Universidade Federal da Bahia
Escola Politécnica
Departamento de Engenharia Elétrica e de Computação
Docente: Wagner Oliveira
Discentes: Caio Mendes, Fábio Miguel, Gerson Daniel, Heverton Reis e José Alves
Semestre 2024.2

Processador MIPS 32 bits - Documento de Arquitetura

1. Módulo TOP

O *processador* é o componente principal de um sistema de computação. É responsável por executar instruções, processar dados e coordenar o funcionamento de outros componentes. Ele interpreta e realiza operações básicas, como cálculos aritméticos, operações lógicas e movimentação de dados, por meio de uma sequência de instruções

Neste projeto, a *Unidade Central de Processamento* (CPU) será o módulo principal que integra e coordena todos os componentes do processador. Este módulo é responsável por orquestrar o fluxo de dados entre os demais módulos funcionais. No projeto, abarca todos os módulos principais, como a *Unidade de Controle* (CUnit), *Unidade Lógica e Aritmética* (ALU) e *Banco de Registradores*, para realizar operações e movimentar dados.

O módulo TOP será o nosso processador pipeline. As memórias serão módulos *IP's* do software *Quartus*. Os módulos que estarão dentro do nosso processador serão os estágios da nossa pipeline (IF_Stage, ID_Stage, EX_Stage, MEM_Stage e WB_Stage). As entradas do nosso processador são dois clocks com frequências distintas (clk e clk_rom) e um sinal de reset.

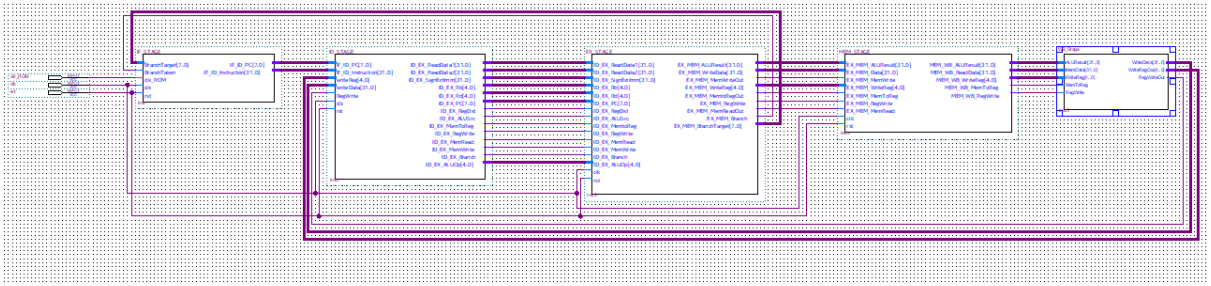

Figura 1- Diagrama de blocos do circuito.

2. Layout de instruções

Seguindo as especificações, o processador deve contar com trinta e dois registradores de uso geral. Logo, para endereçar estes registradores, é necessário o uso de até cinco bits como se segue:

REGISTRADORES	CODE
R0	00000
R1	00001
R2	00010
R3	00011
R4	00100
R5	00101
R6	00110
R7	00111
R8	01000
R9	01001
R10	01010
R11	01011
R12	01100
R13	01101
R14	01110
R15	01111
R16	10000
R17	10001
R18	10010
R19	10011
R20	10100
R21	10101
R22	10110
R23	10111
R24	11000
R25	11001
R26	11010
R27	11011
R28	11100
R29	11101
R30	11110
R31	11111

Figura 2 - Código dos registradores.

A palavra terá uma largura de dados de trinta e dois bits, conforme especificado, com dezenove operações previstas. Assim, define-se que cada *Código de Operação* (OPCode) deverá contar com um mínimo de cinco bits. Segue a lista de OPCodes previstos:

TIPO	INTRUÇÃO	OPCODE
TRANSFERENCIA DE DADOS	LW_1	00000
	LW_2	00001
	LW_3	00010
	SW_1	00011
	SW_2	00100
	MOV	00101
ARITMETICA	ADD	00110
	SUB	00111
	MUL	01000
	DIV	01001
LÓGICA	AND	01010
	OR	01011
	SHL	01100
	SHR	01101
	CMP	01110
	NOT	01111
TRANSFERENCIA DE CONTROLE	JR	10000
	JPC	10001
	BRFL	10010
	CALL	10011
	RET	10100
	NOP	10101

Figura 3 - Opcode de instruções.

Para instruções de *Transferência de Dados*:

INSTRUÇÃO	ENDEREÇAMENTO
LW_1	Deslocamento
LW_2	Direto
LW_3	Imediato
SW_1	Deslocamento
SW_2	Direto

Figura 4 - formato de instrução de transferência de dados.

***Obs:** optou-se por diferenciar qual o tipo de endereçamento das instruções LW e SW a partir de seu OPCode.

TRANSFERÊNCIA DE DADOS		
OPERAÇÃO	LW_1	Leitura de dados
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador de base
RD	5 bits	Registrador de destino
COMPLETAR COM 0		
IM	16 bits	Valor imediato
OPERAÇÃO	LW_2	Leitura de dados
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador de base
RD	5 bits	Registrador de destino
COMPLETAR COM 0		
OPERAÇÃO	LW_3	Leitura de dados
OPCODE	5 bits	Código da operação
RD	5 bits	Registrador de destino
COMPLETAR COM 0		
IM	16 bits	Valor imediato
OPERAÇÃO	SW_1	Escrita de dados
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador de base
RD	5 bits	Registrador de destino
COMPLETAR COM 0		
IM	16 bits	Valor imediato
OPERAÇÃO	SW_2	Escrita de dados
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador Base
RD	5 bits	Registrador de Destino
COMPLETAR COM 0		
OPERAÇÃO	MOV	Escrita de dados
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador Base
RD	5 bits	Registrador de Destino
COMPLETAR COM 0		

Figura 5 - Formato de instrução de transferência de dados.

ARITMÉTICA		
OPERAÇÃO	**ADD**	Soma de valores
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador de base
RD	5 bits	Registrador de destino
COMPLETAR COM 0		
OPERAÇÃO	**SUB**	Subtração de valores
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador de base
RD	5 bits	Registrador de destino
COMPLETAR COM 0		
OPERAÇÃO	**MUL**	Leitura de dados
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador de base
RD	5 bits	Registrador de destino
COMPLETAR COM 0		
OPERAÇÃO	**DIV**	Leitura de dados
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador de base
RD	5 bits	Registrador de destino
COMPLETAR COM 0		

Figura 5 - Formato de instrução aritmética.

LÓGICA		
OPERAÇÃO	AND	AND lógico
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador de base
RD	5 bits	Registrador de destino
COMPLETAR COM 0		
OPERAÇÃO	OR	Leitura de dados
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador de base
RD	5 bits	Registrador de destino
COMPLETAR COM 0		
OPERAÇÃO	SHL	Shift para esquerda
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador de base
RD	5 bits	Registrador de destino
COMPLETAR COM 0		
OPERAÇÃO	SHR	Shift para direita
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador de base
RD	5 bits	Registrador de destino
COMPLETAR COM 0		
OPERAÇÃO	CMP	Comparador lógico
OPCODE	5 bits	Código da operação
RB	5 bits	Registrador de destino
RD	5 bits	Registrador de dados
COMPLETAR COM 0		
OPERAÇÃO	NOT	Negação lógica
OPCODE	5 bits	Código da operação
RD	5 bits	Registrador de destino
COMPLETAR COM 0		

Figura 6 - Formato de instrução lógica.

