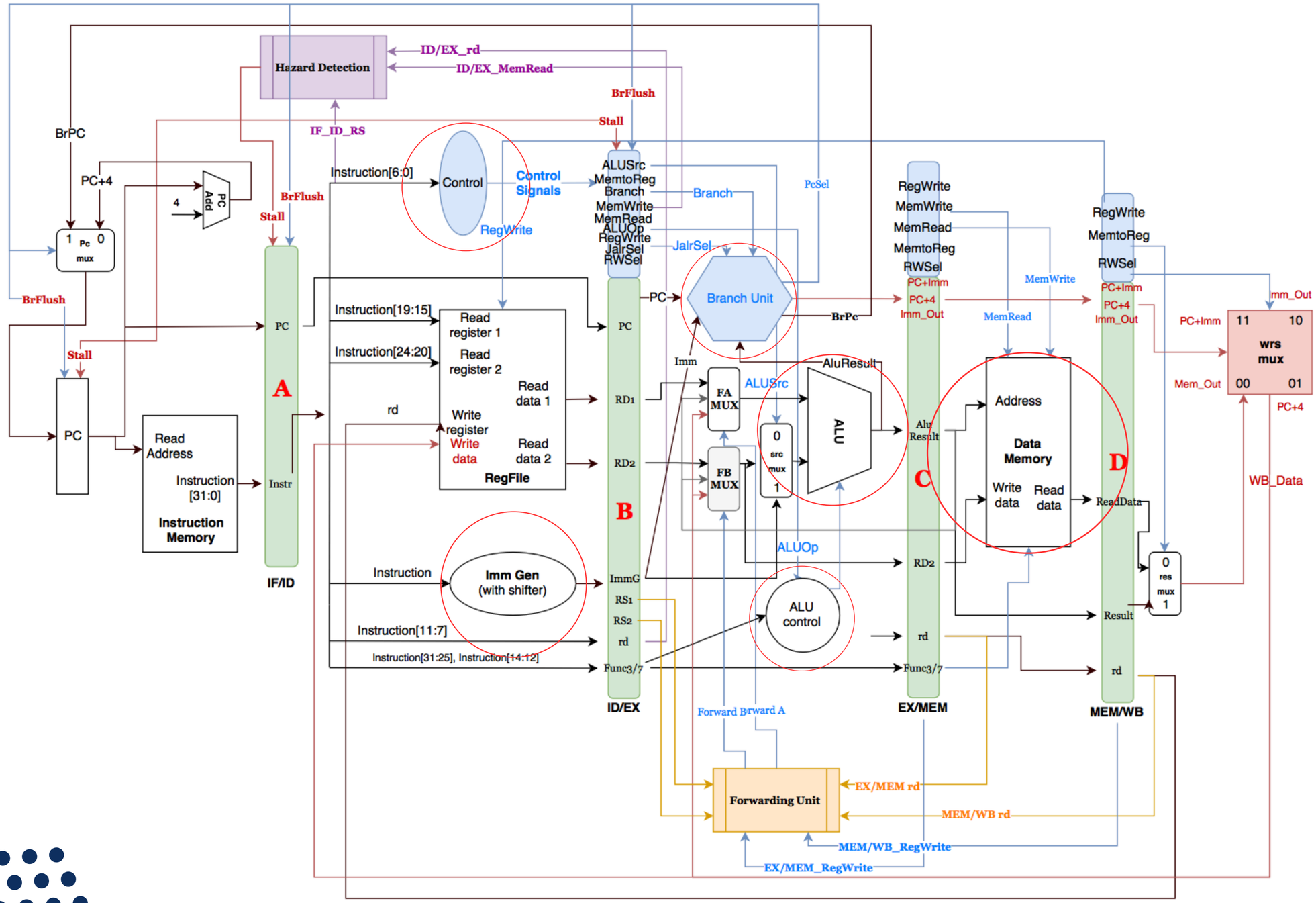
PROJETO RISC-V

Implementando o ISA RV32I

INTEGRANTES

Carolina Gabriela de Arruda Brito dos Santos (cgabs)
Isabela Maria De Moura Nascimento (immn)
Karina Lima de Oliveira (klo)
Kauanny Karolinna Davalon Araujo E Barros (kkdab)
Maria Clara Laranjeira Tenorio (mclt)


