Pontifícia Universidade Católica de Campinas – PUC CAMPINAS

Projeto 3 – CPU com Pipeline

Cesar Marrote Manzano RA: 18051755

Fabrício Silva Cardoso RA: 18023481

Matheus Henrique Moretti RA:18082974

Pedro Ignácio Trevisan RA:18016568

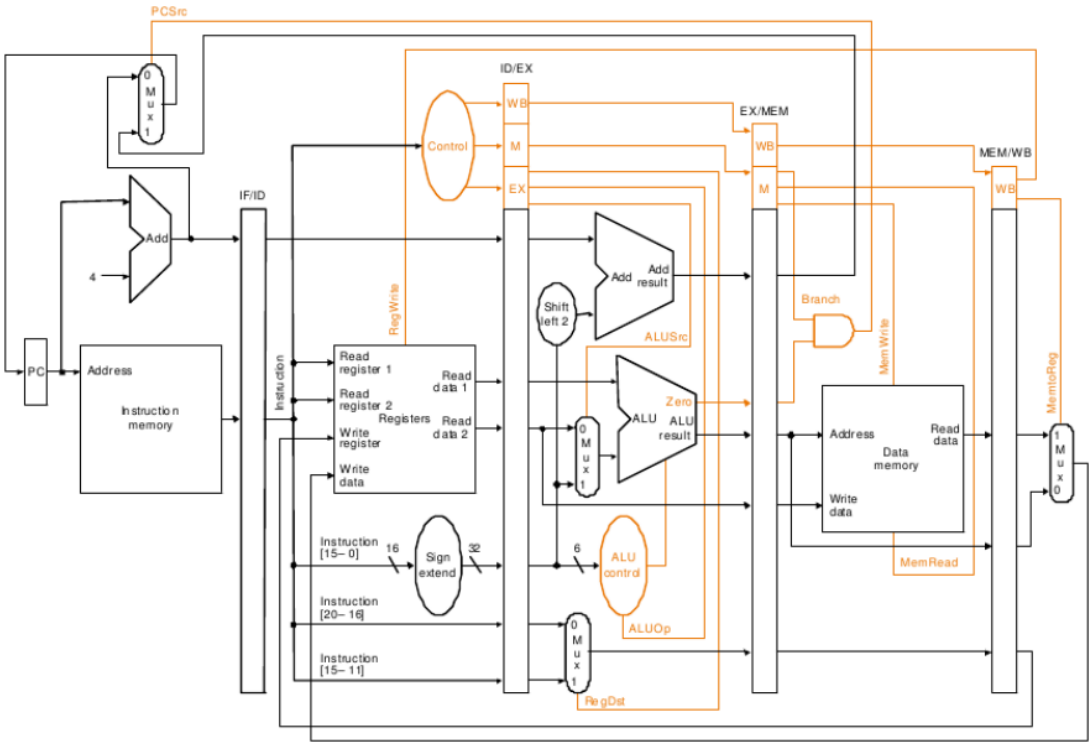
Índice

1. Introdução: descrição do que é projeto

O terceiro projeto teve como objetivo o desenvolvimento de uma CPU com pipeline, que executa as instruções abaixo:

Category	Name	Instruction syntax	Meaning	Format	Notes
Arithmetic	Add	add $1,$2,$3	$1 = $2 + $3	R	Adds two registers
	Subtract	sub $1,$2,$3	$1 = $2 – $3	R	Subtracts two registers
	Add immediate	addi $1,$2,CONST	$1 = $2 + CONST	I	Used to add constants
	Sub immediate	subi $1,$2,CONST	$1 = $2 - CONST	I	Used to sub constants
Data Transfer	Load word	lw $1,CONST($2)	$1 = Memory[$2 + CONST]	I	Loads the word stored from: MEM[$s2+CONST] and the following 3 bytes
	Store word	sw $1,CONST($2)	Memory[$2 + CONST] = $1	I	Stores a word into: MEM[$2+CONST] and the following 3 bytes
Logical	And	and $1,$2,$3	$1 = $2 & $3	R	Bitwise and
	And immediate	andi $1,$2,CONST	$1 = $2 & CONST	I	
	Or	or $1,$2,$3	$1 = $2 | $3	R	Bitwise or
	Or immediate	ori $1,$2,CONST	$1 = $2 | CONST	I	
Conditional branch	Branch on equal	beq $1,$2,CONST	if ($1 == $2) go to PC+4+CONST	I	Goes to the instruction at the specified address if two registers are equal
Unconditional jump	Jump	j CONST	goto address CONST	J	Unconditionally jumps to the instruction at the specified address
	Jump register	jr $1	goto address $1	R	Jumps to the address contained in the specified register

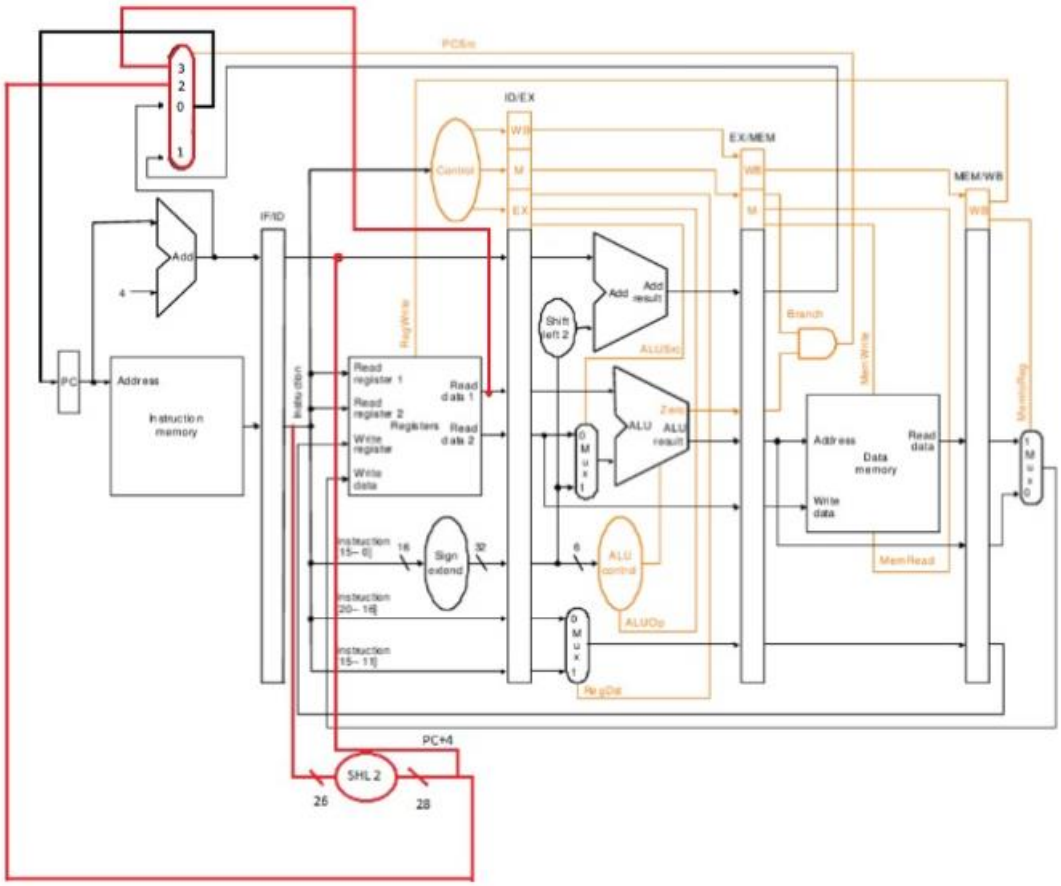
Inicialmente as instruções que serão usadas e verificadas estão no componente chamado de memória de instruções, cada instrução a de ser decodificada e carregada na etapa subsequente a sua leitura onde será separado as partes da instrução aos seus devidos lugares, separados e escolhidos os registradores, e a unidade de controle gerará os sinais que serão usados pela instrução (estes serão propagados entre as etapas, assim como outros dados, através dos registradores de pipeline). Na terceira etapa é onde ocorrem o cálculo do endereço do “branch” (desvio), e utilização da unidade lógica e aritmética, na quarta etapa é onde ocorre o acesso a memória de dados, e na quinta etapa ocorre o “write back” (escrita), portanto qualquer atualização a um registrador será feita nessa etapa.


PCSrc
M u x
0
1
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
Add
4
IF/ID
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add result
ALUSrc
ALU
Zero
ALU result
Branch
MemWrite
MemtoReg
Data memory
Read data
MemRead
Instruction [15– 0]
16
Sign extend
32
6
ALU contrd
ALUOp
Instruction [20– 16]
Instruction [15– 11]
RegDst

2. Especificação: descrição de todos os detalhes relevantes do projeto

INSTRUÇÃO	OPCODE	FORMATO
NOP	000000	---
ADD	000001	Tipo R
SUB	000010	Tipo R
ADDI	000011	Tipo I
SUBI	000100	Tipo I
LOADWORD	000101	Tipo I
SAVEWORD	000110	Tipo I
AND	000111	Tipo R
ANDI	001000	Tipo I
OR	001001	Tipo R
ORI	001010	Tipo I
BEQ	001011	Tipo I
JUMP	001100	Tipo J
JUMP REGISTER	001101	Tipo R

A tabela acima mostra todas as operações feitas pela CPU com pipeline, e o tamanho das instruções são de 32-bits, porém o JUMP e o JUMP REGISTER são instruções que não são feitas pelo datapath acima, então tivemos que modificar o datapath para que essas instruções fossem executadas (as modificações feitas estão em vermelho).



A modificação feita no datapath para que execute o JUMP e o JUMP REGISTER foram as seguintes, para o JUMP, pegar o endereço que veio da instrução (que inicialmente possui 26 bits), passar pelo shift left 2 transformando em 28-bits, em seguida, ele concatena com os 4 bits mais significativos de PC+4 e se torna uma das opções do multiplexador de PCSource; para o JUMP REGISTER o endereço provindo do registrador escolhido (rs) é usado diretamente como uma opção no PCSource, assim dependendo da instrução que está sendo executada o sinal de PCSource muda de acordo com o necessário.