Para instruções de Transferência de controle:

TRANSFERÊNCIA DE CONTROLE		
OPERAÇÃO	**JR**	Leitura de dados
OPCODE	5 bits	Código da operação
RD	5 bits	Registrador de destino
COMPLETAR COM 0		
OPERAÇÃO	**JPC**	Escrita de dados
OPCODE	5 bits	Código da operação
COMPLETAR COM 0		
IM	16 bits	Valor imediato
OPERAÇÃO	**BRFL**	Leitura de dados
OPCODE	5 bits	Código da operação
RD	5 bits	Registrador de destino
IM	5 bits	Vetor comparado com RFlags
M	5 bits	Máscara RFlags
COMPLETAR COM 0		
OPERAÇÃO	**CALL**	Leitura de dados
OPCODE	5 bits	Código da operação
RD	5 bits	Registrador de destino
COMPLETAR COM 0		
OPERAÇÃO	**RET**	Leitura de dados
OPCODE	5 bits	Código da operação
COMPLETAR COM 0		
OPERAÇÃO	**NOP**	Leitura de dados
OPCODE	5 bits	Código da operação
COMPLETAR COM 0		

Figura 5 - Formato de transferência de controle

3. Estágios do pipeline

O pipeline no processador divide a execução das instruções em diferentes estágios, permitindo que várias instruções sejam processadas ao mesmo tempo, cada uma em uma etapa específica. No caso de um pipeline de cinco estágios, cada instrução passa pelas seguintes fases:

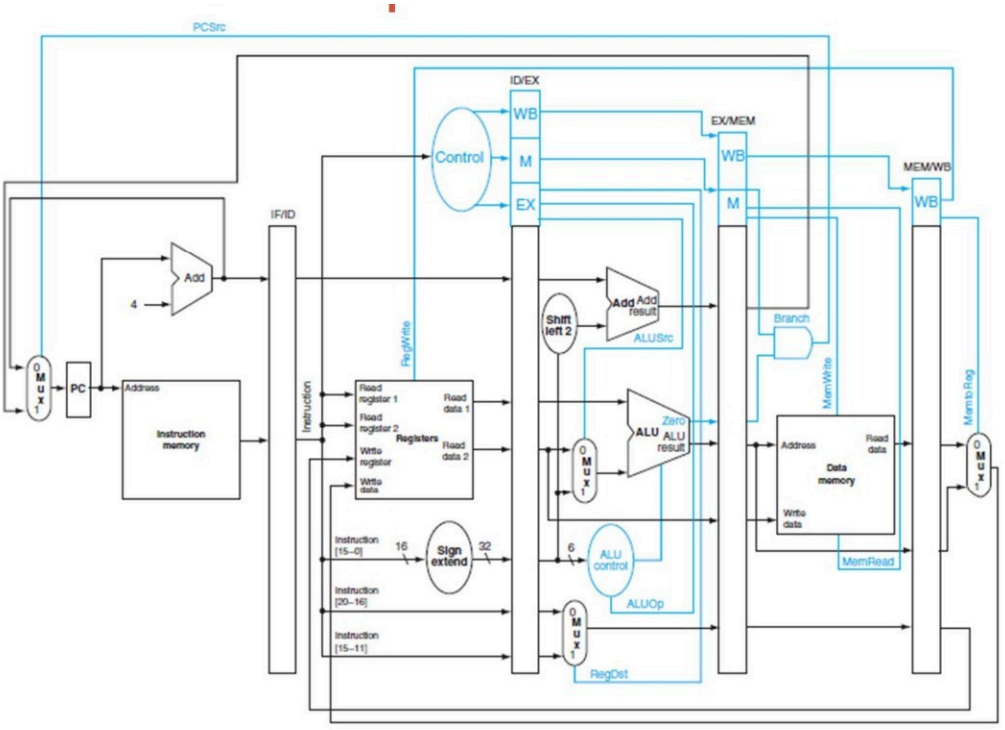

Figura 7 - Estágio pipeline

IF (Instruction Fetch): O processador busca a próxima instrução na memória de instruções. Os módulos funcionais e registradores internos serão:

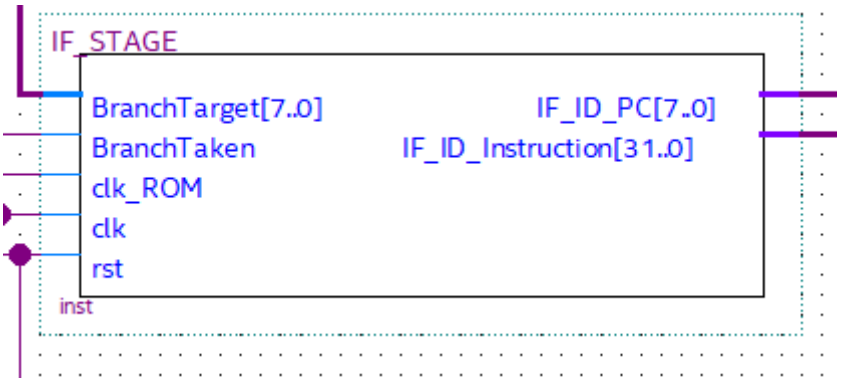

Figura 8 - Módulo IF Stage

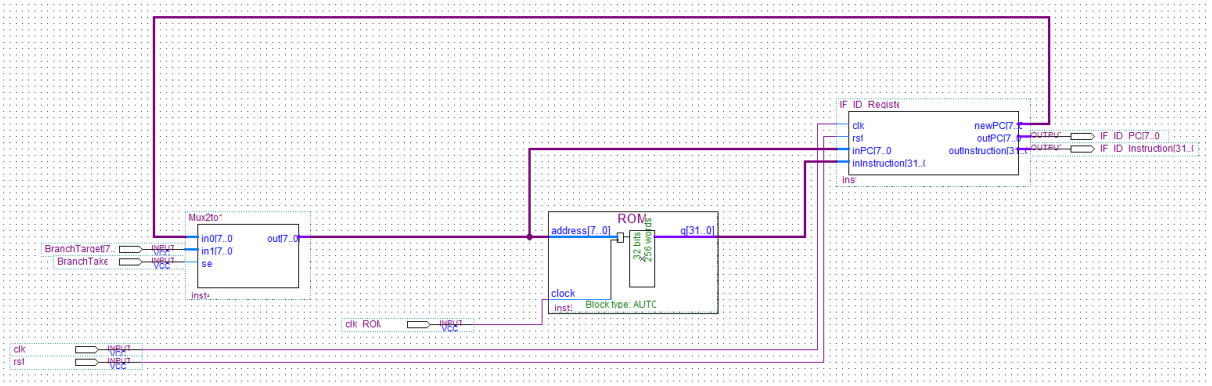

Figura 9 - Dentro do módulo IF Stage

Sinais de controle:

- **BranchTaken**: Sinaliza se o PC (Program Counter) será atualizado ou não com o valor calculado do Branch, permitindo que o processador direcione o fluxo de execução para um novo endereço em caso de desvio.