Hazard Detection
ID/EX_rd
ID/EX_MemRead
BrFlush
Stall
IF_ID_RS
BrPC
PC+4
4
PC Add
1 Pc 0 mux
BrFlush
Stall
PC
Read Address
Instruction [31:0]
Instruction Memory
PC
Instr
A
IF/ID
Instruction[6:0]
Control
Control Signals
RegWrite
ALUSrc
MemtoReg
Branch
MemWrite
MemRead
ALUOp
RegWrite
JalrSel
RWSel
Branch
PcSel
JalrSel
Branch Unit
PC
BrPc
Instruction[19:15]
Instruction[24:20]
rd
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
RegFile
PC
RD1
RD2
B
ImmG
RS1
RS2
rd
Func3/7
ID/EX
Imm
FA MUX
FB MUX
ALUSrc
0 src mux 1
ALU
AluResult
ALUOp
ALU control
Instruction
Imm Gen (with shifter)
Instruction[11:7]
Instruction[31:25], Instruction[14:12]
RegWrite
MemWrite
MemRead
MemtoReg
RWSel
PC+Imm
PC+4
Imm_Out
Alu Result
RD2
C
rd
Func3/7
EX/MEM
MemWrite
MemRead
Address
Data Memory
Write data
Read data
RegWrite
MemtoReg
RWSel
PC+Imm
PC+4
Imm_Out
ReadData
D
Result
rd
MEM/WB
mm_Out
PC+Imm
11
10
wrs mux
Mem_Out
00
01
PC+4
0 res mux 1
WB_Data
Forward Brward A
Forwarding Unit
EX/MEM rd
MEM/WB rd
MEM/WB_RegWrite
EX/MEM_RegWrite

Quais arquivos foram modificados?

Branch Unit

ALUController

Controller

imm_Gen

ALU

Datapath

RISC_V

RegPack

DataMemory

assembler.py

Datapath

- Responsável pela execução das instruções, movimentação de dados e operações lógicas e aritméticas.
- Para suportar o halt, o jal e o jalr, adicionamos três novos sinais de controle.
- No bloco ID_EX_Reg B, adicionamos B.Halt, B.JALRSel e B.JALSel, então atribuimos Halt, JALRSel e JALSel a eles, respectivamente. Para que não sejam ignorados.
- No bloco BranchUnit #(9) brunit, adicionamos B.JALSel, B.JALRSel e B.Halt. Para a unidade de controle de desvios suportas as adições.
- Os blocos EX_MEM_Reg C e MEM_WB_Reg D foram alterados para permitir que sejam reconhecidos na fase MEM e processados na fase WB
- Nova multiplexação no ultimo bloco para selecionar o valor final correto para o endereço de retorno.