3. Desenvolvimento: descrição de todos os detalhes relevantes da implementação

Os componentes usados para implementar o projeto proposto são: AddBranch, Data Memory, IF_ID, ID_EX, Instruction Memory, EX_MEM, MEM_WB, MuxRegDst, MUXs_32bits, PC, Registers, Shift Left 2, Sign Extend, ULA e Unidade Controle e suas devidas funcionalidades são, respectivamente:

- O AddBranch recebe o endereço do PC+4 e o valor do endereço *shiftado* e os soma, dando o endereço do Branch;

- O Data Memory pode ler ou escrever um dado a partir do endereço desejado.;

- O registrador de Pipeline IF_ID recebe a instrução a ser executada e recebe o endereço da próxima instrução (PC+4);

- O registrador IF_EX recebe o conteúdo dos registradores rs e rt, contém o enderenço com sinal estendido;

- O EX_MEM recebe o endereço do Branch, o resultado de ALU, o sinal do ALU e o registrador que guardará o conteúdo;

- O MEM_WB recebe o dado lido do Data Memory e a saída da ALU;

- O Instruction Memory tem as instruções a serem executadas pela CPU;

- O MuxRegDst escolhe o registrador destino;

- O Muxs_32bits recebe duas entradas e escolhe uma delas. Esses multiplexadores são implementados na segunda entrada da ULA e no estágio de WB;

- O PC recebe o endereço da próxima instrução;

- O Shift Left 2 recebe o sign extend e realiza a operação Shift Left duas vezes;

- O Sign Extend estende o bit do sinal até o bit 31. A ULA realiza operações logicas e aritméticas, que no programa são: adição, subtração, AND e OR;

- A Unidade de Controle é responsável pelos sinais do Pipeline;

- Os registradores (Registers) são os componentes responsáveis para guardar dados que tem acesso mais rápido que a memória.

A respeito de como as instruções são lidas e interpretadas no código funcionam da seguinte forma:

Tipo R:

```
Inst(x)      <= “aaaaaabb”
Inst(x+1)    <= “bbbccccc”
```

Inst(x+2) <= “dddddeee”

Inst(x+3) <= “eeffffff”

Sendo, inst(x) o byte da instrução (as quatro linhas de inst(x+n) formam uma instrução), (a) os bits do opcode, (b) os bits do Rs, (c) os bits do Rt, (d) os bits do Rd, (e) os bits do shift amount, e (f) os bits do function.

Tipo I

Inst(x) <= “aaaaaabb”

Inst(x+1) <= “bbbccccc”

Inst(x+2) <= “ddddddd”

Inst(x+3) <= “dddddddd”

Sendo, inst(x) o byte da instrução (as quatro linhas de inst(x+n) formam uma instrução), (a) os bits do opcode, (b) os bits do Rs, (c) os bits do Rt, (d) os bits do valor imediato.

4. Resultados: descrição e imagens dos testes realizados demonstrando o funcionamento da CPU.

Na tabela abaixo é mostrado a ordem de execução das instruções utilizadas no projeto para os testes.

Instrução	Código binário
ADDI Reg1, Reg0, 7	00001100000000010000000000000111
ADDI Reg2, Reg0, 5	00001100000000010000000000000101
ADDI Reg3, Reg0, 3	00001100000000011000000000000011
NOP	00000000000000000000000000000000
NOP	00000000000000000000000000000000
NOP	00000000000000000000000000000000
ADD Reg1, Reg2, Reg3	00000100010000110000100000000000
SUB Reg1, Reg2, Reg3	00001000010000110000100000000000
AND Reg1, Reg2, Reg3	00011100010000110000100000000000
OR Reg1, Reg2, Reg3	00100100010000110000100000000000
SUBI Reg1, Reg2, 3	00010000010000010000000000000011
ANDI Reg1, Reg2, 7	00100000010000010000000000000111
ORI Reg1, Reg2, 7	00101000010000010000000000000111
SW Reg2, 0(Reg7)	00011000111000100000000000000000
NOP	00000000000000000000000000000000
NOP	00000000000000000000000000000000
NOP	00000000000000000000000000000000
LW Reg5, 0(Reg7)	00010100111001010000000000000000
ADDI Reg1, 0, 0	00001100000000010000000000000000

NOP	00000000000000000000000000000000
NOP	00000000000000000000000000000000
BEQ Reg1, Reg0, 27	00101100001000000000000000011011
NOP	00000000000000000000000000000000
NOP	00000000000000000000000000000000
NOP	00000000000000000000000000000000
NOP	00000000000000000000000000000000
NOP	00000000000000000000000000000000
ADD Reg1, Reg2, Reg3	00000100010000110000100000000000
J 30	00110000000000000000000000011110
NOP	00000000000000000000000000000000
NOP	00000000000000000000000000000000
ADDI Reg1, Reg0, 4	00001100000000010000000000000100

Fotos dos testes:

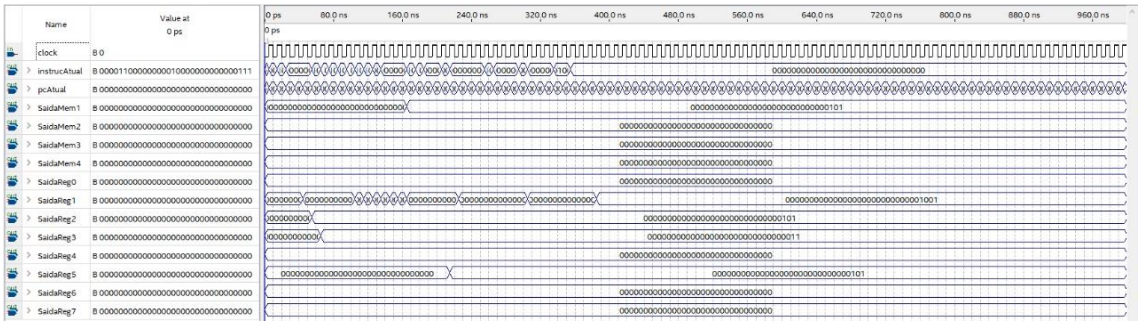


Figura 1 Visao geral dos testes

As fotos abaixo mostrarão detalhadamente a execução de cada execução (ignorando os NOPs).