- **BranchTarget**: Define o próximo valor do PC, especialmente em operações com desvio, garantindo que o processador continue a execução a partir do endereço correto quando uma instrução de desvio é detectada.

- **clk**: Sinal de clock que sincroniza as operações no estágio IF, garantindo que a busca e a preparação das instruções ocorram no momento adequado.

- **clk_ROM**: Sinal de clock específico para o acesso à memória ROM, onde as instruções estão armazenadas, assegurando que a leitura das instruções seja realizada de forma sincronizada e confiável.

- **IF_ID_Instruction**: Representa a instrução que foi buscada da memória e é passada para o estágio ID (Instruction Decode). Esse sinal carrega a instrução que será decodificada e preparada para execução nos estágios subsequentes.

- **IF_ID_PC**: Representa o valor atual do PC (Program Counter) no momento em que a instrução foi buscada. Esse valor é passado para o estágio ID, permitindo que o processador mantenha o controle do fluxo de execução e saiba de qual endereço a instrução foi obtida.

Módulos do estágio:

- **Mux2to1:** (Multiplexador 2 para 1): Este módulo seleciona entre dois valores possíveis para o próximo endereço do PC. Ele decide se o PC será atualizado com o valor incrementado pelo PC Adder (fluxo sequencial) ou com o valor calculado pelo BranchTarget (em caso de desvio). A seleção é controlada pelo sinal BranchTaken.

- **IF_ID_Register:** Este registrador temporário armazena duas informações críticas que são passadas para o estágio ID (Instruction Decode): o valor atual do PC (IF_ID_PC) e a instrução que foi buscada da memória (IF_ID_Instruction). Ele garante que essas informações sejam mantidas e transferidas de forma sincronizada para o próximo estágio.

- **ROM (Memória de Instruções):** Este módulo é responsável por armazenar as instruções do programa. Quando o PC fornece um endereço, a ROM busca a instrução correspondente (IR - Instruction Register) e a disponibiliza para ser armazenada no IF_ID_Register e posteriormente decodificada no estágio ID.

Descrição em Verilog:

```
module IF_STAGE(
    clk,
    rst,
    clk_ROM,
    BranchTaken,
    BranchTarget,
    IF_ID_Instruction,
    IF_ID_PC
);

input wire  clk;
input wire  rst;
input wire  clk_ROM;
input wire  BranchTaken;
input wire  [7:0] BranchTarget;
```

```
output wire [31:0] IF_ID_Instruction;
output wire [7:0] IF_ID_PC;

wire    [31:0] SYNTHESIZED_WIRE_0;
wire    [7:0] SYNTHESIZED_WIRE_4;
wire    [7:0] SYNTHESIZED_WIRE_3;

IF_ID_Register  b2v_inst(
    .clk(clk),
    .rst(rst),
    .inInstruction(SYNTHESIZED_WIRE_0),
    .inPC(SYNTHESIZED_WIRE_4),
    .newPC(SYNTHESIZED_WIRE_3),
    .outInstruction(IF_ID_Instruction),
    .outPC(IF_ID_PC));

ROM b2v_inst3(
    .clock(clk_ROM),
    .address(SYNTHESIZED_WIRE_4),
    .q(SYNTHESIZED_WIRE_0));

Mux2to1 b2v_inst4(
    .sel(BranchTaken),
    .in0(SYNTHESIZED_WIRE_3),
    .in1(BranchTarget),
    .out(SYNTHESIZED_WIRE_4));

endmodule
```

ID (Instruction Decode): A instrução é decodificada, e o processador identifica os registradores e valores envolvidos. Os módulos funcionais e registradores internos serão:

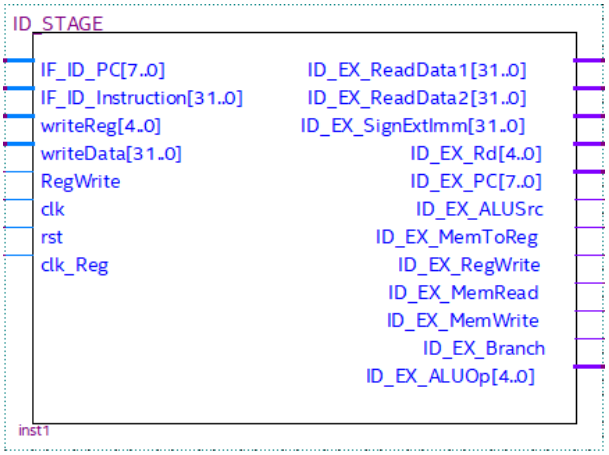

Figura 10 - Módulo ID Stage

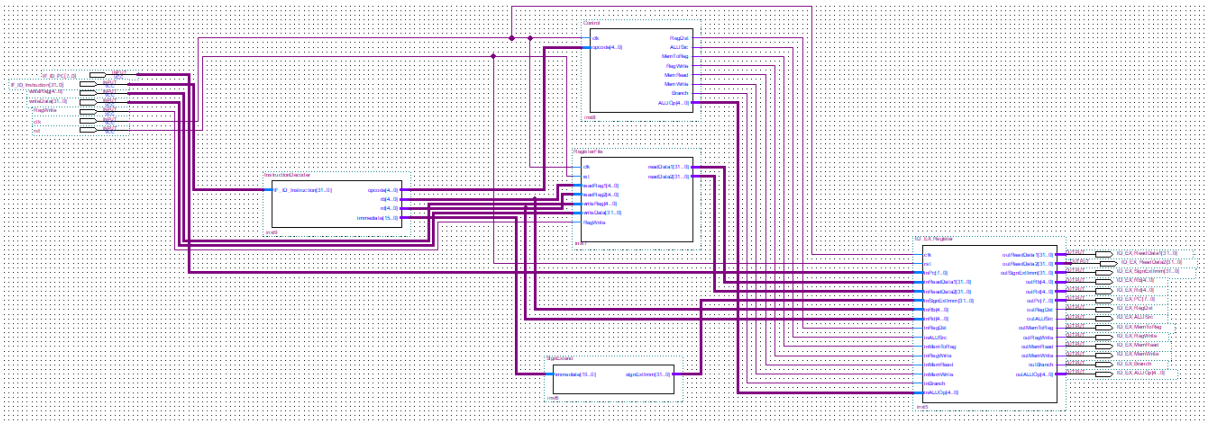

Figura 11 - Dentro do módulo ID Stage

Sinais de controle:

- **IF_ID_PC:** Contém o valor do Program Counter (PC) da instrução buscada. Esse valor é propagado para o próximo estágio para referência em cálculos de desvios ou acessos à memória.

- **IF_ID_Instruction:** Representa a instrução completa buscada da memória, que será decodificada para determinar os operandos, destino e operação a ser realizada.

- **writeReg:** Especifica o registrador de destino para operações que gravam valores no banco de registradores. Esse registrador é determinado pelo formato da instrução (R, I, J).

- **writeData:** Contém o dado a ser escrito no registrador de destino (writeReg). Esse valor pode vir da ALU, da memória ou de um imediato estendido.

