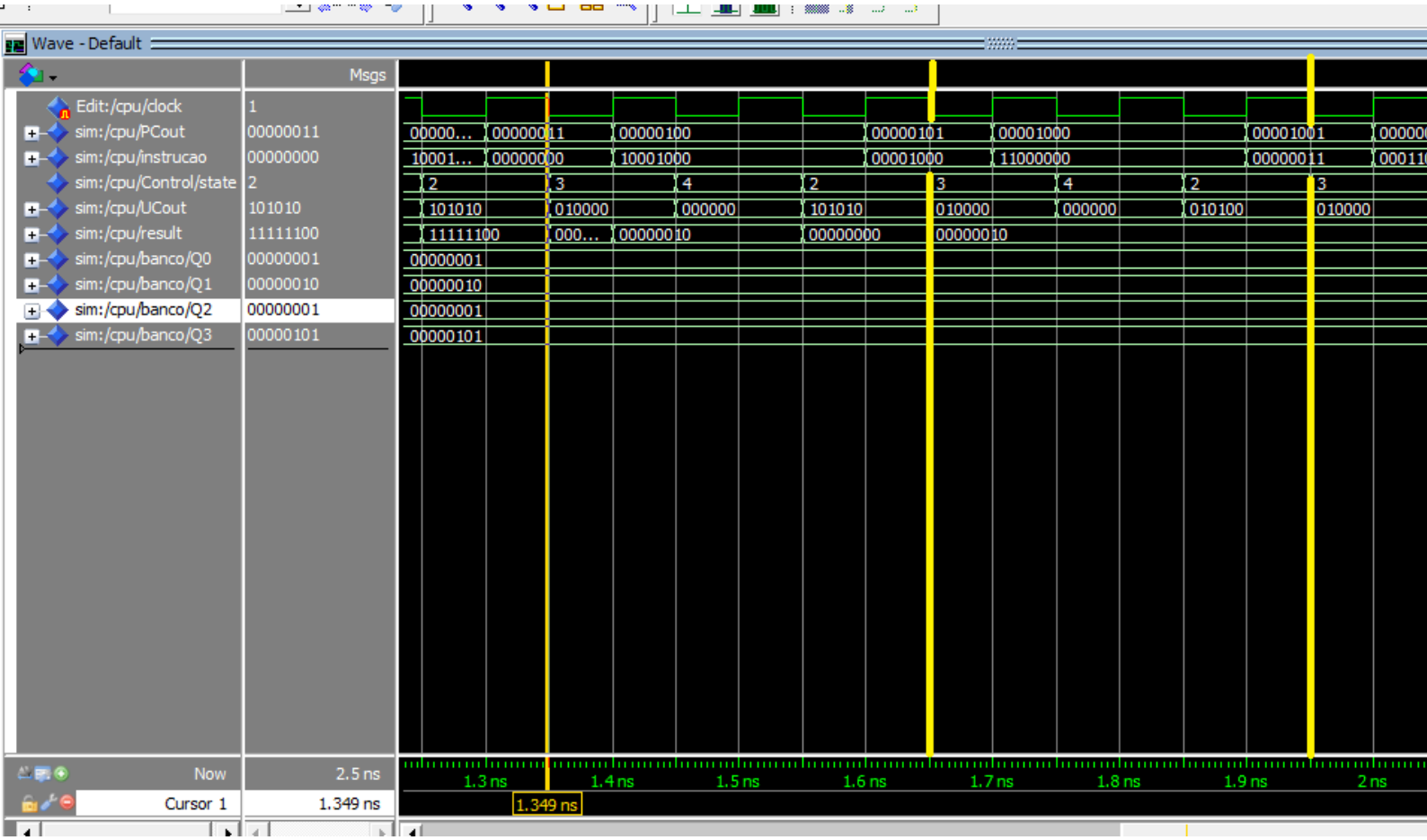


PCout = 10 | Instrução 10001100

Branch. Vai para o endereço que está na memória (00000000) caso R0 seja igual a R3. Q0 = 01 e Q3 = 101, então o branch vai ser “not taken”.

Quando há chamada de branch mas ocorre um “not taken”, a CPU precisa pular a próxima instrução na memória, que seria o endereço quando é “taken”, e ir para a próxima instrução válida. Nesse caso, PC estava na instrução 10 (2) e vai para a instrução 100 (10+1+1 = 100 = 4 em decimal).