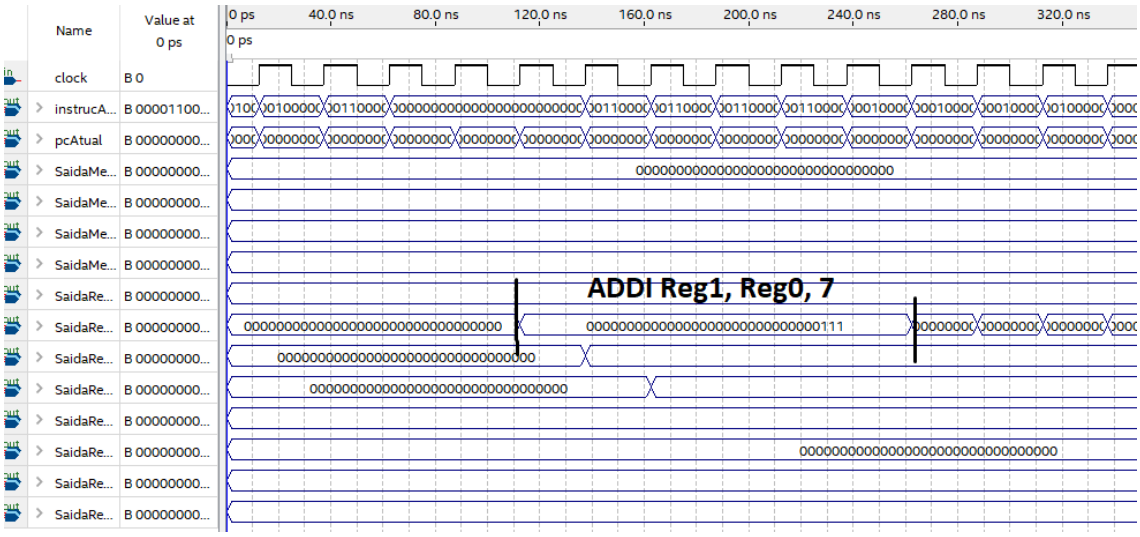


Figura 2 Primeira instrucao de addi

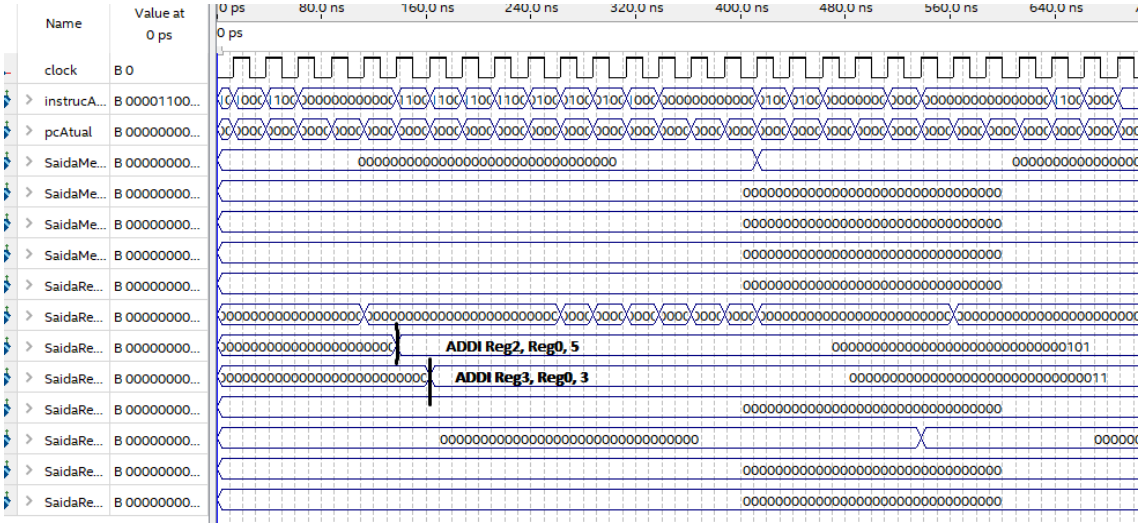


Figura 3 Instrucões 2 e 3 de addi

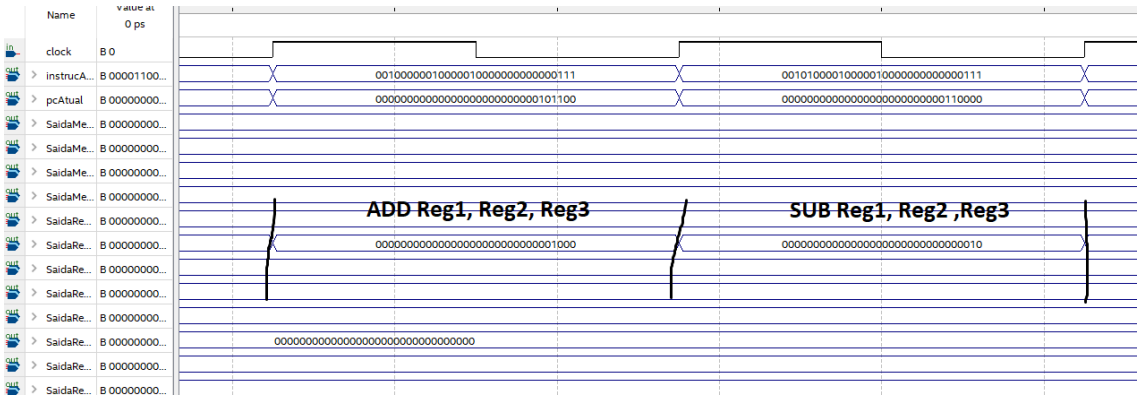


Figura 4 Instruções de add e sub

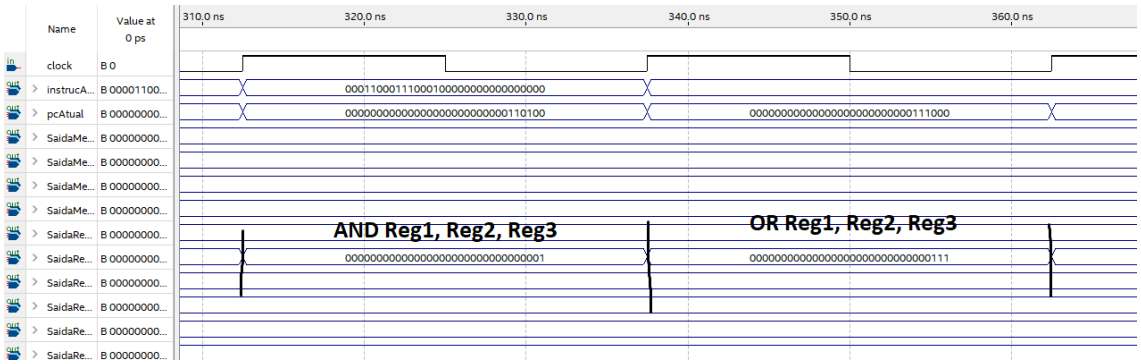


Figura 5 Instruções de and e or

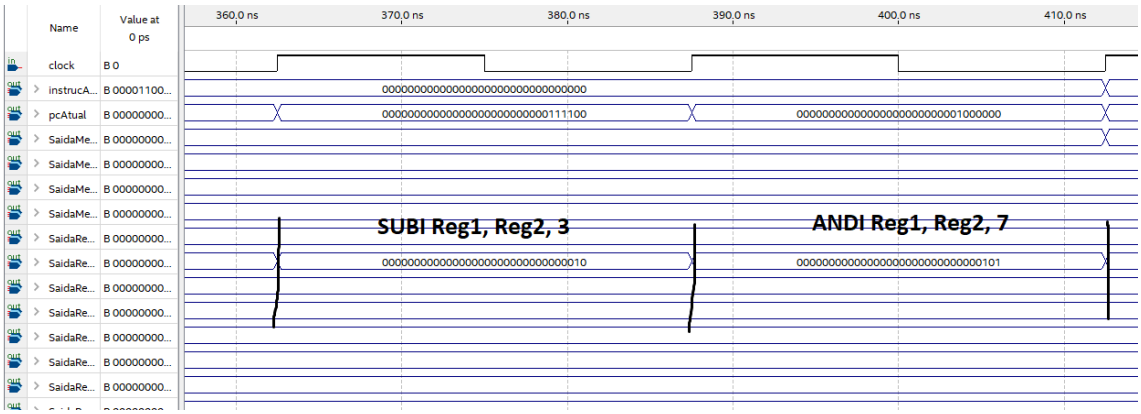


Figura 6 Instruções de subi e andi

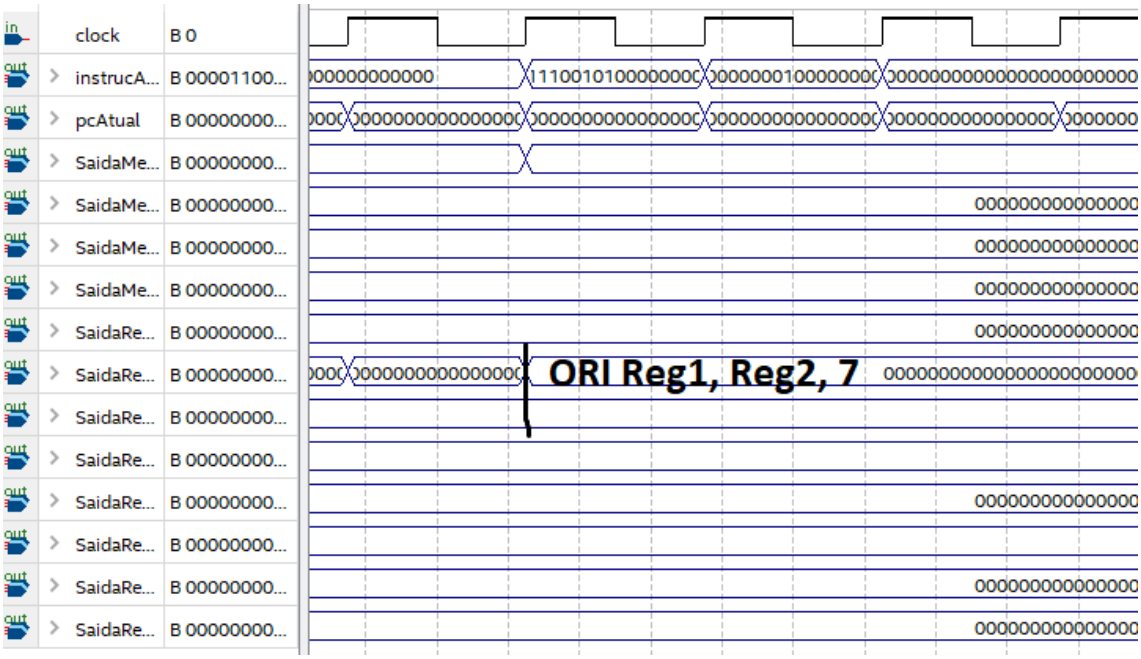


Figura 7 Instrucao de ori

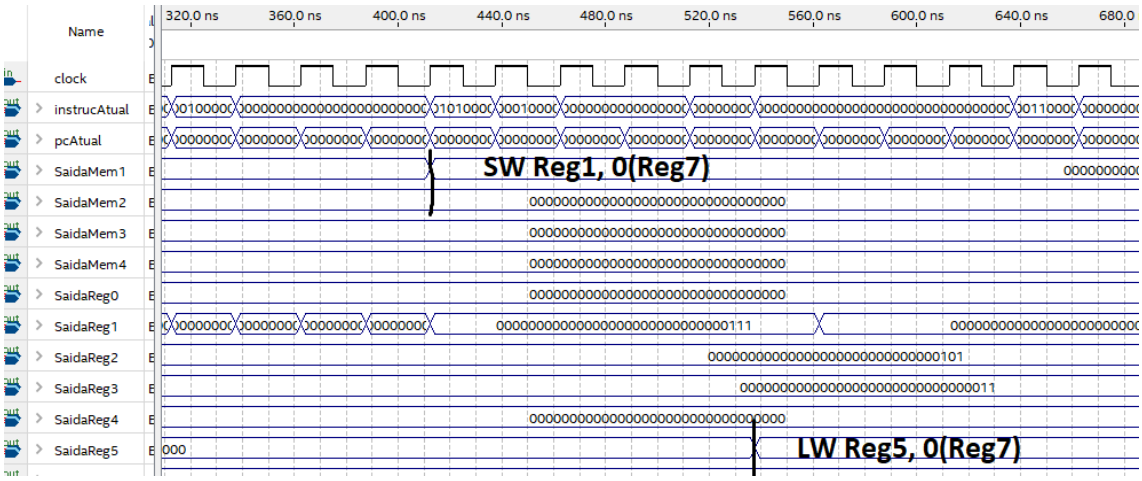


Figura 8 Instruções de sw e lw

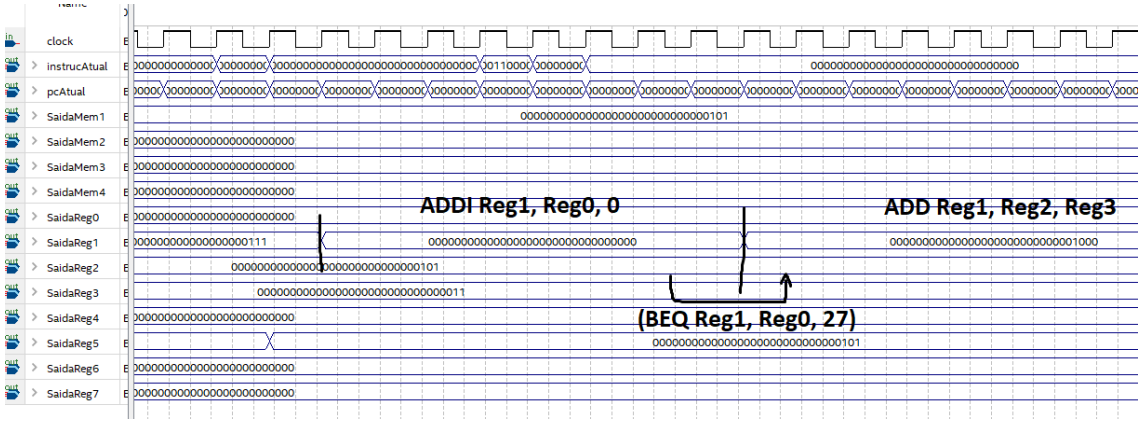


Figura 9 Instrução de beq

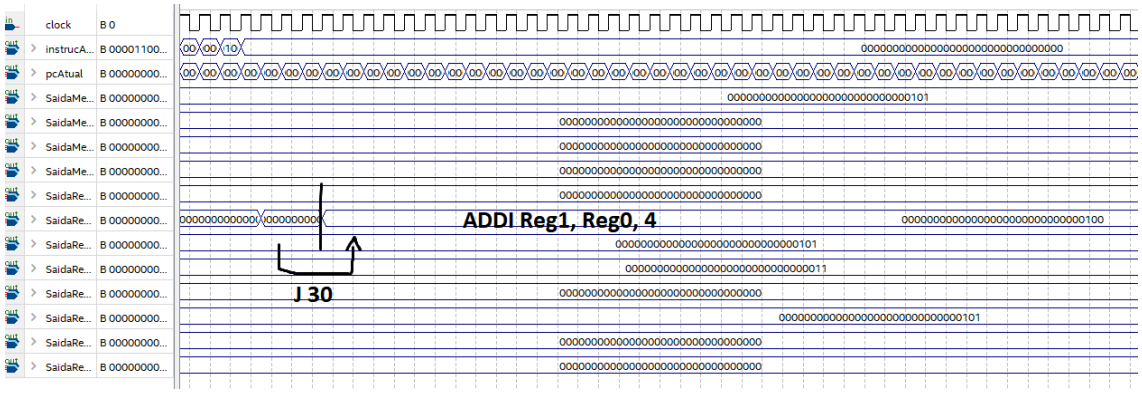


Figura 10 Instrução de jump

5. Conclusões: análise sobre o que foi inicialmente proposto e os resultados alcançados

Todas as instruções testadas obtiveram resultado totalmente positivo, com execeção da instrução de Jump Register (jr). Primeiramente foram testadas as instruções lógicas e aritméticas (add, addi, sub, subi, and, andi, or e ori), depois as instruções que envolviam acesso à memória (lw e sw) e por fim as instruções de desvio (beq, j e jr). Separamos dessa forma para facilitar a análise dos testes e corrigir mais facilmente eventuais erros.

6. Bibliografia: referências bibliográficas utilizadas; Anexos - Código VHDL produzido

Brown, s.; Vranesic, z. Fundamentals of digital logic with vhdl design, mcgraw hill, 2005.

AdderBranch

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

-- Declaração das variaveis
entity AdderBranch is

    port(
        entrada1: in std_logic_vector(0 to 31);
        entrada2: in std_logic_vector(0 to 31);
        saidaAdder: out std_logic_vector(0 to 31)
        );

end AdderBranch;

-- Faz a soma dos valores que estão nas entradas, e fazem com que a saida receba esse valor
architecture a of AdderBranch is
begin
    saidaAdder <= entrada1 + entrada2;
end;
```

CPU

```vhdl
-- Declaração das variaveis no CPU
entity CPU is
    port(clock: in std_logic; -- Controle das instruções
            pcAtual: out std_logic_vector(0 to 31); -- Responsável por mostrar o PC Atual
            instrucAtual: out std_logic_vector(0 to 31); -- Contém a instrução que será executada
            SaidaReg0: out std_logic_vector(0 to 31);
            SaidaReg1: out std_logic_vector(0 to 31);
            SaidaReg2: out std_logic_vector(0 to 31);
            SaidaReg3: out std_logic_vector(0 to 31);
            SaidaReg4: out std_logic_vector(0 to 31);
            SaidaReg5: out std_logic_vector(0 to 31);
            SaidaReg6: out std_logic_vector(0 to 31);
            SaidaReg7: out std_logic_vector(0 to 31);
            SaidaMem1: out std_logic_vector(0 to 31);
            SaidaMem2: out std_logic_vector(0 to 31);
            SaidaMem3: out std_logic_vector(0 to 31);
            SaidaMem4: out std_logic_vector(0 to 31)

        --  debug_ULA0: out std_logic_vector(0 to 31);
        --  debug_ULA1: out std_logic_vector(0 to 31);
        --  debug_ULAOut: out std_logic_vector(0 to 31);
        --  debug_AddrDM: out std_logic_vector(0 to 31);
        --  debug_WriteDataDM: out std_logic_vector(0 to 31);
        --  debug_MemRead:out std_logic;
        --  debug_MemWrite:out std_logic;
        --  debug_AddBranch:out std_logic_vector(0 to 31);
        --  debug_BranchSig:out std_logic;
        --  debug_PCSrc: out std_logic_vector(0 to 31)
        --debug_branch:out std_logic;
        --debug_Zero:out std_logic
            );

end CPU;
```

```vhdl
    -- Declaração dos componentes utilizados durante a execução do pipeline
architecture components of CPU is

    --OBS: Explicação de cada componente se encontra no proprio arquivo do componente


    -- Declaração do PC, utilizado ao longo do programa
    component PC is

        port(clock: in std_logic;
                pc4: in std_logic_vector(0 to 31);
                pc: out std_logic_vector(0 to 31));

    end component;


    -- Declaração da Memoria de Instruções, responsavel por obter as instruções do programa
    component InstructionMemory

        port (endereco: in std_logic_vector(0 to 31);
                instrucao: out std_logic_vector(0 to 31));

    end component;
```

```
-- Declaração dos registradores usados pelo programa
component Registers

    port(regWrite: in std_logic;
            clock: in std_logic;
            readRegister1: in std_logic_vector(0 to 4);
            readRegister2: in std_logic_vector(0 to 4);
            writeRegister: in std_logic_vector(0 to 4);
            writeData: in std_logic_vector(0 to 31);
            readData1: out std_logic_vector(0 to 31);
            readData2: out std_logic_vector(0 to 31);
            register1: out std_logic_vector(0 to 31);
            register2: out std_logic_vector(0 to 31);
            register3: out std_logic_vector(0 to 31);