- **RegWrite:** Sinal de controle que habilita a escrita no banco de registradores. Quando ativado, permite que o valor em writeData seja armazenado no registrador especificado por writeReg.
- **clk:** Sinal de clock que sincroniza a execução do estágio de decodificação, garantindo a transição correta dos valores.
- **rst:** Sinal de reset, utilizado para inicializar ou limpar os valores dos registradores de pipeline durante a reinicialização do processador.
- **ID_EX_ReadData1:** Representa o primeiro operando lido do banco de registradores. Esse valor será utilizado como entrada na ALU para operações aritméticas e lógicas.
- **ID_EX_ReadData2:** Representa o segundo operando lido do banco de registradores. Esse valor pode ser usado na ALU ou como dado a ser armazenado na memória.
- **ID_EX_SignExtImm:** Guarda o valor imediato estendido para 32 bits. Esse valor é utilizado em operações que envolvem constantes, como instruções do tipo I (addi, lw, sw).
- **ID_EX_Rd:** Contém o número do registrador de destino (campo rd da instrução), que pode ser utilizado para armazenar o resultado de uma operação ALU.
- **ID_EX_PC:** Propaga o valor do PC para o estágio EX, utilizado em cálculos de desvios condicionais e chamadas de subrotinas.
- **ID_EX_ALUSrc:** Define a segunda entrada da ALU. Quando ativado, a ALU usará um valor imediato (SignExtImm) em vez do conteúdo de um registrador.
- **ID_EX_MemToReg:** Controla se o valor a ser escrito no registrador virá da memória (lw) ou da ALU (operações aritméticas/lógicas).
- **ID_EX_RegWrite:** Habilita a escrita no banco de registradores no estágio WB.
- **ID_EX_MemRead:** Ativado quando uma instrução de leitura de memória (lw) está presente, permitindo a busca de dados na memória RAM.
- **ID_EX_MemWrite:** Ativado quando uma instrução de escrita na memória (sw) está presente, permitindo o armazenamento de valores na RAM.
- **ID_EX_Branch:** Indica se a instrução atual é uma instrução de desvio (beq, bne), permitindo que o pipeline avalie a necessidade de alterar o fluxo de execução.
- **ID_EX_ALUOp:** Define a operação que a ALU deve executar, baseado no opcode da instrução. Esse sinal determina se a operação será soma, subtração, AND, OR, shift, etc.

Módulos do estágio:

- **InstructionDecoder:** Responsável por decodificar a instrução recebida do estágio IF. Ele extrai os diferentes campos da instrução, como opcode, registradores-fonte, registrador de destino e operandos imediatos. Esses campos são enviados para os módulos de controle e banco de registradores para determinar a operação a ser realizada.

- **Control:** Gera os sinais de controle do processador com base no opcode da instrução. Esse módulo define o comportamento do pipeline, ativando sinais como RegWrite, MemRead, MemWrite, ALUSrc e Branch, garantindo que a execução da instrução ocorra corretamente nos estágios seguintes.

- **RegisterFile:** Implementa o banco de registradores, permitindo a leitura e escrita de dados. Ele recebe os endereços dos registradores especificados na instrução, fornecendo seus valores para a execução da operação. Caso a instrução seja do tipo escrita, ele atualiza o conteúdo do registrador de destino com o dado apropriado.

- **SignExtend:** Realiza a extensão de sinal para operandos imediatos de 16 bits, convertendo-os para 32 bits. Essa conversão é necessária para garantir operações corretas na ALU, especialmente em instruções que utilizam valores constantes.

- **ID_EX_Register:** Funciona como um registrador de pipeline, armazenando os sinais de controle e os dados processados no estágio ID para serem utilizados no estágio EX. Ele garante que as informações sejam mantidas corretamente ao longo do pipeline, evitando perda de dados e possibilitando a sincronização entre estágios.

Descrição em Verilog:

```
module ID_STAGE(
        RegWrite,
        clk,
        rst,
        clk_Reg,
        IF_ID_Instruction,
        IF_ID_PC,
        writeData,
        writeReg,
        ID_EX_ALUSrc,
        ID_EX_MemToReg,
        ID_EX_RegWrite,
        ID_EX_MemRead,
        ID_EX_MemWrite,
        ID_EX_Branch,
        ID_EX_ALUOp,
        ID_EX_PC,
        ID_EX_Rd,
```

```
        ID_EX_ReadData1,
        ID_EX_ReadData2,
        ID_EX_SignExtImm
);


input wire      RegWrite;
input wire      clk;
input wire      rst;
input wire      clk_Reg;
input wire      [31:0] IF_ID_Instruction;
input wire      [7:0] IF_ID_PC;
input wire      [31:0] writeData;
input wire      [4:0] writeReg;
output wire     ID_EX_ALUSrc;
output wire     ID_EX_MemToReg;
output wire     ID_EX_RegWrite;
output wire     ID_EX_MemRead;
output wire     ID_EX_MemWrite;
output wire     ID_EX_Branch;
output wire     [4:0] ID_EX_ALUOp;
output wire     [7:0] ID_EX_PC;
output wire     [4:0] ID_EX_Rd;
output wire     [31:0] ID_EX_ReadData1;
output wire     [31:0] ID_EX_ReadData2;
output wire     [31:0] ID_EX_SignExtImm;

wire    SYNTHESIZED_WIRE_0;
wire    SYNTHESIZED_WIRE_1;
wire    SYNTHESIZED_WIRE_2;
wire    SYNTHESIZED_WIRE_3;
wire    SYNTHESIZED_WIRE_4;
wire    SYNTHESIZED_WIRE_5;
wire    SYNTHESIZED_WIRE_6;
wire    [4:0] SYNTHESIZED_WIRE_7;
wire    [4:0] SYNTHESIZED_WIRE_17;
wire    [4:0] SYNTHESIZED_WIRE_18;
wire    [31:0] SYNTHESIZED_WIRE_10;
wire    [31:0] SYNTHESIZED_WIRE_11;
wire    [31:0] SYNTHESIZED_WIRE_12;
wire    [15:0] SYNTHESIZED_WIRE_13;
wire    [4:0] SYNTHESIZED_WIRE_16;

ID_EX_Register  b2v_inst5(
        .clk(clk),
        .rst(rst),
        .inRegDst(SYNTHESIZED_WIRE_0),
        .inALUSrc(SYNTHESIZED_WIRE_1),
        .inMemToReg(SYNTHESIZED_WIRE_2),
        .inRegWrite(SYNTHESIZED_WIRE_3),
        .inMemRead(SYNTHESIZED_WIRE_4),
        .inMemWrite(SYNTHESIZED_WIRE_5),
        .inBranch(SYNTHESIZED_WIRE_6),
        .inALUOp(SYNTHESIZED_WIRE_7),
        .inPc(IF_ID_PC),
        .inRb(SYNTHESIZED_WIRE_17),
```

```
        .inRd(SYNTHESIZED_WIRE_18),
        .inReadData1(SYNTHESIZED_WIRE_10),
        .inReadData2(SYNTHESIZED_WIRE_11),
        .inSignExtImm(SYNTHESIZED_WIRE_12),
        .outALUSrc(ID_EX_ALUSrc),
        .outMemToReg(ID_EX_MemToReg),
        .outRegWrite(ID_EX_RegWrite),
        .outMemRead(ID_EX_MemRead),
        .outMemWrite(ID_EX_MemWrite),
        .outBranch(ID_EX_Branch),
        .outALUOp(ID_EX_ALUOp),
        .outPc(ID_EX_PC),
        .outRd(ID_EX_Rd),
        .outReadData1(ID_EX_ReadData1),
        .outReadData2(ID_EX_ReadData2),
        .outSignExtImm(ID_EX_SignExtImm));


SignExtend      b2v_inst6(
        .immediate(SYNTHESIZED_WIRE_13),
        .signExtImm(SYNTHESIZED_WIRE_12));


RegisterFile    b2v_inst7(
        .clk(clk_Reg),
        .rst(rst),
        .RegWrite(RegWrite),
        .readReg1(SYNTHESIZED_WIRE_17),
        .readReg2(SYNTHESIZED_WIRE_18),
        .writeData(writeData),
        .writeReg(writeReg),
        .readData1(SYNTHESIZED_WIRE_10),
        .readData2(SYNTHESIZED_WIRE_11));


Control b2v_inst8(
        .opcode(SYNTHESIZED_WIRE_16),
        .RegDst(SYNTHESIZED_WIRE_0),
        .ALUSrc(SYNTHESIZED_WIRE_1),
        .MemToReg(SYNTHESIZED_WIRE_2),
        .RegWrite(SYNTHESIZED_WIRE_3),
        .MemRead(SYNTHESIZED_WIRE_4),
        .MemWrite(SYNTHESIZED_WIRE_5),
        .Branch(SYNTHESIZED_WIRE_6),
        .ALUOp(SYNTHESIZED_WIRE_7));
        defparam b2v_inst8.ADD = 5'b00110;
        defparam b2v_inst8.AND = 5'b01010;
        defparam b2v_inst8.BRFL = 5'b10010;
        defparam b2v_inst8.CALL = 5'b10011;
        defparam b2v_inst8.CMP = 5'b01110;
        defparam b2v_inst8.DIV = 5'b01001;
        defparam b2v_inst8.JPC = 5'b10001;
        defparam b2v_inst8.JR = 5'b10000;
        defparam b2v_inst8.LW_1 = 5'b00000;
        defparam b2v_inst8.LW_2 = 5'b00001;
        defparam b2v_inst8.LW_3 = 5'b00010;
```

```
        defparam b2v_inst8.MOV = 5'b00101;
        defparam b2v_inst8.MUL = 5'b01000;
        defparam b2v_inst8.NOP = 5'b10101;
        defparam b2v_inst8.NOT = 5'b01111;
        defparam b2v_inst8.OR = 5'b01011;
        defparam b2v_inst8.RET = 5'b10100;
        defparam b2v_inst8.SHL = 5'b01100;
        defparam b2v_inst8.SHR = 5'b01101;
        defparam b2v_inst8.SUB = 5'b00111;
        defparam b2v_inst8.SW_1 = 5'b00011;
        defparam b2v_inst8.SW_2 = 5'b00100;


InstructionDecoder      b2v_inst9(
        .IF_ID_Instruction(IF_ID_Instruction),
        .immediate(SYNTHESIZED_WIRE_13),
        .opcode(SYNTHESIZED_WIRE_16),
        .rb(SYNTHESIZED_WIRE_17),
        .rd(SYNTHESIZED_WIRE_18));


endmodule
```