Datapath

```
input  logic                clk,
reset,
RegWrite,
MemtoReg,  // Register file writing enable   // Memory or ALU MUX
ALUsrc,
MemWrite,  // Register file or Immediate MUX // Memroy Writing Enable
MemRead,  // Memroy Reading Enable
Branch,  // Branch Enable
JALRSel,                          // Jump and link enable.
JALSel,                          // Jump and link register enable.
Halt,
input  logic [1:0] ALUOp,
input  logic [ALU_CC_W-1:0] ALU_CC,         // ALU Control Code ( input of the ALU )
output logic [6:0] opcode,
output logic [6:0] Funct7,
output logic [2:0] Funct3,
output logic [1:0] ALUOp_Current,
output logic [DATA_W-1:0] WB_Data,        //Result After the last MUX
```

```
always @(posedge clk) begin
  if ((reset) || (Reg_Stall) || (PcSel))   // initialization or flush or generate a NOP if hazard
      begin
    B.ALUSrc <= 0;
    B.Halt <= 0;
    B.MemtoReg <= 0;
    B.RegWrite <= 0;
    B.MemRead <= 0;
    B.MemWrite <= 0;
    B.ALUOp <= 0;
    B.Branch <= 0;
    B.JALRSel <= 0;
    B.JALSel <= 0;
    B.Curr_Pc <= 0;
    B.RD_One <= 0;
    B.RD_Two <= 0;
    B.RS_One <= 0;
    B.RS_Two <= 0;
    B.rd <= 0;
    B.ImmG <= 0;
    B.func3 <= 0;
    B.func7 <= 0;
    B.Curr_Instr <= A.Curr_Instr;  //debug tmp
```

```
  end else begin
    B.ALUSrc <= ALUsrc;
    B.Halt <= Halt;
    B.MemtoReg <= MemtoReg;
    B.RegWrite <= RegWrite;
    B.MemRead <= MemRead;
    B.MemWrite <= MemWrite;
    B.ALUOp <= ALUOp;
    B.Branch <= Branch;
    B.JALRSel <= JALRSel;
    B.JALSel <= JALSel;
    B.Curr_Pc <= A.Curr_Pc;
    B.RD_One <= Reg1;
    B.RD_Two <= Reg2;
    B.RS_One <= A.Curr_Instr[19:15];
    B.RS_Two <= A.Curr_Instr[24:20];
    B.rd <= A.Curr_Instr[11:7];
    B.ImmG <= ExtImm;
    B.func3 <= A.Curr_Instr[14:12];
    B.func7 <= A.Curr_Instr[31:25];
    B.Curr_Instr <= A.Curr_Instr;  //debug tmp
  end
end
```

```
// EX_MEM_Reg C;
always @(posedge clk) begin
  if (reset)   // initialization
      begin
    C.RegWrite <= 0;
    C.MemtoReg <= 0;
    C.MemRead <= 0;
    C.MemWrite <= 0;
    C.JALSel <= 0;
    C.Pc_Imm <= 0;
    C.Pc_Four <= 0;
    C.Imm_Out <= 0;
    C.Alu_Result <= 0;
    C.RD_Two <= 0;
    C.rd <= 0;
    C.func3 <= 0;
    C.func7 <= 0;
  end else begin
    C.RegWrite <= B.RegWrite;
    C.MemtoReg <= B.MemtoReg;
    C.MemRead <= B.MemRead;
    C.MemWrite <= B.MemWrite;
    C.JALSel <= B.JALSel;
    C.Pc_Imm <= BrImm;
    C.Pc_Four <= Old_PC_Four;
    C.Imm_Out <= B.ImmG;
    C.Alu_Result <= ALUResult;
    C.RD_Two <= FBmux_Result;
    C.rd <= B.rd;
    C.func3 <= B.func3;
    C.func7 <= B.func7;
    C.Curr_Instr <= B.Curr_Instr;  // debug tmp
  end
end
```

Datapath

```
BranchUnit #(9) brunit (
    B.Curr_Pc,
    B.ImmG,
    B.Branch,
    B.JALSel,
    B.JALRSel,
    B.Halt,
    ALUResult,
    BrImm,
    Old_PC_Four,
    BrPC,
    PcSel
);
```

Datapath

```
// MEM_WB_Reg D;
always @(posedge clk) begin
  if (reset)   // initialization
      begin
    D.RegWrite <= 0;
    D.MemtoReg <= 0;
    D.JALSel <= 0;
    D.Pc_Imm <= 0;
    D.Pc_Four <= 0;
    D.Imm_Out <= 0;
    D.Alu_Result <= 0;
    D.MemReadData <= 0;
    D.rd <= 0;
  end else begin
    D.RegWrite <= C.RegWrite;
    D.MemtoReg <= C.MemtoReg;
    D.JALSel <= C.JALSel;
    D.Pc_Imm <= C.Pc_Imm;
    D.Pc_Four <= C.Pc_Four;
    D.Imm_Out <= C.Imm_Out;
    D.Alu_Result <= C.Alu_Result;
    D.MemReadData <= ReadData;
    D.rd <= C.rd;
    D.Curr_Instr <= C.Curr_Instr;
  end
end
```

```
mux2 #(32) resmux (
    D.Alu_Result,
    D.MemReadData,
    D.MemtoReg,
    temp
);
mux2 #(32) jalmux(
    temp,
        D.Pc_Four,
        D.JALSel,
        WrmuxSrc
);
```