            register4: out std_logic_vector(0 to 31);
            register5: out std_logic_vector(0 to 31);
            register6: out std_logic_vector(0 to 31);
            register7: out std_logic_vector(0 to 31);
            register8: out std_logic_vector(0 to 31));

end component;


-- Declaração da memoria de Dados
component DataMemory

    port(endereco: in std_logic_vector(0 to 31);
            clock: in std_logic;
            memWrite: in std_logic;
            writeData: in std_logic_vector(0 to 31);
            memRead: in std_logic;
            readData: out std_logic_vector(0 to 31);
            mem1: out std_logic_vector(0 to 31);
            mem2: out std_logic_vector(0 to 31);
            mem3: out std_logic_vector(0 to 31);
            mem4: out std_logic_vector(0 to 31));

end component;
```

```
-- Declaração do somador responsavel por calcular o Branch se necessario
component AdderBranch

    port(entrada1: in std_logic_vector(0 to 31);
         entrada2: in std_logic_vector(0 to 31);
         saidaAdder: out std_logic_vector(0 to 31));

end component;


-- Declaração do multiplexador de 32 bits de forma generica, para multiplos usos ao longo do programa
component Muxs_32bits is

    port(entrada1: in std_logic_vector(0 to 31);
         entrada2: in std_logic_vector(0 to 31);
         sinalControle: in std_logic;
         saidaMux32: out std_logic_vector(0 to 31));

end component;


-- Declaração do multiplexador usado para o controle do RegDst
component MuxRegDst is

    port(regRt: in std_logic_vector(0 to 4);
         regRd: in std_logic_vector(0 to 4);
         regDst: in std_logic;
         saidaMux: out std_logic_vector(0 to 4));

end component;


 -- Declaração do sinal extendido, necessario para o bom funcionamento do pipeline
 component SignExtend is

     port(entradaSE: in std_logic_vector(0 to 15);
          saidaSE:    out std_logic_vector(0 to 31));

 end component;

     -- Declaração do shitf left 2, para o programa apenas é necessario mover 2 bits
 component ShiftLeft2 is

     port(entradaSHL: in std_logic_vector(0 to 31);
          saidaSHL: out std_logic_vector(0 to 31));

 end component;

     -- Declaração da unidade Logica e Aritmetica, responsavel pelas operações necessarias durante o programa
 component ULA is

     port(aluSrcA: in std_logic_vector(0 to 31);
          aluSrcB: in std_logic_vector(0 to 31);
          aluOp: in std_logic_vector(0 to  1);
          aluResult: out std_logic_vector(0 to 31);
          zero: out std_logic);

 end component;
```

```
-- Registrador IF/ID
    component IF_ID is

    port(clock: in std_logic;
            pcIn: in std_logic_vector(0 to 31);
            pcOut: out std_logic_vector(0 to 31);
            InstructionIn:  in std_logic_vector(0 to 31);
            InstructionOut: out std_logic_vector(0 to 31));

end component;

-- Registrador ID/EX
    component ID_EX is

    port(clock: in std_logic;
            entradaWB: in std_logic_vector(0 to 1);
            entradaMEM: in std_logic_vector(0 to 2);
            entradaEX: in std_logic_vector(0 to 3);
            saidaWB: out std_logic_vector(0 to 1);
            saidaMEM: out std_logic_vector(0 to 2);
            saidaEX: out std_logic_vector(0 to 3);
            entradaPC: in std_logic_vector(0 to 31);
            saidaPC: out std_logic_vector(0 to 31);
            entrada_ReadData1: in std_logic_vector(0 to 31);
            saida_ReadData1: out    std_logic_vector(0 to 31);
            entrada_ReadData2: in std_logic_vector(0 to 31);
            saida_ReadData2: out    std_logic_vector(0 to 31);
            entrada_imed: in std_logic_vector(0 to 31);
            saida_imed: out std_logic_vector(0 to 31);
            entradaRT: in std_logic_vector(0 to 4);
            saidaRT: out std_logic_vector(0 to 4);
            entradaRD: in std_logic_vector(0 to 4);
            saidaRD: out    std_logic_vector(0 to 4));

end component;
```

```
-- Registrador EX/MEM
    component EX_MEM is

    port(clock:in std_logic;
            entradaWB: in std_logic_vector(0 to 1);
            entradaMEM: in std_logic_vector(0 to 2);
            saidaWB: out std_logic_vector(0 to 1);
            saidaMEM: out std_logic_vector(0 to 2);
            entradaPC: in std_logic_vector(0 to 31);
            saidaPC: out std_logic_vector(0 to 31);
            entradaZERO: in std_logic;
            saidaZERO: out std_logic;
            entradaResultado: in std_logic_vector(0 to 31);
            saidaResultado: out std_logic_vector(0 to 31);
            entrada_DadoWriteRegister: in std_logic_vector(0 to 31);
            saida_DadoWriteRegister: out    std_logic_vector(0 to 31);
            entrada_RegDST: in std_logic_vector(0 to 4);
            saida_RegDST: out std_logic_vector(0 to 4));

end component;