EX (Execute): A ULA executa a operação aritmética ou lógica especificada. Os módulos funcionais e registradores internos serão:

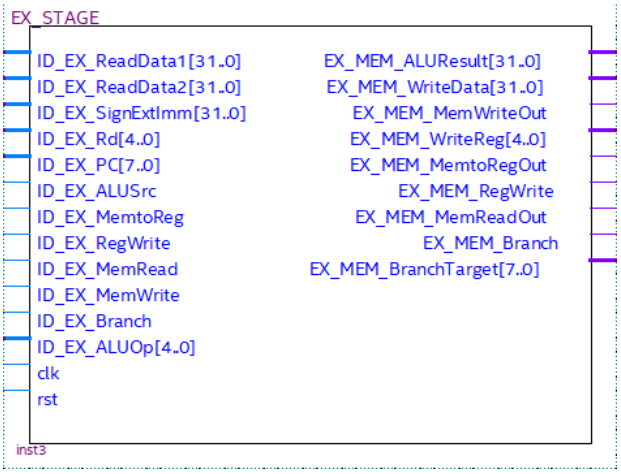

Figura 12 - Módulo EX Stage

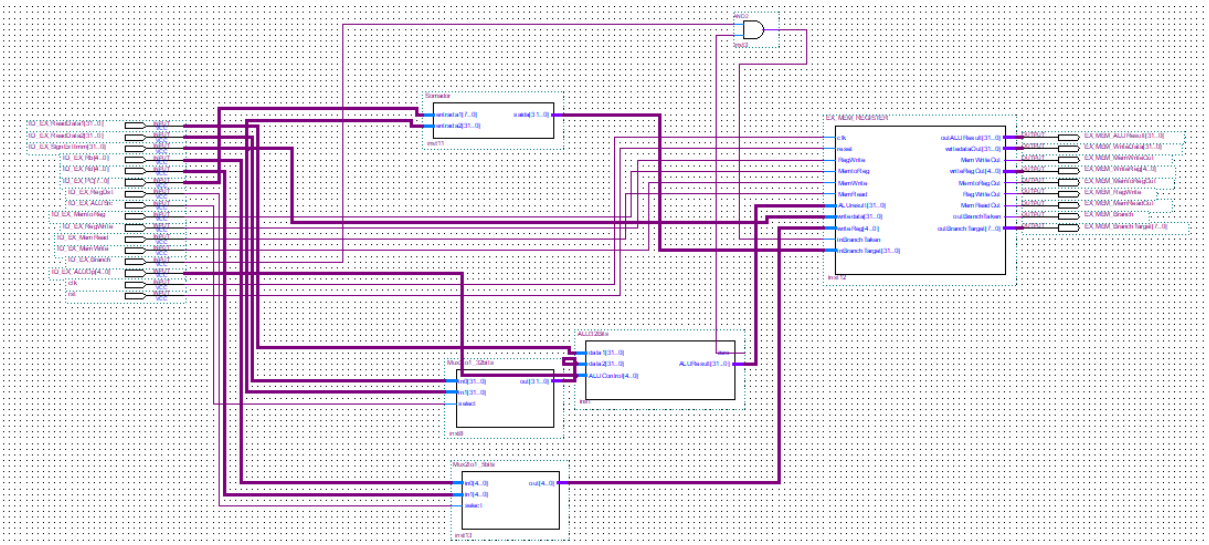

Figura 13 - Dentro do módulo EX Stage

Universidade Federal da Bahia
Escola Politécnica
Departamento de Engenharia Elétrica e de Computação
Docente: Wagner Oliveira
Discentes: Caio Mendes, Fábio Miguel, Gerson Daniel, Heverton Reis e José Alves
Semestre 2024.2

Sinais de controle:

- **ID_EX_ReadData1:** Contém o primeiro operando vindo do banco de registradores. Esse valor é utilizado como entrada na ALU para operações aritméticas e lógicas.
- **ID_EX_ReadData2:** Contém o segundo operando vindo do banco de registradores. Dependendo do valor de `ALUSrc`, ele pode ser usado diretamente na ALU ou como dado a ser armazenado na memória.
- **ID_EX_SignExtImm:** Representa o valor imediato estendido para 32 bits. Esse valor pode ser utilizado na ALU em operações que envolvem constantes.
- **ID_EX_Rd:** Contém o número do registrador de destino (`rd`), utilizado para armazenar o resultado de operações ALU em instruções do tipo R.
- **ID_EX_PC:** Propaga o valor do PC para o estágio EX, sendo utilizado em cálculos de desvios condicionais e instruções que alteram o fluxo de execução.
- **ID_EX_ALUSrc:** Define se a ALU utilizará o segundo operando vindo do registrador (`ReadData2`) ou um valor imediato (`SignExtImm`).
- **ID_EX_MemtoReg:** Indica se o valor a ser escrito no registrador virá da memória (`lw`) ou da ALU.
- **ID_EX_RegWrite:** Habilita a escrita no banco de registradores no estágio WB.
- **ID_EX_MemRead:** Ativado quando a instrução requer uma leitura de memória (`lw`).
- **ID_EX_MemWrite:** Ativado quando a instrução requer uma escrita na memória (`sw`).
- **ID_EX_Branch:** Indica se a instrução atual é uma instrução de desvio (`beq`, `bne`), permitindo que o pipeline avalie a necessidade de alterar o fluxo de execução.
- **ID_EX_ALUOp:** Define a operação que a ALU deve executar, determinando se será uma soma, subtração, operação lógica ou shift.
- **clk:** Sinal de clock que sincroniza as operações do estágio EX, garantindo a execução no momento correto.
- **rst:** Sinal de reset utilizado para inicializar ou limpar os valores dos registradores de pipeline durante a reinicialização do processador.
- **EX_MEM_ALUResult:** Contém o resultado da operação realizada pela ALU. Esse valor pode ser armazenado em um registrador ou usado como endereço de memória em instruções de acesso à RAM.
- **EX_MEM_WriteData** Contém o valor a ser escrito na memória quando a instrução `sw` está em execução. Esse valor vem originalmente do registrador `rt`.
- **EX_MEM_MemWriteOut**: Sinal que indica se a operação atual é uma escrita na memória (`sw`).
- **EX_MEM_MemReadOut:** Sinal que indica se a operação atual é uma leitura de memória (`lw`).
- **EX_MEM_WriteReg:** Define o registrador de destino que receberá o resultado da ALU ou da memória no estágio WB.
- **EX_MEM_MemtoRegOut:** Indica se o valor a ser escrito no registrador no estágio WB virá da memória ou da ALU.
- **EX_MEM_RegWrite:** Propaga o sinal de escrita em registradores para o estágio WB, garantindo que os resultados das operações sejam armazenados corretamente.

- **EX_MEM_Branch:** Indica se a instrução atual é um desvio condicional e se o processador deve alterar o fluxo de execução.
- **EX_MEM_BranchTarget:** Contém o endereço de destino calculado para instruções de desvio, permitindo que o PC seja atualizado corretamente no estágio MEM caso o branch seja tomado.

Módulos do estágio:

- **Mux2to1_5bits:** Multiplexador de 2 entradas e 1 saída, operando com sinais de 5 bits. Ele seleciona qual registrador de destino (rd ou rt) será utilizado para armazenar o resultado da ALU, com base no sinal de controle RegDst.

- **Mux2to1_32bits:** Multiplexador de 2 entradas e 1 saída, operando com sinais de 32 bits. Ele seleciona se o segundo operando da ALU será um valor vindo do registrador (ReadData2) ou um valor imediato estendido (SignExtImm), de acordo com o sinal de controle ALUSrc.

- **ALU32Bits:** Unidade Lógica e Aritmética (ALU) de 32 bits, responsável por realizar operações matemáticas e lógicas, como soma, subtração, AND, OR e shifts. As operações são definidas pelo sinal de controle ALUOp. O resultado da ALU é armazenado e enviado para os estágios posteriores.

- **Somador:** Realiza a soma do valor do PC (ID_EX_PC) com o deslocamento do desvio, gerando o endereço de destino para instruções de branch (beq, bne). Esse resultado é propagado como EX_MEM_BranchTarget.

- **EX_MEM_Register:** Registrador de pipeline que armazena os sinais de controle e os valores gerados no estágio EX, garantindo a transferência correta das informações para o estágio MEM. Ele mantém valores como ALUResult, WriteData, MemRead, MemWrite e Branch, evitando perda de dados entre estágios.

```
module EX_STAGE(
        ID_EX_ALUSrc,
        rst,
        clk,
        ID_EX_RegWrite,
        ID_EX_MemtoReg,
        ID_EX_MemWrite,
        ID_EX_MemRead,
        ID_EX_Branch,
        ID_EX_ALUOp,
        ID_EX_PC,
        ID_EX_Rd,
        ID_EX_ReadData1,
```

```
        ID_EX_ReadData2,
        ID_EX_SignExtImm,
        EX_MEM_MemtoRegOut,
        EX_MEM_MemWriteOut,
        EX_MEM_MemReadOut,
        EX_MEM_Branch,
        EX_MEM_RegWrite,
        EX_MEM_ALUResult,
        EX_MEM_BranchTarget,
        EX_MEM_WriteData,
        EX_MEM_WriteReg
);


input wire      ID_EX_ALUSrc;
input wire      rst;
input wire      clk;
input wire      ID_EX_RegWrite;
input wire      ID_EX_MemtoReg;
input wire      ID_EX_MemWrite;
input wire      ID_EX_MemRead;
input wire      ID_EX_Branch;
input wire      [4:0] ID_EX_ALUOp;
input wire      [7:0] ID_EX_PC;
input wire      [4:0] ID_EX_Rd;
input wire      [31:0] ID_EX_ReadData1;
input wire      [31:0] ID_EX_ReadData2;
input wire      [31:0] ID_EX_SignExtImm;
output wire     EX_MEM_MemtoRegOut;
output wire     EX_MEM_MemWriteOut;
output wire     EX_MEM_MemReadOut;
output wire     EX_MEM_Branch;
output wire     EX_MEM_RegWrite;
output wire     [31:0] EX_MEM_ALUResult;
output wire     [7:0] EX_MEM_BranchTarget;
output wire     [31:0] EX_MEM_WriteData;
output wire     [4:0] EX_MEM_WriteReg;

wire    [31:0] SYNTHESIZED_WIRE_0;
wire    SYNTHESIZED_WIRE_1;
wire    [31:0] SYNTHESIZED_WIRE_2;
wire    [31:0] SYNTHESIZED_WIRE_3;
wire    SYNTHESIZED_WIRE_4;

ALU32Bits       b2v_inst(
        .ALUControl(ID_EX_ALUOp),
        .data1(ID_EX_ReadData1),
        .data2(SYNTHESIZED_WIRE_0),
        .PC(ID_EX_PC),
        .zero(SYNTHESIZED_WIRE_4),
        .ALUResult(SYNTHESIZED_WIRE_2),
        .BranchTarget(SYNTHESIZED_WIRE_3));
        defparam b2v_inst.ADD = 5'b00110;
        defparam b2v_inst.AND = 5'b01010;
        defparam b2v_inst.BRFL = 5'b10010;
        defparam b2v_inst.CALL = 5'b10011;
```

```
        defparam b2v_inst.CMP = 5'b01110;
        defparam b2v_inst.DIV = 5'b01001;
        defparam b2v_inst.JPC = 5'b10001;
        defparam b2v_inst.JR = 5'b10000;
        defparam b2v_inst.LW_1 = 5'b00000;
        defparam b2v_inst.LW_2 = 5'b00001;
        defparam b2v_inst.LW_3 = 5'b00010;
        defparam b2v_inst.MOV = 5'b00101;
        defparam b2v_inst.MUL = 5'b01000;
        defparam b2v_inst.NOP = 5'b10101;
        defparam b2v_inst.NOT = 5'b01111;
        defparam b2v_inst.OR = 5'b01011;
        defparam b2v_inst.RET = 5'b10100;
        defparam b2v_inst.SHL = 5'b01100;
        defparam b2v_inst.SHR = 5'b01101;
        defparam b2v_inst.SUB = 5'b00111;
        defparam b2v_inst.SW_1 = 5'b00011;
        defparam b2v_inst.SW_2 = 5'b00100;


EX_MEM_REGISTER  b2v_inst12(
        .clk(clk),
        .reset(rst),
        .RegWrite(ID_EX_RegWrite),
        .MemtoReg(ID_EX_MemtoReg),
        .MemWrite(ID_EX_MemWrite),
        .MemRead(ID_EX_MemRead),
        .inBranchTaken(SYNTHESIZED_WIRE_1),
        .ALUresult(SYNTHESIZED_WIRE_2),
        .inBranchTarget(SYNTHESIZED_WIRE_3),
        .writedata(ID_EX_SignExtImm),
        .writeReg(ID_EX_Rd),
        .MemWriteOut(EX_MEM_MemWriteOut),
        .MemtoRegOut(EX_MEM_MemtoRegOut),
        .RegWriteOut(EX_MEM_RegWrite),
        .MemReadOut(EX_MEM_MemReadOut),
        .outBranchTaken(EX_MEM_Branch),
        .outALUResult(EX_MEM_ALUResult),
        .outBranchTarget(EX_MEM_BranchTarget),
        .writedataOut(EX_MEM_WriteData),
        .writeRegOut(EX_MEM_WriteReg));

assign  SYNTHESIZED_WIRE_1 = ID_EX_Branch & SYNTHESIZED_WIRE_4;


Mux2to1_32bits  b2v_inst8(
        .select(ID_EX_ALUSrc),
        .in0(ID_EX_ReadData2),
        .in1(ID_EX_SignExtImm),
        .out(SYNTHESIZED_WIRE_0));


endmodule
```