RegPack

```
// Reg B
typedef struct packed {
  logic        ALUSrc;
  logic        Halt;
  logic        MemtoReg;
  logic        RegWrite;
  logic        MemRead;
  logic        MemWrite;
  logic [1:0]  ALUOp;
  logic        Branch;
  logic        JALRSel;
  logic        JALSel;
  logic [8:0]  Curr_Pc;
  logic [31:0] RD_One;
  logic [31:0] RD_Two;
  logic [4:0]  RS_One;
  logic [4:0]  RS_Two;
  logic [4:0]  rd;
  logic [31:0] ImmG;
  logic [2:0]  func3;
  logic [6:0]  func7;
  logic [31:0] Curr_Instr;
} id_ex_reg;
```

```
// Reg C
typedef struct packed {
  logic        RegWrite;
  logic        MemtoReg;
  logic        MemRead;
  logic        MemWrite;
  logic        JALSel;
  logic [31:0] Pc_Imm;
  logic [31:0] Pc_Four;
  logic [31:0] Imm_Out;
  logic [31:0] Alu_Result;
  logic [31:0] RD_Two;
  logic [4:0]  rd;
  logic [2:0]  func3;
  logic [6:0]  func7;
  logic [31:0] Curr_Instr;
} ex_mem_reg;

// Reg D
typedef struct packed {
  logic        RegWrite;
  logic        MemtoReg;
  logic        JALSel;
  logic [31:0] Pc_Imm;
  logic [31:0] Pc_Four;
  logic [31:0] Imm_Out;
  logic [31:0] Alu_Result;
  logic [31:0] MemReadData;
  logic [4:0]  rd;
  logic [31:0] Curr_Instr;
} mem_wb_reg;
```

- **Otimiza o salvamento e restauração de registradores em chamadas de função e trocas de contexto, reduzindo acessos à memória e melhorando a eficiência.**

- **Compacta e recompacta registradores, economizando espaço e acelerando operações**

imm_Gen

31 30 25	24 21 20	19 15	14 12	11 8	7	6 0	
funct7	rs2	rs1	funct3	rd		opcode	R-type
imm[11:0]		rs1	funct3	rd		opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]		opcode	S-type
imm[12] imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]				rd		opcode	U-type
imm[20] imm[10:1]	imm[11]	imm[19:12]		rd		opcode	J-type

Figure 2.3: RISC-V base instruction formats showing immediate variants.

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

imm_Gen

```
`timescale 1ns / 1ps

module imm_Gen (
    input  logic [31:0] inst_code,
    output logic [31:0] Imm_out
);

  always_comb
    case (inst_code[6:0])
      7'b0000011:  /*I-type load part*/
      Imm_out = {inst_code[31] ? 20'hFFFFF : 20'b0, inst_code[31:20]};

      7'b0010011:  /*I-type*/ // Imediatas
        case(inst_code[14:12])
        3'b001: /*SLLI*/
              Imm_out = inst_code[24:20];
        3'b101: /*SRLI, SRAI*/
              Imm_out = inst_code[24:20];
        default: /*ADDI, SLTI*/
              Imm_out = {inst_code[31] ? 20'hFFFFF : 20'b0, inst_code[31:20]};
        endcase
```

```
7'b0100011:  /*S-type*/
Imm_out = {inst_code[31] ? 20'hFFFFF : 20'b0, inst_code[31:25], inst_code[11:7]};

7'b1100011:  /*B-type*/
Imm_out = {
  inst_code[31] ? 19'h7FFFF : 19'b0,
  inst_code[31],
  inst_code[7],
  inst_code[30:25],
  inst_code[11:8],
  1'b0
};

7'b1101111: /*J-type*/ // JAL
Imm_out = {
        inst_code[31] ? 11'h7FF : 11'b0,
        inst_code[31],
        inst_code[19:12],
        inst_code[20],
        inst_code[30:21],
        1'b0
      };
```

```
      7'b1100111: // JALR
      Imm_out = {inst_code[31] ? 20'hFFFFF : 20'b0, inst_code[31:20]};

      default: Imm_out = {32'b0};
    endcase
endmodule
```