-- Registrador MEM/WB
    component MEM_WB is

    port(clock: in std_logic;
            wbIn: in std_logic_vector(0 to 1);
            wbOut: out std_logic_vector(0 to 1);
            readDataIn: in std_logic_vector(0 to 31);
            readDataOut: out std_logic_vector(0 to 31);
            addrIn: in std_logic_vector(0 to 31);
            addrOut: out std_logic_vector(0 to 31);
            regDstIn: in std_logic_vector(0 to 4);
            regDstOut: out std_logic_vector(0 to 4));

end component;
```

```
-- Declaração da Unidade de Controle, responsavel pelas "escolha" das operações.
    component UnidadeControle is

    port(opcode:    in std_logic_vector(0 to 5);
            PCSrc: out std_logic;
            jumpSignal: out std_logic := '0';
            SignalWB: out std_logic_vector(0 to 1);
            SignalMEM: out std_logic_vector(0 to 2);
            SignalEX: out std_logic_vector(0 to 3));

end component;

-------------------------------------------------------------------------------------------------------------


-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------


signal deb_reg1: std_logic_vector(0 to 31);
-------------------------------------------------------------------------------------------------------------
-- Fetch da Instrução
-- Sinais responsaveis por fazer o fetch da instrução

signal instrucaoPC: std_logic_vector(0 to 31); -- Instrução vai ser feita o fetch

signal instrucao_IF_ID: std_logic_vector(0 to 31); -- Instrução pós fetch ||
Saida da memoria de instrução e entrada no registrador IF/ID

signal PCSrc: std_logic; -- PC Source responsavel pelo controle do que vai passar para o PC

signal PCSrc_0: std_logic_vector(0 to 31); -- Entrada do 0 no Mux do PCSrc - PC + 4 - Proxima instrução

signal PCSrc_1: std_logic_vector(0 to 31); -- Entrada do 1 no Mux do PCSrc - Pos Branch

signal atualizaPC: std_logic_vector(0 to 31); -- Responsavel por atualizar o
--PC baseado na saida do multiplexador do PCSrc

-------------------------------------------------------------------------------------------------------------

signal PC_plus4:std_logic_vector(0 to 31); -- Sinal responsavel por receber o PC + 4

signal Sinal_regWrite: std_logic; -- Sinal do regWrite - Responsavel pelo controle de escrita ou não do registrador.

signal instrucao: std_logic_vector(0 to 31); -- Responsavel por receber a instrução que será decoficada

signal OPCode: std_logic_vector(0 to 5); -- Recebe o OPCode

signal ReadReg1: std_logic_vector(0 to 4); -- Responsavel por receber os bits correspondentes ao RS

signal ReadReg2: std_logic_vector(0 to 4); -- Responsavel por receber os bits correspondentes ao RT

signal WriteReg: std_logic_vector(0 to 4); -- Responsavel por receber os bits correspondentes ao RD

signal WriteDataReg: std_logic_vector(0 to 31); -- Responsavel por receber os bits correspondentes a escrita,
se necessario, após toda a operação

signal DataRead1: std_logic_vector(0 to 31); -- Saida dos dados do Registrador 1, correspondete ao RS

signal DataRead2: std_logic_vector(0 to 31); -- Saida dos dados do Registrador 2, correspondete ao RT

signal imed: std_logic_vector(0 to 15); -- Sinal Imediato para as operações

signal imed_extended_ID: std_logic_vector(0 to 31); -- Imediato extendido para operação

signal RegRt_ID: std_logic_vector(0 to 4); -- Entrada do RT no registrador ID

signal RegRd_ID: std_logic_vector(0 to 4); -- Entrada do RD no registrador ID


--Sinais do Jump
signal jump_imediate4: std_logic_vector(0 to 31);

signal Concatenat_Jump: std_logic_vector(0 to 31);

signal InstJumpType:    std_logic;

-- ------------------------------------------------------------------------------------------------------------------------

signal verificaPC_0: std_logic_vector(0 to 31); -- Entrada0 do mux de PCSrc

signal verificaPC_1: std_logic_vector(0 to 31); -- Entrada1 do mux de PCSrc

signal SinalBranch: std_logic; -- Responsavel por indicar se a operação é um branch ou não

signal controle_WB_ID: std_logic_vector(0 to 1); -- Controle do WB para o Registrador de pipeline ID/EX

signal controle_ME_ID: std_logic_vector(0 to 2); -- Controle do MEM para o Registrador de pipeline ID/EX

signal controle_EX_ID: std_logic_vector(0 to 3); -- Controle do EX para o Registrador de pipeline ID/EX
```

```
signal imediate_extended_EX: std_logic_vector(0 to 31); -- Imediato extendido na etapa de execução
signal PC_plus4_EX: std_logic_vector(0 to 31); -- Pc mais 4 na etapa de execução
signal Imed_extend_4: std_logic_vector(0 to 31); -- Usado para o calculo do Branch
signal AddressBranch: std_logic_vector(0 to 31); -- Calculo do Branch com a ajuda do somador responsavel
signal SrcA_ULA: std_logic_vector(0 to 31); -- Entrada A na ULA - Vem do DataRead1
signal SrcB_ULA: std_logic_vector(0 to 31); -- Entrada B da ULA - Depende do resultado do ALUSrc
signal ResultadoULA: std_logic_vector(0 to 31); -- Resultado da ULA
signal Sinal_Zero: std_logic; -- Saida zero da ULA
signal ALUSrc: std_logic; -- Controle da entrada B na ULA - Controle do Multiplexador
signal ALUScr_0: std_logic_vector(0 to 31); -- Entrada 0 do multiplexador controlado pelo ALUSrc
signal ALUScr_1: std_logic_vector(0 to 31); -- Entrada 0 do multiplexador controlado pelo ALUSrc
signal ULA_Operation: std_logic_vector(0 to 1); -- Responsavel pelo controle de qual operação vai ser feita na ULA
signal RegDst: std_logic; -- Sinal de controle representando o RegDst
signal regDst_0: std_logic_vector(0 to 4); -- Entrada 0 do multiplexador controlado pelo RegDst
signal regDst_1: std_logic_vector(0 to 4); -- Entrada 1 do multiplexador controlado pelo RegDst
signal regDst_Saida: std_logic_vector(0 to 4); -- Saida do multiplexador controlado pelo RegDst
signal controle_WB_EX: std_logic_vector(0 to 1);  -- Sinal de controle do WB na etapa do registrador EX/MEM
signal controle_ME_EX: std_logic_vector(0 to 2); -- Sinal de controle do MEM na etapa do registrador EX/MEM
signal controle_EX_EX: std_logic_vector(0 to 3); -- Sinal de controle do EX na etapa do registrador EX/MEM

-------------------------------------------------------------------------------------------------------


signal endereco_MEM: std_logic_vector(0 to 31); -- Endereço de memoria onde o dado vai ser salvo

signal memWrite: std_logic; -- Sinal de controle representando o memWrite

signal MEM_writeData: std_logic_vector(0 to 31); -- Entrada da memoria de dados, vem da saida do DataRead2,
após passar por dois registradores(ID/EX e EX/MEM)

signal memRead:  std_logic;  -- Sinal de controle representando o memRead

signal MEM_readData: std_logic_vector(0 to 31); -- Saida da memoria de dados || Entrada do registrador MEM/WB

signal andBranch0: std_logic; -- Entrada 0 do AND responsavel por calcular se é um Branch ou não

signal andBranch1: std_logic; -- Entrada 1 do AND responsavel por calcular se é um Branch ou não || Saida Zero da ULA

signal regDst_MEM: std_logic_vector(0 to 4); -- Controle do multiplexador responsavel pela entrada de dados no registrador EX/MEM

signal controle_WB_ME: std_logic_vector(0 to 1); -- Sinal de controle do WB na etapa do registrador MEM/WB

signal controle_ME_ME: std_logic_vector(0 to 2); -- Sinal de controle do ME na etapa do registrador MEM/WB

-------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------
-- Sinais Writeback

signal MemToReg0: std_logic_vector(0 to 31); -- Entrada 0 do multiplexador responsavel pelo controle de Escrita no Registrador

signal MemToReg1: std_logic_vector(0 to 31); -- Entrada 1 do multiplexador responsavel pelo controle de Escrita no Registrador

signal memToReg: std_logic;  -- Sinal de controle representando o memToReg

signal controle_WB_WB: std_logic_vector(0 to 1); -- Sinal de controle do WB na etapa de saida do registrador MEM/WB


 -- Sinais para WaveForm

 pcAtual <= instrucaoPC;
 instrucAtual <= instrucao_IF_ID;

 -------------------------------------------------------------------------------------------------------

 -------------------------------------------------------------------------------------------------------
 -- Componentes Fetch Da Instrução
 Instruction_Memory: InstructionMemory port map (instrucaoPC, instrucao_IF_ID); -- OK

 PC4: AdderBranch port map (instrucaoPC, "00000000000000000000000000000100", PCSrc_0); -- OK

 MuxBranch: Muxs_32bits port map (PCSrc_0, PCSrc_1, SinalBranch, verificaPC_0); -- OK OK

