Lab 6.

CPU Design

Instructors
Sergei Bykovskii, sergei_bykovskii@itmo.ru
Aglaia Ilina, agilina@itmo.ru

References

1. Stanislav Zhelnio, https://github.com/zhelnio/schoolRISCV
2. David Money Harris and Sarah L Harris. Digital Design and Computer Architecture
3. RISC-V Specification https://riscv.org/technical/specifications/

What is schoolRISCV core

– a simple processor core for practical teaching of the basics of digital circuit

– written in the Verilog language

– implements a subset of the RISCV architecture

– grew out of a similar schoolMIPS project

The features of schoolRISCV core

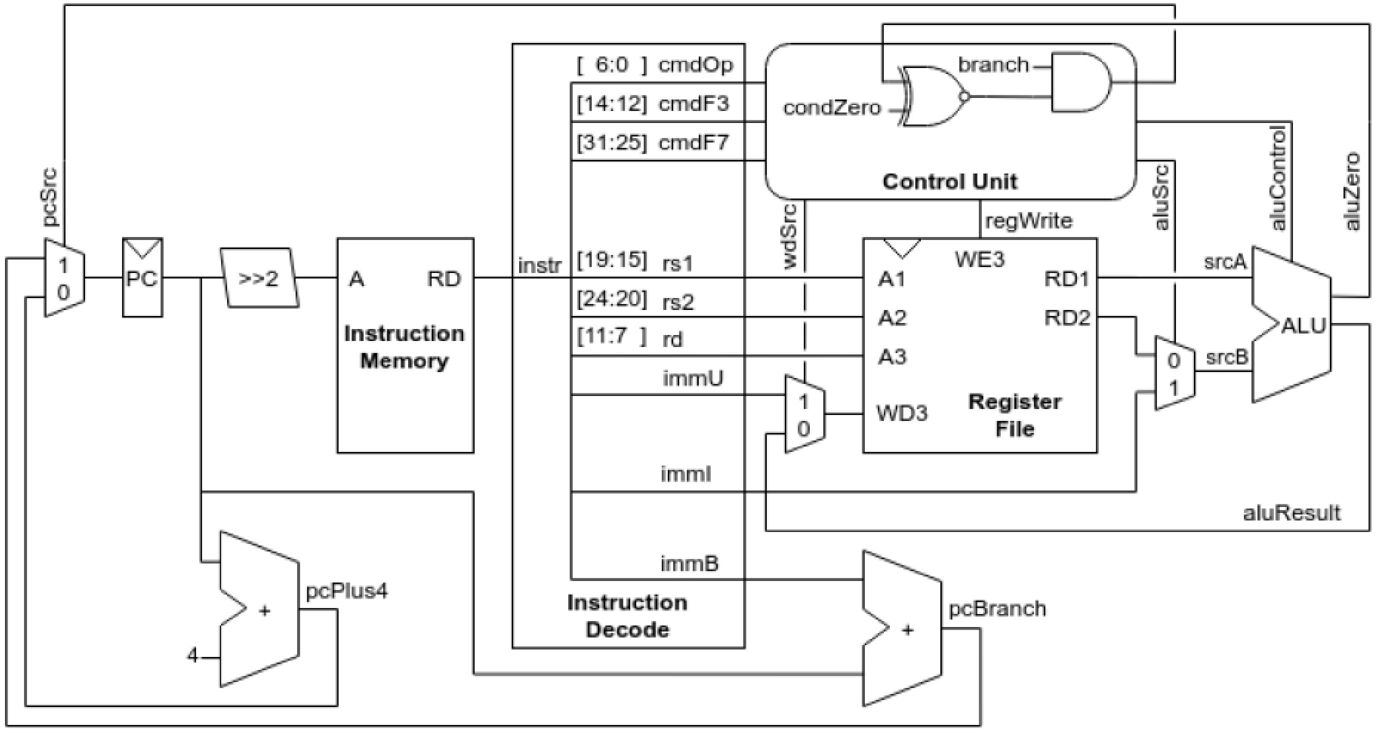
– One-cycle implementation

– Without data memory

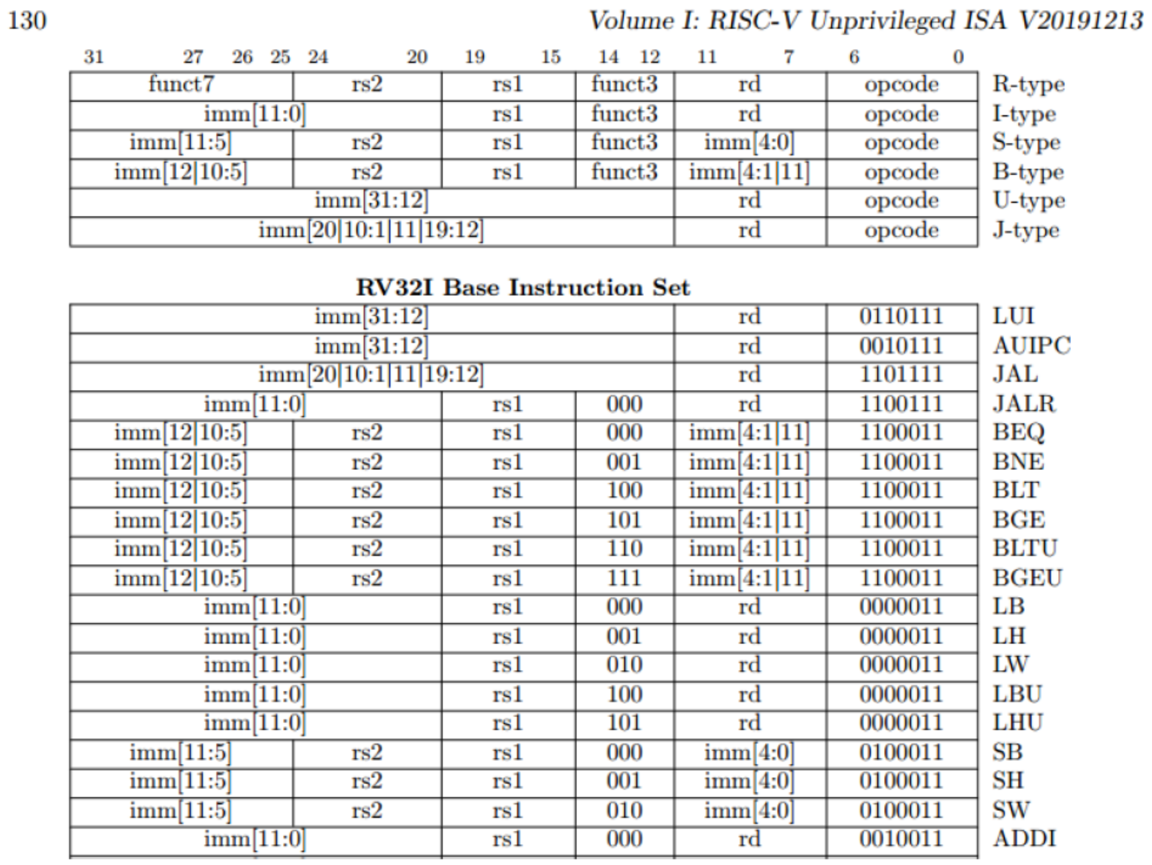