Controller

```
logic [6:0] R_TYPE, LW, SW, BR, I_TYPE, JAL, JALR, HALT;

assign R_TYPE = 7'b0110011;  //funcoes que usam reg
assign I_TYPE = 7'b0010011; //funcoes imediatas
assign LW = 7'b0000011;  //lw
assign SW = 7'b0100011;  //sw
assign BR = 7'b1100011;  //beq
assign JAL = 7'b1101111; // JAL
assign JALR = 7'b1100111; // JALR
assign HALT = 7'b1111111; //HALT
```

- **ALUSrc:**
  - **0 - 2º operador é um registrador**
  - **1 - 2º operador refere-se ao valor imediato ou deslocamento**
- **MemtoReg:**
  - **0 - Valor a ser escrito vai ser recebido da ALU**
  - **1 - Valor a ser escrito vai ser recebido do data memory**
- **RegWrite: caso 1, permite a escrita no registrador de destino**
- **MemRead : caso 1, permite a leitura da memória**
- **MemWrite: Caso 1, permite escrever na memória de dados**
- **ALUOp: 00: LW/SW; 01:Branch; 10: R\I_type; 11 - JALR**

```
  assign ALUSrc = (Opcode == LW || Opcode == SW || Opcode == I_TYPE || Opcode == JALR);
  assign MemtoReg = (Opcode == LW);
  assign RegWrite = (Opcode == R_TYPE || Opcode == LW || Opcode == I_TYPE || Opcode == JAL || Opcode == JALR);
  assign MemRead = (Opcode == LW);
  assign MemWrite = (Opcode == SW);
  assign ALUOp[0] = (Opcode == BR || Opcode == JALR);
  assign ALUOp[1] = (Opcode == R_TYPE || Opcode == I_TYPE || Opcode == JALR);
  assign Branch = (Opcode == BR);
  assign JALSel = (Opcode == JAL || Opcode == JALR);
  assign JALRSel = (Opcode == JALR);
  assign Halt = (Opcode == HALT);
endmodule
```

assembler.py

- **Arquivo em Python responsável pela “tradução” das instruções dadas, utilizando opcode, registradores e valores usados para cada instrução.**

- **OPCODE**

```
# bits 0-6
opcode = {
 "lui": "0110111",
```

•••

```
 "and": "0110011",
 "halt": "1111111",
}
```

- **TRADUTOR**

```
# translates an instruction to binary (assembly to machine code)
def translate_instruction(instruction):
        instr = instruction.split(" ")[0].strip()

        if instr == "halt":
                binary = "0" * 25 + opcode[instr]  #Correção do halt (32 bits)
                return binary
```

000: 01111111; -- halt
001: 00000000;
002: 00000000;
003: 00000000;

ALU

```
case(Operation)
4'b0000:        // AND
        ALUResult = SrcA & SrcB;
                  4'b0001:        // OR
        ALUResult = SrcA | SrcB;
4'b0010:        // XOR
        ALUResult = SrcA ^ SrcB;
4'b0011:        // ADD, ADDI, LW, SW
        ALUResult = $signed(SrcA) + $signed(SrcB);
4'b0100:        // BNE
        ALUResult = (SrcA != SrcB) ? 1 : 0;
4'b0101:        // BLT
        ALUResult = (SrcA < SrcB) ? 1 : 0;
4'b0110:        // BGE
        ALUResult = (SrcA >= SrcB) ? 1 : 0;
4'b0111:        // SLT, SLTI
        ALUResult = (SrcA < SrcB) ? 1 : 0;
4'b1000:        // BEQ
        ALUResult = (SrcA == SrcB) ? 1 : 0;
4'b1001:        // SUB
        ALUResult = $signed(SrcA) - $signed(SrcB);
```