 ProgramCounter: PC  port map (clock, atualizaPC, instrucaoPC); -- OK

 -------------------------------------------------------------------------------------------------------
```

```
  -- Registrador IF/ID

  Reg_IF_ID: IF_ID port map (clock, PCSrc_0, PC_plus4, instrucao_IF_ID, instrucao); -- OK

  ------------------------------------------------------------------------------------------------------------

  ------------------------------------------------------------------------------------------------------------
  -- Declaração das intruções de Decodificação
  OPCode <= instrucao(0 to 5);

  ReadReg1    <= instrucao(6 to 10);

  ReadReg2    <= instrucao(11 to 15);

  imed <= instrucao(16 to 31);

  RegRt_ID <= instrucao(11 to 15);

  RegRd_ID <= instrucao(16 to 20);

  Registradores: Registers port map (Sinal_regWrite, clock, ReadReg1, ReadReg2, WriteReg, WriteDataReg,
  DataRead1, DataRead2, deb_reg1, SaidaReg1, SaidaReg2, SaidaReg3,
  SaidaReg4, SaidaReg5, SaidaReg6, SaidaReg7); -- OK

  Sing_Extend: SignExtend port map (imed, imed_extended_ID); -- OK


MuxJumpType: Muxs_32bits port map (Concatenat_Jump, DataRead1, InstJumpType, verificaPC_1); -- MUX Jump

-- ------------------------------------------------------------------------------------------------------------

MuxPCSrc: Muxs_32bits port map (verificaPC_0, verificaPC_1, PCSrc, atualizaPC); -- OK

--debug_PCSrc <= atualizaPC;

-- ------------------------------------------------------------------------------------------------------------

ShiftLeftpJump: ShiftLeft2 port map (instrucao, jump_imediate4); -- Shift Left usado para a instrução jump

-- ------------------------------------------------------------------------------------------------------------


-- ------------------------------------------------------------------------------------------------------------

Concatenat_Jump <= PC_plus4(0 to 3) & jump_imediate4(4 to 31); -- Concatenação do PC+4 com o imediato necessario para o jump

-- ------------------------------------------------------------------------------------------------------------

ControlUnity: UnidadeControle port map (OPCode, PCSrc, InstJumpType, controle_WB_ID, controle_ME_ID, controle_EX_ID); -- OK


  -- Registrador ID/EX

  Reg_ID_EX: ID_EX port map (clock, controle_WB_ID, controle_ME_ID, controle_EX_ID, controle_WB_EX,
  controle_ME_EX, controle_EX_EX, PC_plus4, PC_plus4_EX, DataRead1, SrcA_ULA,DataRead2,
  ALUScr_0, imed_extended_ID, imediate_extended_EX,RegRt_ID, regDst_0, RegRd_ID, regDst_1); -- OK

  ------------------------------------------------------------------------------------------------------------

  ------------------------------------------------------------------------------------------------------------
  -- Componentes necessarios para a execução do programa

  CalculoBranch: AdderBranch  port map (PC_plus4_EX, Imed_extend_4, AddressBranch); -- OK

  --debug_AddBranch <= AddressBranch;

  ALU: ULA port map (SrcA_ULA, SrcB_ULA, ULA_Operation, ResultadoULA, Sinal_Zero); -- OK

  --debug_ULA0 <= SrcA_ULA;
  --debug_ULA1 <= SrcB_ULA;
  --debug_ULAOut <= ResultadoULA;
  --debug_Zero <= Sinal_Zero;

  MuxALUSrc: Muxs_32bits  port map (ALUScr_0, ALUScr_1, ALUSrc, SrcB_ULA); -- OK

  Mux_RegDst: MuxRegDst port map (regDst_0, regDst_1, RegDst, regDst_Saida); -- OK

  ShiftEX: ShiftLeft2 port map (imediate_extended_EX, Imed_extend_4); -- OK

  ALUScr_1 <= imediate_extended_EX;
```

```
process(controle_EX_EX)
begin
    ALUSrc <= controle_EX_EX(0);
    ULA_Operation <= controle_EX_EX(1 to 2);
    RegDst <= controle_EX_EX(3);
end process;

-------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------
-- Registrador EX/MEM

Reg_EX_MEM: EX_MEM port map (clock, controle_WB_EX, controle_ME_EX,controle_WB_ME,
controle_ME_ME,AddressBranch, PCSrc_1, Sinal_Zero, andBranch1,ResultadoULA,
endereco_MEM, ALUScr_0, MEM_writeData,regDst_Saida, regDst_MEM); -- OK

-------------------------------------------------------------------------------------------------


 -- Declaração dos componentes da Memoria de Dados

 Data_Memory: DataMemory port map (endereco_MEM, clock, memWrite, MEM_writeData,
 memRead, MEM_readData, SaidaMem1, SaidaMem2, SaidaMem3, SaidaMem4); -- OK

 memWrite <= controle_ME_ME(0);

 memRead <= controle_ME_ME(1);

 andBranch0 <= controle_ME_ME(2);

 --debug_branch <= andBranch0;

 SinalBranch <= andBranch0 and andBranch1;

 --debug_BranchSig <= SinalBranch;

 --debug_AddrDM <= endereco_MEM;
 --debug_WriteDataDM <= MEM_writeData;
 --debug_MemRead <= memRead;
 --debug_MemWrite <= memWrite;

  -------------------------------------------------------------------------------------------------
  -- Registrador MEM/WB
  Reg_MEM_WB: MEM_WB port map (clock, controle_WB_ME, controle_WB_WB,
  MEM_readData, MemToReg1,endereco_MEM, MemToReg0, regDst_MEM, WriteReg); -- OK

  -------------------------------------------------------------------------------------------------

  -------------------------------------------------------------------------------------------------
  -- Declaração do funcionamento do WriteBack

  MuxMEMtoReg: MUXs_32bits port map (MemToReg0, MemToReg1, memToReg, WriteDataReg); -- OK
  --debug_MemtoReg0 <= MemToReg0;
  --debug_MemtoReg1 <= MemToReg1;
  --debug_MemtoRegSig <= memToReg;

  memToReg <= controle_WB_WB(0);
  Sinal_regWrite <= controle_WB_WB(1);

end;
```

DataMemory

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity DataMemory is

    port(
        -- Entradas da memória
        endereco: in std_logic_vector(0 to 31);
        clock: in std_logic;
        memWrite: in std_logic;
        writeData: in std_logic_vector(0 to 31);
        memRead: in std_logic;

        -- Saída da memória
        readData: out std_logic_vector(0 to 31);

        mem1: out std_logic_vector(0 to 31);
        mem2: out std_logic_vector(0 to 31);
        mem3: out std_logic_vector(0 to 31);
        mem4: out std_logic_vector(0 to 31)
        );

end DataMemory;

architecture Data of DataMemory is
    type memoria is array (0 to 100) of std_logic_vector(0 to 7);
    signal memory: memoria;
begin

    mem1 <= memory(0) & memory(1) & memory(2) & memory(3);
    mem2 <= memory(4) & memory(5) & memory(6) & memory(7);
    mem3 <= memory(8) & memory(9) & memory(10) & memory(11);
    mem4 <= memory(12) & memory(13) & memory(14) & memory(15);
```

```vhdl
    process(clock)
    begin
        if (clock'event and clock = '1') then
            -- Se MemWrite estiver ativo
            if (memWrite = '1') then
                memory(to_integer(unsigned(endereco))) <= writeData(0 to 7);
                memory(to_integer(unsigned(endereco)) +1) <= writeData(8 to 15);
                memory(to_integer(unsigned(endereco)) +2) <= writeData(16 to 23);
                memory(to_integer(unsigned(endereco)) +3) <= writeData(24 to 31);
            end if;

            -- Se MemRead estiver ativo
            if (memRead = '1') then
            readData <= memory(to_integer(unsigned(endereco)))&
                        memory(to_integer(unsigned(endereco)) +1)&
                        memory(to_integer(unsigned(endereco)) +2)&
                        memory(to_integer(unsigned(endereco)) +3);
            else
                readData <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
            end if;
        end if;
    end process;
end;
```

EX_MEM

```
entity EX_MEM is

    port (clock: in std_logic;

            -- Declaração da Entrada e Saida do WB e ME
            entradaWB: in std_logic_vector(0 to 1);
            entradaMEM: in std_logic_vector(0 to 2);
            saidaWB: out std_logic_vector(0 to 1) := "00";
            saidaMEM: out std_logic_vector(0 to 2) := "000";

            -- Entrada e Saido do PC, responavel pela instrução/proxima instrução
            entradaPC: in std_logic_vector(0 to 31);
            saidaPC: out std_logic_vector(0 to 31) := "00000000000000000000000000000000";

            -- Entrada e Saido do sinal ZERO da ULA
            entradaZERO: in std_logic;
            saidaZERO: out std_logic := '0';

            -- Entrada e saido do resultado da operação da ALU/ULA
            entradaResultado: in std_logic_vector(0 to 31);
            saidaResultado: out std_logic_vector(0 to 31) := "00000000000000000000000000000000";

            -- Entrada e Saida dos Dados que estão no Write Register
            entrada_DadoWriteRegister: in std_logic_vector(0 to 31);
            saida_DadoWriteRegister: out std_logic_vector(0 to 31) := "00000000000000000000000000000000";

            entrada_RegDST: in std_logic_vector(0 to 4);
            saida_RegDST: out std_logic_vector(0 to 4) := "00000"
            );

end EX_MEM;

architecture EXMEM of EX_MEM is

begin
    process(clock)
    -- Permite que a saida recebe os dados de entrada do Registrador. Isso só ocorre na subida do clock.
    begin
        if (clock'event and clock = '1') then
            saidaWB <= entradaWB;
            saidaMEM <= entradaMEM;
            saidaPC <= entradaPC;
            saidaZERO <= entradaZERO;
            saidaResultado  <= entradaResultado;
            saida_DadoWriteRegister <= entrada_DadoWriteRegister;
            saida_RegDST <= entrada_RegDST;
        end if;
    end process;
end;
```