Branch 2 e Jump



PCout = 100 | Instrução 10001000

Branch. Vai para o endereço que está na memória (00001000) caso R0 seja igual a R2. Q0 = 01 e Q2 = 01, então o branch vai ser “taken”.

Nesse caso, a CPU pega o endereço em memória, que seria 1000 (8 em decimal) e atualiza o PC para esse valor. Por isso que na próxima instrução o PCout vai para 00001000.

PCout = 1000 | Instrução 11000000

Jump. 11 é OPcode de jump, e 000000 é o endereço do jump. Concatenado com 00, o endereço completo é 00000000, que é atualizado em PC e vai para a primeira instrução da simulação, que seria 00011011, como visto no começo do texto.

4. Bibliografia

1 – BROWN, Stephen e VRANESIC, Svonko – Fundamentals of Digital Logic with
VHDL Design.

2 - PATTERSON, David A. e HENNESSY, John L. – Computer Organization and
Design – The Hardware and Software Interface

Agradecimento especial ao monitor LUIS MARCELO STEIN DAVILA que nos ajudou em nossas dúvidas e discussões sobre o projeto.

ANEXO

Código .vhd dos componentes.

CPU.vhd

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all;

ENTITY CPU IS
	GENERIC ( n : INTEGER := 8 ) ;
	PORT(
		-- Usar wave.do para instaciar o clock.
		clock		: IN STD_LOGIC;	-- Lembrar: Clock precisa
começar com 1. Unico sinal que precisa por pra CPU rodar.
		setTeste	IN STD_LOGIC
	);

END CPU ;

ARCHITECTURE Structure OF CPU IS

--- Instanciamento de componentes
	COMPONENT registrador
		PORT ( D	: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
				reset, load, clock	: IN STD_LOGIC ;
				Q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0)
			) ;
	END COMPONENT;

	COMPONENT banco_registradores
		PORT (RegWrite, clock, reset	: IN STD_LOGIC;
		ReadReg1, ReadReg2, WriteReg : IN STD_LOGIC_VECTOR (1
DOWNTO 0);
		WriteData	: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
		ReadData1, ReadData2	: OUT STD_LOGIC_VECTOR (n-1
DOWNTO 0);
		SetTest: IN STD_LOGIC --Para iniciar valores dos regs
	) ;
	END COMPONENT;

	COMPONENT Memoria
		PORT (
		PC_endereco: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
		instrucao_out: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
		BEQout: OUT STD_LOGIC_VECTOR (7 downto 0)
		);
	END COMPONENT;
```

```
COMPONENT Reg_Instrucao
        PORT (
        instrucao : IN STD_LOGIC_VECTOR(7 DOWNTO 0) ;
        clock  : IN STD_LOGIC ;
        Jumpin: OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
        OP, RS, RT, RD : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
        ) ;
END COMPONENT;

COMPONENT PC
        PORT (
        PCin,BEQin  : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
        reset, PCload, Clock: IN STD_LOGIC ;
        PCSource: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        PCout : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        PCmsb: OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
        ) ;
END COMPONENT;

COMPONENT ULA
        PORT(
        A: IN std_logic_vector (7 downto 0);
        B: IN std_logic_vector (7 downto 0);
        ALUop: IN std_logic;
        Result : OUT std_logic_vector (7 downto 0);
        Zero: OUT std_logic
        );
END COMPONENT;

COMPONENT UC
        PORT(
        OPin   : IN std_logic_vector(1 downto 0);
        Clock  : IN std_logic;
        reset, SetTest: OUT std_logic;
        UCSig : OUT std_logic_vector(5 downto 0)
        );
END COMPONENT;

COMPONENT Jumper
        PORT(
        PCmsb: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
        instadd: IN STD_LOGIC_VECTOR (5 DOWNTO 0);
        newpc : OUT STD_LOGIC_VECTOR (7 downto 0)
);
END COMPONENT;

--Sinais de PC
SIGNAL PCWrite: STD_LOGIC;
SIGNAL PCout: STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
SIGNAL PCmsb: STD_LOGIC_VECTOR(1 DOWNTO 0);
```

```
	SIGNAL Jumpin: STD_LOGIC_VECTOR(5 DOWNTO 0);

	--Sinais de Memoria
	--Pega o PCout de PC, usa o valor de indice em Vetor[i], e manda o conteudo para reg_int pelo sinal instrucao.