MEM (Memory Access): Para operações de carga (LW) ou armazenamento (SW), o processador lê ou escreve dados na memória. Os módulos funcionais e registradores internos serão:

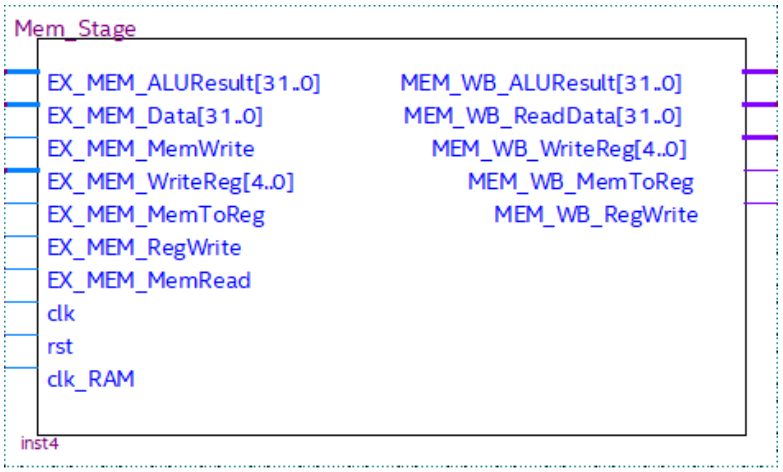

Figura 14 - Módulo MEM Stage

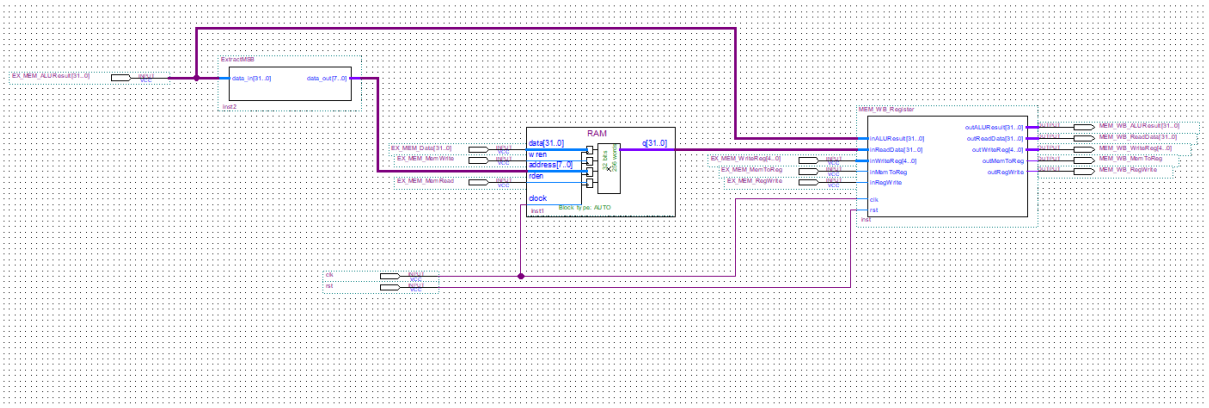

Figura 15 - Dentro do módulo EX Stage

Sinais de controle:

- **EX_MEM_ALUResult:** Contém o resultado da ALU, que pode ser um endereço de memória (para `lw` ou `sw`) ou um valor computado para ser escrito em um registrador.
- **EX_MEM_Data:** Valor a ser escrito na memória caso a instrução seja um `sw` (store word).
- **EX_MEM_MemWrite:** Sinal de controle que indica se a instrução atual requer uma escrita na memória (`sw`).
- **EX_MEM_WriteReg:** Número do registrador de destino que pode receber um valor no estágio WB (Write Back).

- **EX_MEM_MemToReg:** Indica se o valor escrito no registrador virá da memória (`lw`) ou da ALU.
- **EX_MEM_RegWrite:** Propaga o controle de escrita no banco de registradores para o estágio WB.
- **EX_MEM_MemRead:** Indica se a instrução atual requer uma leitura de memória (`lw`).
- **clk:** Sinal de clock para sincronização do estágio MEM com o pipeline.
- **clk_RAM:** Sinal de clock para sincronização das leituras e escritas da memória RAM
- **rst:** Sinal de reset utilizado para inicializar ou limpar os registradores de pipeline.

Saídas do estágio MEM (passadas para MEM/WB):

- **MEM_WB_ALUResult:** Propaga o resultado da ALU para o estágio WB (caso a instrução não seja um `lw`).
- **MEM_WB_ReadData:** Contém o valor carregado da memória em uma instrução `lw`, que será escrito no banco de registradores no estágio WB.
- **MEM_WB_WriteReg:** Mantém o número do registrador de destino que será escrito no estágio WB.
- **MEM_WB_MemToReg:** Indica se o valor a ser escrito no registrador no estágio WB virá da memória (`lw`) ou da ALU.
- **MEM_WB_RegWrite:** Sinal que confirma se a instrução exige escrita no banco de registradores no estágio WB.