ID_EX

```vhdl
-- Declaração de Variaveis
entity ID_EX is
    port (
            clock: in std_logic;

            -- Declaração das entradas e saidas do registrador de Pipeline
            entradaWB: in std_logic_vector(0 to 1);
            entradaMEM: in std_logic_vector(0 to 2);
            entradaEX: in std_logic_vector(0 to 3);
            saidaWB: out std_logic_vector(0 to 1) := "00";
            saidaMEM: out std_logic_vector(0 to 2) := "000";
            saidaEX: out std_logic_vector(0 to 3) := "0000";

            -- Entrada e saida do valor do PC, responsavel pela instrução
            entradaPC: in std_logic_vector(0 to 31);
            saidaPC: out std_logic_vector(0 to 31) := "00000000000000000000000000000000";

            -- Declaração da Entrada e Saida do dado 1
            entrada_ReadData1: in std_logic_vector(0 to 31);
            saida_ReadData1: out std_logic_vector(0 to 31) := "00000000000000000000000000000000";

            -- Declaração da Entrada e Saida do dado 2
            entrada_ReadData2: in std_logic_vector(0 to 31);
            saida_ReadData2: out std_logic_vector(0 to 31) := "00000000000000000000000000000000";

            -- Entrada e Saida do imediato
            entrada_imed: in std_logic_vector(0 to 31);
            saida_imed: out std_logic_vector(0 to 31) := "00000000000000000000000000000000";

            -- Entrada e Saida dos Registradores RT e RD, cada um com 5 bits
            entradaRT: in std_logic_vector(0 to 4);
            saidaRT: out std_logic_vector(0 to 4) := "00000";
            entradaRD: in std_logic_vector(0 to 4);
            saidaRD: out std_logic_vector(0 to 4) := "00000");
end ID_EX;
```

```vhdl
architecture IDEX of ID_EX is

begin
    process(clock)
    begin
        -- Define que a saida recebe a entrada na subida do clock
        if (clock'event and clock = '1') then
            saidaWB <= entradaWB;
            saidaMEM <= entradaMEM;
            saidaEX <= entradaEX;
            saidaPC <= entradaPC;
            saida_ReadData1 <= entrada_ReadData1;
            saida_ReadData2 <= entrada_ReadData2;
            saida_imed <= entrada_imed;
            saidaRT <= entradaRT;
            saidaRD <= entradaRD;
        end if;
    end process;
end;
```

IF_ID

```
entity IF_ID is
    port(
        clock: in   std_logic;

        -- 32 bits para o PC
        pcIn: in std_logic_vector(0 to 31);
        pcOut: out std_logic_vector(0 to 31) := "00000000000000000000000000000000";

        -- 32 bits para a instrução
        InstructionIn:  in std_logic_vector(0 to 31);
        InstructionOut: out std_logic_vector(0 to 31) := "00000000000000000000000000000000"
        );

end IF_ID;

architecture IFID of IF_ID is
begin
    process(clock)
        begin
            if (clock'event and clock = '1') then
                InstructionOut <= InstructionIn;
                pcOut <= pcIn;
            end if;
    end process;
end;
```

InstructionMemory

```
entity InstructionMemory is

    port(
        -- Endereço que chega na memória de instrução
        endereco: in std_logic_vector(0 to 31);

        -- Saída da memória de instrução
        instrucao: out std_logic_vector(0 to 31) := "00000000000000000000000000000000"

        );

end InstructionMemory;
```

```
architecture Instructions of InstructionMemory is

    type instrucoes is array (0 to 384) of std_logic_vector(0 to 7);
    signal inst: instrucoes;

begin
        --Instruções tipo R
        --inst(x) <= "aaaaaabb"
        --inst(x) <= "bbbccccc"
        --inst(x) <= "dddddeee"
        --inst(x) <= "eeffffff"
        -- a: opcode
        -- b: Rs
        -- c: Rt
        -- d: Rd
        -- e: shamt
        -- f: funct

        --Instrucoes tipo I
        --inst(x) <= "aaaaaabb"
        --inst(x) <= "bbbccccc"
        --inst(x) <= "dddddddd"
        --inst(x) <= "dddddddd"
        -- a: opcode
        -- b: Rs
        -- c: Rt
        -- d: Imediato ou endereço

        -- Addi: Reg1, 0, 7
        inst(0) <= "00001100";
        inst(1) <= "00000001";
        inst(2) <= "00000000";
        inst(3) <= "00000111";
```

```
--Addi: Reg2, 0, 5
inst(4) <= "00001100";
inst(5) <= "00000010";
inst(6) <= "00000000";
inst(7) <= "00000101";

--Addi: Reg3, 0, 3
inst(8) <= "00001100";
inst(9) <= "00000011";
inst(10) <= "00000000";
inst(11) <= "00000011";

-- nop
inst(12) <= "00000000";
inst(13) <= "00000000";
inst(14) <= "00000000";
inst(15) <= "00000000";

-- nop
inst(16) <= "00000000";
inst(17) <= "00000000";
inst(18) <= "00000000";
inst(19) <= "00000000";

-- nop
inst(20) <= "00000000";
inst(21) <= "00000000";
inst(22) <= "00000000";
inst(23) <= "00000000";

-- Add: Reg1, Reg2, Reg3
inst(24) <= "00000100";
inst(25) <= "01000011";
inst(26) <= "00001000";
inst(27) <= "00000000";
```

```
-- SUB Reg1, Reg2, Reg3; reg2-reg3
inst(28) <= "00001000";
inst(29) <= "01000011";
inst(30) <= "00001000";
inst(31) <= "00000000";

-- AND Reg1, Reg2, Reg3;
inst(32) <= "00011100";
inst(33) <= "01000011";
inst(34) <= "00001000";
inst(35) <= "00000000";

-- OR Reg1, Reg2, Reg3;
inst(36) <= "00100100";
inst(37) <= "01000011";
inst(38) <= "00001000";
inst(39) <= "00000000";

-- SUBi Reg1, Reg2, 3
inst(40) <= "00010000";
inst(41) <= "01000001";
inst(42) <= "00000000";
inst(43) <= "00000011";

-- ANDi Reg1, Reg2, 7
inst(44) <= "00100000";
inst(45) <= "01000001";
inst(46) <= "00000000";
inst(47) <= "00000111";

-- ORi Reg1, Reg2, 7
inst(48) <= "00101000";
inst(49) <= "01000001";
inst(50) <= "00000000";
inst(51) <= "00000111";

--Sw Reg2, 0(Reg7)
inst(52) <= "00011000";
inst(53) <= "11100010";
inst(54) <= "00000000";
inst(55) <= "00000000";
```

```
-- nop
inst(56) <= "00000000";
inst(57) <= "00000000";
inst(58) <= "00000000";
inst(59) <= "00000000";

-- nop
inst(60) <= "00000000";
inst(61) <= "00000000";
inst(62) <= "00000000";
inst(63) <= "00000000";

-- nop
inst(64) <= "00000000";
inst(65) <= "00000000";
inst(66) <= "00000000";
inst(67) <= "00000000";

--LW Reg5, 0(Reg7)
inst(68) <= "00010100";
inst(69) <= "11100101";
inst(70) <= "00000000";
inst(71) <= "00000000";

--ADDI REG1,0,0
inst(72) <= "00001100";
inst(73) <= "00000001";
inst(74) <= "00000000";
inst(75) <= "00000000";

-- nop
inst(76) <= "00000000";
inst(77) <= "00000000";
inst(78) <= "00000000";
inst(79) <= "00000000";

-- nop
inst(80) <= "00000000";
inst(81) <= "00000000";
inst(82) <= "00000000";
inst(83) <= "00000000";

-- BEQ REG1, Reg0, 27
inst(84) <= "00101100";
inst(85) <= "00100000";
inst(86) <= "00000000";
inst(87) <= "00011011";
```

```
-- NOP
inst(88) <= "00000000";
inst(89) <= "00000000";
inst(90) <= "00000000";
inst(91) <= "00000000";

-- NOP
inst(92) <= "00000000";
inst(93) <= "00000000";
inst(94) <= "00000000";
inst(95) <= "00000000";

-- NOP
inst(96) <= "00000000";
inst(97) <= "00000000";
inst(98) <= "00000000";
inst(99) <= "00000000";

-- NOP
inst(100) <= "00000000";
inst(101) <= "00000000";
inst(102) <= "00000000";
inst(103) <= "00000000";

-- Add Reg1, Reg2, Reg3
inst(104) <= "00000100";
inst(105) <= "01000011";
inst(106) <= "00001000";
inst(107) <= "00000000";

-- J 30
inst(108) <= "00110000";
inst(109) <= "00000000";
inst(110) <= "00000000";
inst(111) <= "00011110";

-- NOP
inst(112) <= "00000000";
inst(113) <= "00000000";
inst(114) <= "00000000";
inst(115) <= "00000000";

-- NOP
inst(116) <= "00000000";
inst(117) <= "00000000";
inst(118) <= "00000000";
inst(119) <= "00000000";
```

```
            -- ADDI Reg1, Reg0, 4
            inst(120) <= "00001100";
            inst(121) <= "00000001";
            inst(122) <= "00000000";
            inst(123) <= "00000100";

            -- ADDI Reg2, Reg0, 8
            inst (125) <= "00001100";
            inst (126) <= "00001100";
            inst (127) <= "00001100";
            inst (128) <= "00001100";


    process(endereco)
        begin
            -- Saída recebe entrada
            instrucao <= inst(to_integer(unsigned(endereco))) &
            inst(to_integer(unsigned(endereco)) + 1) &
            inst(to_integer(unsigned(endereco)) + 2) &
            inst(to_integer(unsigned(endereco)) + 3);
    end process;
end;
```

MEM_WB

```
entity MEM_WB is
    port(
            clock: in   std_logic;

            -- 2 bits para o sinal de controle WB
            wbIn: in std_logic_vector(0 to 1);
            wbOut: out  std_logic_vector(0 to 1) := "00";

            -- 32 bits para o  readData
            readDataIn: in std_logic_vector(0 to 31);
            readDataOut: out    std_logic_vector(0 to 31) := "00000000000000000000000000000000";

            -- 32 bits para o  endereço
            addrIn: in  std_logic_vector(0 to 31);
            addrOut: out std_logic_vector(0 to 31) := "00000000000000000000000000000000";

            -- 5 bits para o  RegDst
            regDstIn: in std_logic_vector(0 to 4);
            regDstOut: out  std_logic_vector(0 to 4) := "00000"
        );

end MEM_WB;

architecture MEMWB of MEM_WB is

begin
    process(clock)
    begin
        if (clock'event and clock = '1') then
            wbOut <= wbIn;
            readDataOut <= readDataIn;
            addrOut <= addrIn;
            regDstOut <= regDstIn;
        end if;
    end process;
end;
```