	--Sinais do Registrador de Instrução
	SIGNAL RegIntLoad: STD_LOGIC ;
	SIGNAL instrucao: STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
	SIGNAL OP, RS, RT, RD  : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
	SIGNAL ENDtoPC: STD_LOGIC_VECTOR(n-1 downto 0);
	SIGNAL BEQ: STD_LOGIC_VECTOR (n-1 downto 0);

	--Sinais de Banco de Registradores
	SIGNAL RegWrite: STD_LOGIC;
	SIGNAL WriteData, ReadData1, ReadData2    : STD_LOGIC_VECTOR (n-1 DOWNTO 0);

	--Sinais da ULA
	SIGNAL AluOp: STD_LOGIC ;
	SIGNAL Zero: STD_LOGIC ;
	SIGNAL A, B, result: STD_LOGIC_VECTOR (n-1 downto 0);

	--Registradores
	--SIGNAL loadA, loadB, loadALUout: STD_LOGIC;

	-- UC
	SIGNAL UCout: STD_LOGIC_VECTOR(5 downto 0);
	SIGNAL PCWriteCond, reset: STD_LOGIC;
	SIGNAL PCSource: STD_LOGIC_VECTOR(1 downto 0);
	SIGNAL SetTest: STD_LOGIC;


	BEGIN
		--- Separa os sinais que saem da unidade de controle.
		PCWriteCond <= UCout(5);
		PCWrite <= UCout(4) OR (PCWriteCond AND zero);
		PCSource <= UCout(3 downto 2);
		Aluop <= UCout(1);
		RegWrite <= UCout(0);

		--Instanciação dos componentes

		PC_reg: PC port map (ENDtoPC, BEQ, reset, PCWrite, clock, PCSource, PCout, PCmsb);
			--PCout = PCin se PCload=1, se não PCout = valor anterior + 1

		Mem: Memoria port map (Pcout, instrucao,BEQ);
```

```
		--Pega o valor de PC, transforma num int, e busca a instrucao num vetor pra mandar pro reg_int

		reg_int: Reg_Instrucao port map (instrucao, clock, Jumpin, OP, RS, RT, RD);
		--Pega a instrucao da memoria e quebra ela em 4 sinais de 2 bits cada


		banco: banco_registradores port map (RegWrite, clock, reset, RS, RT, RD, WriteData, ReadData1, ReadData2, SetTest);
			--ReadReg1 = RS / ReadReg2 = RT / WriteReg = RD
			--Usa os sinais do Reg_Int para saber os valores de quais registradores vão pra ULA ou são editados

		ULA1: ULA port map (A, B, AluOp, result, zero);
			--Recebe os valores do RegA e RegB e soma eles ou subtrai, de acordo com AluOp.
			--Resultado sai pelo sinal result

		Jump: Jumper port map (PCmsb, Jumpin, ENDTOPC);
		--Registradores

		Aluout: registrador port map (result, reset, '1', clock, WriteData);
		regA: registrador port map (ReadData1, reset, '1', clock, A);
		regB: registrador port map (ReadData2, reset, '1', clock, B);
			--Pega os valores de RS e RT.
			--Obs: load no reg A, B e no Aluout são sempre 1.

		Control: UC port map (OP, clock, reset, SetTest, UCOut);
			-- Manda os sinais de controle pros outros componentes pelo UCout
			-- Decide que sinais vai mandar pelo OP que recebe
			-- Manda o sinal de reset para os outros componentes
			-- SetTest é só pra iniciar valores nos registradores

END Structure ;
```

UnidadeDeControle.vhd

```
--- Recebe o sinal de OP e traduz esse sinal para todos os sinais de controle
usados na CPU
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY UC IS
	PORT(
		OPin			: IN std_logic_vector(1 downto 0);
		Clock			: IN std_logic;
		Reset, SetTest		: OUT std_logic;
		UCSign			: OUT std_logic_vector(5 downto 0)
		);
END ENTITY;