Realiza as operações em si

```
4'b1010:        // SRAI
        ALUResult = $signed(SrcA) >>> $signed(SrcB);
4'b1011:        // SRLI
        ALUResult = SrcA >> SrcB;
4'b1100:        // SLLI
        ALUResult = SrcA << SrcB;
4'b1101:       // JAL/JALR
        ALUResult = SrcA;
default:
        ALUResult = 0;
```

ALU Controller

- Padrão de AluOp, Funct3 e Funct7 foram visualizados através da documentação do RISC-V e do Controller
- ALUOp: 00: LW/SW; 01:Branch; 10: R\I_type; 11 - JALR
- Ex: SUB (1001)

```
assign Operation[3] = (((ALUOp == 2'b01) && (Funct3 == 3'b000)) ||  // Branch-Beq
    ((ALUOp == 2'b10) && (Funct3 == 3'b000) && (Funct7 == 7'b0100000)) || // R/I-Sub
    ((ALUOp == 2'b10) && (Funct3 == 3'b101) && (Funct7 == 7'b0100000)) || // R/I-Srai
    ((ALUOp == 2'b10) && (Funct3 == 3'b101) && (Funct7 == 7'b0000000)) || // R/I-Srli
    ((ALUOp == 2'b10) && (Funct3 == 3'b001) && (Funct7 == 7'b0000000)) || // R/I-Slli
    ((ALUOp == 2'b11) && (Funct3 == 3'b000))); // U-Jal/Jarl
```

Branch Unit

```
module BranchUnit #(
    parameter PC_W = 9
) (
    input logic [PC_W-1:0] Cur_PC,
    input logic [31:0] Imm,
    input logic Branch,
    input logic JALSel,
    input logic JALRSel,
    input logic Halt,
    input logic [31:0] ALUResult,
    output logic [31:0] PC_Imm,
    output logic [31:0] PC_Four,
    output logic [31:0] BrPC,
    output logic PcSel
);
```

- **Vai receber:**
  - **PC atual**
  - **Sinais de controle do Controller**
  - **Imm do Imm_Gen**
  - **ALUResult da ALU**
- **Vai enviar:**
  - **PcSel e BrPc para um mux para realizar a seleção da próxima instrução a ser lida**
  - **PC_Imm e PC_Four vão passar pelos estágios restantes até chegar na fase Write-Back, onde haverá um mux para escolher qual valor será escrito de volta no registrador de destino**

Branch Unit

```
logic Branch_Sel;
logic [31:0] PC_Full;

assign PC_Full = {23'b0, Cur_PC};

assign PC_Imm = (JALRSel) ? (ALUResult + Imm) : (PC_Full + Imm);
assign PC_Four = PC_Full + 32'b100;
assign Branch_Sel = (Branch && ALUResult[0]) || JALSel;

assign BrPC = (Branch_Sel) ? PC_Imm : ((Halt) ? PC_Full : 32'b0);
assign PcSel = Branch_Sel || JALSel || Halt;
```

- **PC_Full = zero-extension do Cur_PC**
- **PC_Imm = endereço de destino para saltos**
- **PC_Four = próxima instrução sequencial (PC+4)**
- **BrPC = endereço efetivo em caso de branch ou halt**
- **PcSel = sinal de seleção do próximo PC**