Módulos do estágio:

- **ExtractMSB:** Extrai o bit mais significativo (MSB) de um valor, podendo ser usado para identificar sinais importantes como o bit de sinal em operações aritméticas ou para controle de fluxo no pipeline.

- **RAM:** Representa a memória de dados do processador, permitindo leitura (lw) e escrita (sw) de palavras de 32 bits, conforme os sinais de controle do estágio MEM.

- **MEM_WB_Register:** Registrador de pipeline que armazena os resultados do estágio MEM e os propaga para o estágio WB, garantindo a transferência correta de dados entre os estágios do pipeline.

Descrição em Verilog:

```
module Mem_Stage(
        clk,
        EX_MEM_MemWrite,
        rst,
        EX_MEM_MemToReg,
        EX_MEM_RegWrite,
        EX_MEM_MemRead,
        clk_RAM,
        EX_MEM_ALUResult,
```

```
        EX_MEM_Data,
        EX_MEM_WriteReg,
        MEM_WB_MemToReg,
        MEM_WB_RegWrite,
        MEM_WB_ALUResult,
        MEM_WB_ReadData,
        MEM_WB_WriteReg
);


input wire      clk;
input wire      EX_MEM_MemWrite;
input wire      rst;
input wire      X MEN MemToReg;
input wire      EX_MEM_RegWrite;
input wire      EX_MEM_MemRead;
input wire      clk_RAM;
input wire      [31:0] EX_MEM_ALUResult;
input wire      [31:0] EX_MEM_Data;
input wire      [4:0] EX_MEM_WriteReg;
output wire     MEM_WB_MemToReg;
output wire     MEM_WB_RegWrite;
output wire     [31:0] MEM_WB_ALUResult;
output wire     [31:0] MEM_WB_ReadData;
output wire     [4:0] MEM_WB_WriteReg;

wire    [31:0] SYNTHESIZED_WIRE_0;
wire    [7:0] SYNTHESIZED_WIRE_1;

MEM_WB_Register  b2v_inst(
        .clk(clk),
        .rst(rst),
        .inMemToReg(EX_MEM_MemToReg),
        .inRegWrite(EX_MEM_RegWrite),
        .inALUResult(EX_MEM_ALUResult),
        .inReadData(SYNTHESIZED_WIRE_0),
        .inWriteReg(EX_MEM_WriteReg),
        .outMemToReg(MEM_WB_MemToReg),
        .outRegWrite(MEM_WB_RegWrite),
        .outALUResult(MEM_WB_ALUResult),
        .outReadData(MEM_WB_ReadData),
        .outWriteReg(MEM_WB_WriteReg));

RAM     b2v_inst1(
        .wren(EX_MEM_MemWrite),
        .rden(EX_MEM_MemRead),
        .clock(clk_RAM),
        .address(SYNTHESIZED_WIRE_1),
        .data(EX_MEM_Data),
        .q(SYNTHESIZED_WIRE_0));

ExtractMSB      b2v_inst2(
        .data_in(EX_MEM_ALUResult),
        .data_out(SYNTHESIZED_WIRE_1));

endmodule
```

WB (Write Back): O resultado da operação é escrito de volta no Banco de Registradores. Os módulos funcionais e registradores internos serão:

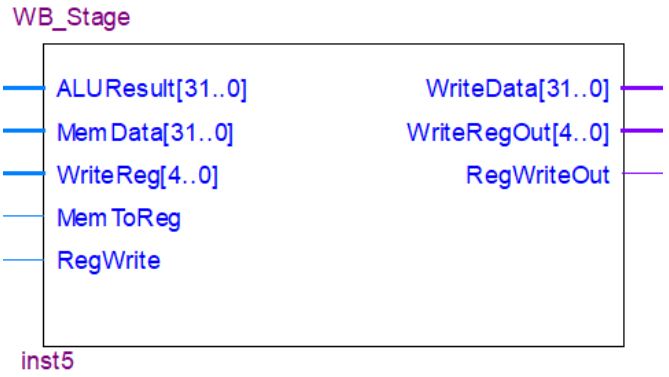

Figura 16 -Módulo WB Stage

Sinais de controle:

- **ALUResult:** Contém o resultado da operação realizada pela ALU no estágio EX, que pode ser escrito de volta no banco de registradores.
- **MemData:** Valor lido da memória de dados no estágio MEM, usado em instruções de carregamento (lw).
- **WriteReg:** Número do registrador de destino que receberá o valor escrito no banco de registradores.
- **MemToReg:** Define se o valor a ser escrito no registrador virá da memória (MemData) ou do resultado da ALU (ALUResult).
- **RegWrite:** Sinal de controle que habilita a escrita no banco de registradores.
- **WriteData:** Dado final a ser escrito no registrador de destino, selecionado entre ALUResult e MemData com base no controle MemToReg.
- **WriteRegOut:** Número do registrador de destino propagado para o banco de registradores.
- **RegWriteOut:** Sinal de controle propagado que confirma a habilitação da escrita no banco de registradores.

Universidade Federal da Bahia
Escola Politécnica
Departamento de Engenharia Elétrica e de Computação
Docente: Wagner Oliveira
Discentes: Caio Mendes, Fábio Miguel, Gerson Daniel, Heverton Reis e José Alves
Semestre 2024.2

Descrição em Verilog:

```
module WB_Stage (
         input wire [31:0] ALUResult,        // Resultado da ALU
         input wire [31:0] MemData,          // Dado lido da memória
         input wire [4:0] WriteReg,          // Endereço do registrador de destino
         input wire MemToReg,                // Seleciona entre dado da memória ou ALU
         input wire RegWrite,                // Habilita escrita no registrador
         output wire [31:0] WriteData,       // Dado a ser escrito no registrador
         output wire [4:0] WriteRegOut,      // Endereço do registrador destino (saída)
         output wire RegWriteOut             // Sinal de controle da escrita (saída)
);

    // Multiplexador para selecionar a fonte do dado a ser escrito
    assign WriteData = (MemToReg) ? MemData : ALUResult;

    // Propagar os sinais de controle e endereço do registrador
    assign WriteRegOut = WriteReg;
    assign RegWriteOut = RegWrite;

endmodule
```

4. Tabela de sinais de controle

Definidos os sinais de controle de cada estágio do pipeline, podemos montar, portanto, uma tabela para cada instrução, evidenciando os sinais de controle que ela aciona.

Opcode	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch
LW_1	1	1	1	1	0	0
LW_2	0	1	1	1	0	0
LW_3	1	1	1	1	0	0
SW_1	1	X	0	0	1	0
SW_2	0	X	0	0	1	0
MOV	0	0	1	0	0	0
ADD	0	0	1	0	0	0
SUB	0	0	1	0	0	0
MUL	0	0	1	0	0	0
DIV	0	0	1	0	0	0
AND	0	0	1	0	0	0
OR	0	0	1	0	0	0
SHL	0	0	1	0	0	0
SHR	0	0	1	0	0	0
CMP	0	0	0	0	0	0
NOT	0	0	1	0	0	0
JR	0	0	0	0	0	1
JPC	1	0	0	0	0	1
BRFL	0	0	0	0	0	1
CALL	0	0	1	0	0	1
RET	0	0	0	0	0	1