Architecture Behavior OF UC IS

SIGNAL state: integer := 0;

BEGIN
	PROCESS (Clock)
	BEGIN
		IF Clock'EVENT AND Clock = '0' THEN

			CASE state IS
				-- Reset nos registradores.
				WHEN 0 => UCSign <= "000000";
						Reset <= '1';
						SetTest <= '0';
						state <= 1;

				-- Set dos valores iniciais do registradores.
				WHEN 1 => UCSign <= "000001";
						Reset <= '0';
						SetTest <= '1';
						state <= 2;

				-- Fetch
				WHEN 2 => UCSign <= "010000";
						SetTest <= '0';
						state <= 3;
				-- Decode
				WHEN 3 => UCSign <= "000000";
						state <= 4;
				-- Execute
				WHEN 4 =>
					IF OPin = "00" THEN
						UCSign <= "000000";
						state <= 5;
```

```
                                        ELSIF OPin = "01" THEN
                                                UCSign <= "000010";
                                                state <= 5;
                                        ELSIF OPin = "10" THEN
                                                UCSign <= "101010";
                                                state <= 2;
                                        ELSE
                                                UCSign <= "010100";
                                                state <= 2;
                                END IF;

                        -- Write Back
                        WHEN 5 => UCSign <= "000001";
                                        state <= 2;

                        WHEN OTHERS => state <= 2;
                END CASE;
        END IF;
    END PROCESS;
END Behavior;
```

Memoria.vhd

```
--Funcionamento
-- Manda a instrução 0, 1, 2 ou N do "vetor_instrucoes" de acordo com o
"to_integer" do "PC_endereco".
-- Entrada PC_endereco / Saida instrucao_out

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;


ENTITY Memoria IS
	PORT (
		PC_endereco: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
		instrucao_out: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
		BEQout: OUT STD_LOGIC_VECTOR (7 downto 0)
	);
END Memoria;

ARCHITECTURE Behavior OF Memoria IS
	TYPE vetor_instrucoes IS ARRAY (0 TO 12) of STD_LOGIC_VECTOR
(7 DOWNTO 0);


	SIGNAL int_address: INTEGER RANGE 0 TO 31;

	CONSTANT instrucao : vetor_instrucoes:=
("00011011","01100110","10001100","00000000","10001000","00001000","000
00001","00000010","11000000","00000011","00000000","00000000","00000000
");

	BEGIN
		int_address <= to_integer(signed(PC_endereco));
		instrucao_out <= instrucao(int_address);
		BEQout <= instrucao(int_address + 1);

END Behavior;
```

PC.vhd

```
--Funcionamento
-- Nao tem entrada (só o clock). Começa com 00000000 e vai somando
00000001 a cada subida de clock.
-- Saida PCout conforme o sinal PCload (PCWrite) e o sinal PCSource.

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all;

ENTITY PC IS
	GENERIC ( N : INTEGER := 8 ) ;
	PORT ( PCin,BEQin : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
		reset, PCload, Clock: IN STD_LOGIC ;
		PCSource: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
		PCout : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
		PCmsb: OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
	);
END PC ;

ARCHITECTURE Behavior OF PC IS
SIGNAL intermediario: STD_LOGIC_VECTOR(N-1 DOWNTO 0) :=
"00000000";

BEGIN

	PROCESS (Clock)
	BEGIN
		IF Clock'EVENT AND Clock = '1' THEN
			IF reset = '1' THEN
				PCout <= ( OTHERS => '0' );
			ELSIF PCload = '1' AND PCSource = "01" THEN
				intermediario <= PCin;
				PCout <= intermediario;
			ELSIF PCload = '1' AND PCSource = "00" THEN
				intermediario <= intermediario + "00000001";
				PCOut <= intermediario;
			ELSIF PCload = '1' AND PCSource = "10" THEN
				intermediario <= BEQin;
				PCout <= intermediario;
			ELSIF PCload = '0' AND PCsource = "10" THEN
				intermediario <= intermediario + "00000001";
				PCOut <= intermediario;
			END IF;
		END IF;
	END PROCESS ;
	PCmsb <= intermediario(7 downto 6);
END Behavior ;
```

Reg_Instrucao.vhd