MuxRegDst

```
entity MuxRegDst is

    port(

        -- Registrador rt
        regRt: in std_logic_vector(0 to 4);

        -- Registrador rd
        regRd: in std_logic_vector(0 to 4);

        -- Sinal RegDst
        regDst: in std_logic;

        -- Saída do mux
        saidaMux: out std_logic_vector(0 to 4)
        );

end MuxRegDst;

architecture muxReg of MuxRegDst is
begin
    process(regDst, regRt, regRd)
        begin
            if (regDst = '0') then
                -- Se o sinal for 0
                saidaMux <= regRt;
            else
                -- Se o sinal for 1
                saidaMux <= regRd;
            end if;
    end process;
end;
```

Muxs_32bits

```vhdl
entity Muxs_32bits is

    port(

        -- 1° entrada do mux
        entrada1: in  std_logic_vector(0 to 31);

        -- 2° entrada do registrador
        entrada2: in  std_logic_vector(0 to 31);

        -- Sinal de cotrole do mux
        sinalControle: in  std_logic;

        -- Saída do mux
        saidaMux32: out std_logic_vector(0 to 31)
        );

end Muxs_32bits;

architecture mux32 of Muxs_32bits is
begin
    process(sinalControle, entrada1, entrada2)
    begin
        if (sinalControle = '0') then
            -- Se o sinal for 0
            saidaMux32 <= entrada1;
        else
            -- Se o sinal for 1
            saidaMux32 <= entrada2;
        end if;
    end process;
end;
```

PC

```vhdl
entity PC is
    port(
        clock:  in  std_logic;

        -- Declaração de pc e pc+4
        pc4:in std_logic_vector(0 to 31);
        pc: out std_logic_vector(0 to 31) := "00000000000000000000000000000000"
        );

end PC;

architecture PC_4 of PC is
begin
    process (clock, pc4)
        begin
            if (clock'event and clock = '1') then
                -- Realiza PC+4
                pc <= pc4;
            end if;
    end process;
end;
```

Registers

```
entity Registers is

    port(
        -- Clock e sinal de RegWrite
        regWrite: in std_logic;
        clock: in std_logic;

        -- Entradas dos registradores que serão lidos
        readRegister1: in std_logic_vector(0 to 4);
        readRegister2: in std_logic_vector(0 to 4);

        -- Registrador que será escrito
        writeRegister: in std_logic_vector(0 to 4);

        -- Dado de escrita
        writeData: in std_logic_vector(0 to 31);

        -- Saída dos banco de registradores
        readData1: out std_logic_vector(0 to 31);
        readData2: out std_logic_vector(0 to 31);

        register1: out std_logic_vector(0 to 31);
        register2: out std_logic_vector(0 to 31);
        register3: out std_logic_vector(0 to 31);
        register4: out std_logic_vector(0 to 31);
        register5: out std_logic_vector(0 to 31);
        register6: out std_logic_vector(0 to 31);
        register7: out std_logic_vector(0 to 31);
        register8: out std_logic_vector(0 to 31)

        );

end Registers;

architecture registrador of Registers is

    -- 10 registradores de 32 bits
    type RegArray is array (0 to 9) of std_logic_vector(0 to 31);
    signal Testereg: RegArray;

    begin

        register1 <= Testereg(0);
        register2 <= Testereg(1);
        register3 <= Testereg(2);
        register4 <= Testereg(3);
        register5 <= Testereg(4);
        register6 <= Testereg(5);
        register7 <= Testereg(6);
        register8 <= Testereg(7);

        process(clock)
        begin
        -- Para conteúdo de writeData poder ser escrito em um registrador
        if (clock'event and clock = '1' and regWrite = '1' and not (writeRegister = "00000") ) then
            Testereg(to_integer(unsigned(writeRegister))) <= writeData;
        end if;
    end process;

    readData1 <= Testereg(to_integer(unsigned(readRegister1)));
    readData2 <= Testereg(to_integer(unsigned(readRegister2)));

end;
```

ShiftLeft2

```
entity ShiftLeft2 is
    -- Declaração das variaveis
    port(
        entradaSHL: in  std_logic_vector(0 to 31);
        saidaSHL:   out std_logic_vector(0 to 31)
        );
end ShiftLeft2;

architecture sll2 of ShiftLeft2 is
begin
    -- Necessario colocar 2 bits para manter o formato da instrução com 32 bits
    saidaSHL <= entradaSHL(2 to 31) & "00";  -- EX: 0101010.... → 00 0101010....
end;
```

SignExtend

```
entity SignExtend is
    -- Declaração de Variaveis
    port(
        entradaSE:  in  std_logic_vector(0 to 15);
        saidaSE:    out std_logic_vector(0 to 31)
        );
end SignExtend;

architecture SignExt of SignExtend is
begin
    process(entradaSE)
    begin
        if (entradaSE(0) = '0') then -- Se o bit inicial for 0, preenche os outros 16 bits com 0
            saidaSE <= "0000000000000000" & entradaSE;
        else -- Caso contrario o bit será 1, e os 16 bits extras serão 1
            saidaSE <= "1111111111111111" & entradaSE;
        end if;
    end process;
end;
```

ULA

```
entity ULA is

    port (
        -- Primeiro registrador que entra na ULA
        aluSrcA: in std_logic_vector(0 to 31);

        -- Segundo registrador que entra na ULA
        aluSrcB: in std_logic_vector(0 to 31);

        -- Operação feita pela ULA (add, sub, and ou or)
        aluOp:  in std_logic_vector(0 to 1);

        -- Saída da ULA
        aluResult: out std_logic_vector(0 to 31);

        -- Sinal de controle zero
        zero: out std_logic);

end ULA;
```

```
architecture UnidadeLogArit of ULA is
     signal conta: std_logic_vector(0 to 31);
begin
    process(aluSrcA, aluSrcB, aluOp)
    begin
        case aluOp is

            when "00" => conta <= aluSrcA + aluSrcB;
            when "01" => conta <= aluSrcA - aluSrcB;
            when "10" => conta <= aluSrcA and aluSrcB;
            when "11" => conta <= aluSrcA or aluSrcB;
            when others => conta <= "00000000000000000000000000000000";

        end case;

        if(conta = "00000000000000000000000000000000") then
            zero <= '1';
        else
            zero <= '0';
        end if;

         aluResult <= conta;
    end process;
end;
```

UnidadeControle

```
entity UnidadeControle is

    port (
        -- Opcode
        opcode: in  std_logic_vector(0 to 5);

        -- Sinal de PCSrc
        PCSrc: out std_logic := '0';

        jumpSignal: out std_logic := '0';

        -- Sinal de controle para o estágio de WB
        SignalWB: out std_logic_vector(0 to 1) := "00";

        -- Sinal de controle para o estágio de MEM
        SignalMEM: out std_logic_vector(0 to 2) := "000";

        -- Sinal de controle para o estáio de EX
        SignalEX: out std_logic_vector(0 to 3) := "0000"

        );

end UnidadeControle;
```

```
architecture control of UnidadeControle is
begin
    process(opcode)
    begin
        case opcode is

            -- NOP
            when "000000" =>
                PCSrc <= '0';
                SignalEX <= "XXXX";
                SignalMEM <= "0X0";
                SignalWB <= "X0";


            -- ADD
            when "000001" =>
                PCSrc <= '0';
                SignalEX <= "0001";
                SignalMEM <= "0X0";
                SignalWB <= "01";


            -- SUB
            when "000010" =>
                PCSrc <= '0';
                SignalEX <= "0011";
                SignalMEM <= "0X0";
                SignalWB <= "01";


            -- ADDI
            when "000011" =>
                PCSrc <= '0';
                SignalEX <= "1000";
                SignalMEM <= "0X0";
                SignalWB <= "01";
```

```
-- SUBI
when "000100" =>
    PCSrc <= '0';
    SignalEX <= "1010";
    SignalMEM <= "0X0";
    SignalWB <= "01";


-- LW
when "000101" =>
    PCSrc <= '0';
    SignalEX <= "1000";
    SignalMEM <= "010";
    SignalWB <= "11";


-- SW
when "000110" =>
    PCSrc <= '0';
    SignalEX <= "100X";
    SignalMEM <= "1X0";
    SignalWB <= "00";


-- AND
when "000111" =>
    PCSrc <= '0';
    SignalEX <= "0101";
    SignalMEM <= "0X0";
    SignalWB <= "01";


-- ANDI
when "001000" =>
    PCSrc <= '0';
    SignalEX <= "1100";
    SignalMEM <= "0X0";
    SignalWB <= "01";
```

```
-- OR
when "001001" =>
    pcsrc <= '0';
    SignalEX <= "0111";
    SignalMEM <= "0X0";
    SignalWB <= "01";


-- ORI
when "001010" =>
    PCSrc <= '0';
    SignalEX <= "1110";
    SignalMEM <= "0X0";
    SignalWB <= "01";


-- BEQ  verificar se PCSrc nao deve ser 1
when "001011" =>
    PCSrc <= '0';
    SignalEX <= "001X";
    SignalMEM <= "0X1";
    SignalWB <= "00";


-- Jump
when "001100" =>
    PCSrc           <= '1';
    jumpSignal  <= '0';
    SignalEX                <= "XXXX";
    SignalMEM           <= "0XX";
    SignalWB                <= "00";


-- Jump Register
when "001101" =>
    pcsrc           <= '1';
    jumpSignal  <= '1';
    SignalEX        <= "XXXX";
    SignalMEM   <= "0XX";
    SignalWB        <= "00";
```

```
            -- OTHERS
            when others =>
                PCSrc <= '0';
                SignalEX <= "XXXX";
                SignalMEM <= "0X0";
                SignalWB <= "00";

        end case;
    end process;
end;
```