Datamemory

```
`timescale 1ns / 1ps

module datamemory #(
    parameter DM_ADDRESS = 9,
    parameter DATA_W = 32
) (
    input logic clk,
    input logic MemRead,  // comes from control unit
    input logic MemWrite,  // Comes from control unit
    input logic [DM_ADDRESS - 1:0] a,  // Read / Write address - 9 LSB bits of the ALU output
    input logic [DATA_W - 1:0] wd,  // Write Data
    input logic [2:0] Funct3,  // bits 12 to 14 of the instruction
    output logic [DATA_W - 1:0] rd  // Read Data
);

  logic [31:0] raddress;
  logic [31:0] waddress;
  logic [31:0] Datain;
  logic [31:0] Dataout;
  logic [ 3:0] Wr;

  Memoria32Data mem32 (
      .raddress(raddress),
      .waddress(waddress),
      .Clk(~clk),
      .Datain(Datain),
      .Dataout(Dataout),
      .Wr(Wr)
  );
```

- **recebe sinais de leitura e escrita, junto com um endereço de dados**
- **a: endereço da memoria que será acessado**
- **wd: dado que será escrito na memoria**
- **rd: dado que será lido da memória**

Datamemory

lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	

Datamemory

```
always_ff @(*) begin
  raddress = {{22{1'b0}}, a};
  waddress = {{22{1'b0}}, {a[8:2], {2{1'b0}}}};
  Datain = wd;
  Wr = 4'b0000;

  if (MemRead) begin
    case (Funct3)
      3'b010: begin //LW
        rd <= Dataout;
      end
      3'b000: begin //LB
        rd <= $signed(Dataout[7:0]);
      end
      3'b001:begin //LH
        rd <= $signed(Dataout[15:0]);
      end
      3'b100:begin //LBU
        rd <= $signed({24'b0,Dataout[7:0]});
      end
      default: rd <= Dataout;
    endcase
```

Datamemory

```
    end else if (MemWrite) begin
      case (Funct3)
        3'b010: begin  //SW
          Wr <= 4'b1111; //ativa a escrita de todos os 4 bytes (bits [31:0])
          Datain <= wd;
        end
        3'b000: begin //SB
          Wr <= 4'b0001; //ativa a escrita somente do byte menos significativo (bits [7:0])
          Datain[7:0] <= wd;
        end
        3'b001: begin //SH
          Wr <= 4'b0011; //ativa a escrita dos bytes menos significativos (bits [15:0])
          Datain[15:0] <= wd;
        end
        default: begin
          Wr <= 4'b1111;
          Datain <= wd;
        end
      endcase
    end
  end
```

RISC_V

```
`timescale 1ns / 1ps

module riscv #(
    parameter DATA_W = 32
) (
    input logic clk,
    reset,  // clock and reset signals
    output logic [31:0] WB_Data,  // The ALU_Result

    output logic [4:0] reg_num,
    output logic [31:0] reg_data,
    output logic reg_write_sig,

    output logic wr,
    output logic rd,
    output logic [8:0] addr,
    output logic [DATA_W-1:0] wr_data,
    output logic [DATA_W-1:0] rd_data
);
```

- **conecta os principais componentes do pipeline**

- **envia sinais de controle que serão usados para configurar a execução das instruções**

```
logic [6:0] opcode;
logic ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, JALRSel, JALSel, Halt;
logic [1:0] ALUop;
logic [1:0] ALUop_Reg;
logic [6:0] Funct7;
logic [2:0] Funct3;
logic [3:0] Operation;
```

RISC_V

```
Controller c (
    opcode,
    ALUSrc,
    MemtoReg,
    RegWrite,
    MemRead,
    MemWrite,
    ALUop,
    Branch,
    JALRSel,
    JALSel,
    Halt
);

ALUController ac (
    ALUop_Reg,
    Funct7,
    Funct3,
    Operation
);
```

```
Datapath dp (
    clk,
    reset,
    RegWrite,
    MemtoReg,
    ALUSrc,
    MemWrite,
    MemRead,
    Branch,
    JALRSel,
    JALSel,
    Halt,
    ALUop,
    Operation,
    opcode,
    Funct7,
    Funct3,
    ALUop_Reg,
    WB_Data,
    reg_num,
    reg_data,
    reg_write_sig,
    wr,
    rd,
    addr,
    wr_data,
    rd_data
);
```

OBRIGADA PELA ATENÇÃO!