```
--Funcionamento:
-- Quebra a instrucao em OP, RS, RT e RD. Sinal "load" precisa ser 1 pra receber uma nova instrucao.
-- Entrada instrucao, load / Saida OP, RS, RT, RD

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
use ieee.numeric_std.all;

ENTITY Reg_Instrucao IS
	PORT (	instrucao: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
		Clock  : IN STD_LOGIC ;
		Jumpin: OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
		OP, RS, RT, RD: OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
	) ;
END Reg_Instrucao ;


ARCHITECTURE Behavior OF Reg_Instrucao IS
BEGIN
		Jumpin <= instrucao(5 downto 0);
		OP <=instrucao ( 7 downto 6 );
		RS	<=	instrucao ( 5 downto 4 );
		RT <= instrucao ( 3 downto 2 );
		RD <=instrucao ( 1 downto 0 );

END Behavior ;
```

Jumper.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;
--- Unidade que realiza Jump
ENTITY Jumper IS
PORT(
	PCmsb			: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
	instadd		: IN STD_LOGIC_VECTOR (5 DOWNTO 0);
	newpc		: OUT STD_LOGIC_VECTOR (7 downto 0)
);
END ENTITY;

ARCHITECTURE Behavior OF Jumper IS
BEGIN

	newpc(7 downto 6) <= PCmsb;
	newpc(5 downto 0) <= instadd;

END ARCHITECTURE;
```

banco_resgitradores.vhd

```
--- O banco de registradores consegue escrever e ler, manipulando qualquer
um de seus 4 registradores internos.
--- R0, R1, R2, R3.

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY banco_registradores IS
	GENERIC ( n : INTEGER := 8 ) ;

	PORT (RegWrite, clock, reset		: IN STD_LOGIC;
		ReadReg1, ReadReg2, WriteReg : IN STD_LOGIC_VECTOR (1
DOWNTO 0);
		WriteData                                       : IN
STD_LOGIC_VECTOR (n-1 DOWNTO 0);
		ReadData1, ReadData2                    : OUT
STD_LOGIC_VECTOR (n-1 DOWNTO 0);
		SetTest                                         : IN
STD_LOGIC
		) ;

END banco_registradores ;


ARCHITECTURE Structure OF banco_registradores IS
	SIGNAL Q0, Q1, Q2, Q3: STD_LOGIC_VECTOR (n-1 DOWNTO 0);
	SIGNAL D0, D1, D2, D3: STD_LOGIC_VECTOR (n-1 DOWNTO 0);

	COMPONENT registrador
		PORT ( D                                        : IN
STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
				reset, load, Clock    : IN STD_LOGIC ;
				Q                                       : OUT
STD_LOGIC_VECTOR(n-1 DOWNTO 0)
				) ;
	END COMPONENT ;


BEGIN
	--- Instanciamento de registradores.
	R0: registrador PORT MAP (D0, reset, RegWrite, clock, Q0);
	R1: registrador PORT MAP (D1, reset, RegWrite, clock, Q1);
	R2: registrador PORT MAP (D2, reset, RegWrite, clock, Q2);
	R3: registrador PORT MAP (D3, reset, RegWrite, clock, Q3);


	--- Escrita dos registradores.
```

```
        -- SetTest sรณ รฉ usado no comeรงo uma vez, sรณ que WriteReg inicia
com "11" quando nao รฉ instanciado, por isso que D3 precisa de um AND a
mais.
        D0      <=      WriteData WHEN RegWrite = '1' AND WriteReg = "00"
ELSE
                        "00000001" WHEN SetTest = '1' ELSE
                         Q0;
        D1      <=      WriteData WHEN RegWrite = '1' AND WriteReg = "01"
ELSE
                        "00000010" WHEN SetTest = '1' ELSE
                        Q1;
        D2      <=      WriteData WHEN RegWrite = '1' AND WriteReg = "10"
ELSE
                        "00000011" WHEN SetTest = '1' ELSE
                        Q2;
        D3      <=      WriteData WHEN RegWrite = '1' AND WriteReg = "11" AND
SetTest = '0' ELSE
                        "00000100" WHEN SetTest = '1' ELSE
                        Q3;

        --- Leitura dos registradores.
        ReadData1 <=        Q0      when ReadReg1 = "00"     else
                                    Q1      when ReadReg1 = "01"     else
                                    Q2      when ReadReg1 = "10"     else
                                    Q3      when ReadReg1 = "11"     else
                                    ( OTHERS => '0' ) ;

        ReadData2   <=      Q0      when ReadReg2 = "00"     else
                                    Q1      when ReadReg2 = "01"     else
                                    Q2      when ReadReg2 = "10"     else
                                    Q3 when ReadReg2 = "11"          else
                                    ( OTHERS => '0' ) ;

END Structure ;
```

registrador.vhd

```
--- Unidade básica de registrador, inicializado com 1 para testes de soma e sub
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY registrador IS
	GENERIC ( N : INTEGER := 8 ) ;
	PORT (	D 					: IN
STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
			reset, load, clock	: IN STD_LOGIC ;
			Q 					: OUT
STD_LOGIC_VECTOR(N-1 DOWNTO 0)
		) ;
END registrador ;

ARCHITECTURE Behavior OF registrador IS


BEGIN
	PROCESS (reset, load, clock )
	BEGIN
		IF clock'EVENT AND clock = '1' THEN
			IF reset = '1' THEN
				Q <= ( OTHERS => '0' );
				--Q <= "00000000";
			ELSIF load = '1' THEN
				Q <= D ;
			END IF;

		END IF ;
	END PROCESS ;
END Behavior ;
```

ULA.vhd

```
--- Unidade lógica e aritmética com 1 bit de Aluop, realiza funções simples de
'+' e '-' por meio de um ripple-carry.
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY ULA IS
	PORT(
		A: IN std_logic_vector (7 downto 0);
		B: IN std_logic_vector (7 downto 0);
		ALUop: IN std_logic;
		Result: OUT std_logic_vector (7 downto 0);
		Zero: OUT std_logic
		);
END ENTITY;

ARCHITECTURE ULA_arch of ULA IS
	COMPONENT ripple_carry
		PORT(
			Rj: IN std_logic_vector (7 downto 0);
			Rk: IN std_logic_vector (7 downto 0);
			Ri: OUT std_logic_vector (7 downto 0)
		);
	END COMPONENT;

	SIGNAL Bsig: std_logic_vector (7 downto 0);
	SIGNAL Sgen: std_logic_vector (7 downto 0);

BEGIN

	Bsig <= std_logic_vector( unsigned(NOT B) + 1) WHEN ALUop = '1'
ELSE
			B WHEN Aluop = '0' ELSE
			(OTHERS => '0');

	Zero <= (NOT (Sgen(0) OR Sgen(1) OR Sgen(2) OR Sgen(3) OR
Sgen(4) OR Sgen(5) OR Sgen(6) OR Sgen(7)));
	Result <= Sgen;
	SOMADOR1: ripple_carry PORT MAP(A,Bsig,Sgen);
END ARCHITECTURE;
```

ripple_carry.vhd

```
--- Unidade básica de soma
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY ripple_carry IS
	GENERIC ( n : INTEGER := 8 ) ;

PORT ( Rj, Rk: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0) ;
		Ri		: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0)
		) ;
END ripple_carry ;


ARCHITECTURE Structure OF ripple_carry IS

	SIGNAL C : STD_LOGIC_VECTOR(0 TO n) ;

	COMPONENT full_adder
		PORT ( Cin, x, y	: IN STD_LOGIC ;
				s, Cout			: OUT STD_LOGIC) ;
	END COMPONENT ;

BEGIN
	C(0) <= '0';


	Generate_label:
	FOR i IN 0 TO n-1 GENERATE
		stage: full_adder PORT MAP ( C(i), Rj(i), Rk(i), Ri(i), C(i+1) ) ;
	END GENERATE ;

END Structure ;
```

full_adder.vhd

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
--- Unidade básica de soma
ENTITY full_adder IS
PORT ( Cin, x, y : IN STD_LOGIC ;
		s, Cout : OUT STD_LOGIC ) ;
END full_adder ;

ARCHITECTURE LogicFunc OF full_adder IS

BEGIN
	s <= x XOR y XOR Cin ;
	Cout <= (x AND y) OR (x AND Cin) OR (y AND Cin) ;
END LogicFunc ;
```