– Word address of instruction memory

– Supports 9 instructions: add, or, srl, sltu, sub, addi, lui, beq, bne

The microarchitecture of the processor



Instruction set

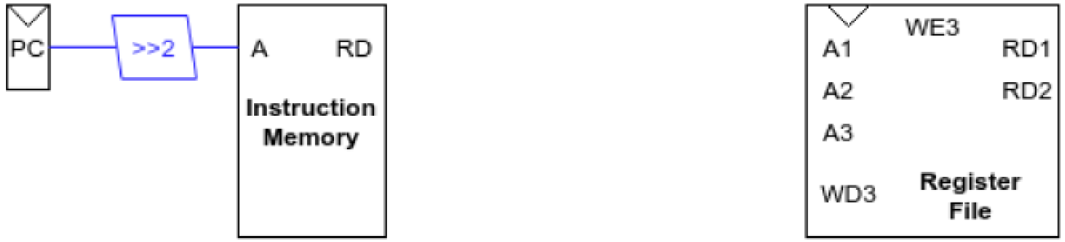


31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

RV32I Base Instruction Set

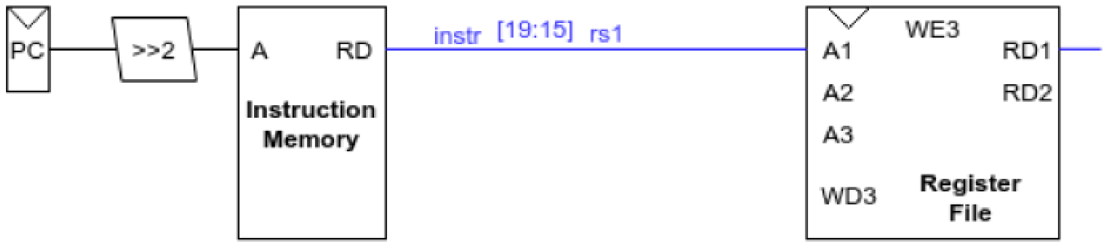
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI

ADDI: instruction fetching



HDU-ITMO Joint Institute
杭州电子科技大学 圣光机联合学院

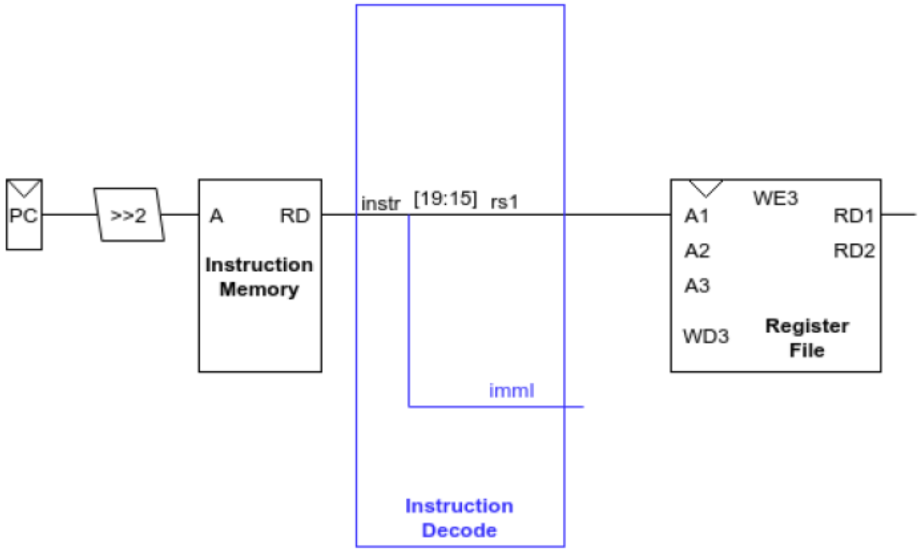
ADDI: reading the operand



ADDI: **I-type**, adds the sign-extended 12-bit immediate to register rs1: **rd = rs1 + imml**

31 ... 20	19 ... 15	14 ... 12	11 ... 7	6 ... 0	
imm[11:0]	rs1	funct3	rd	opcode	**I-type**

ADDI: decoding the constant

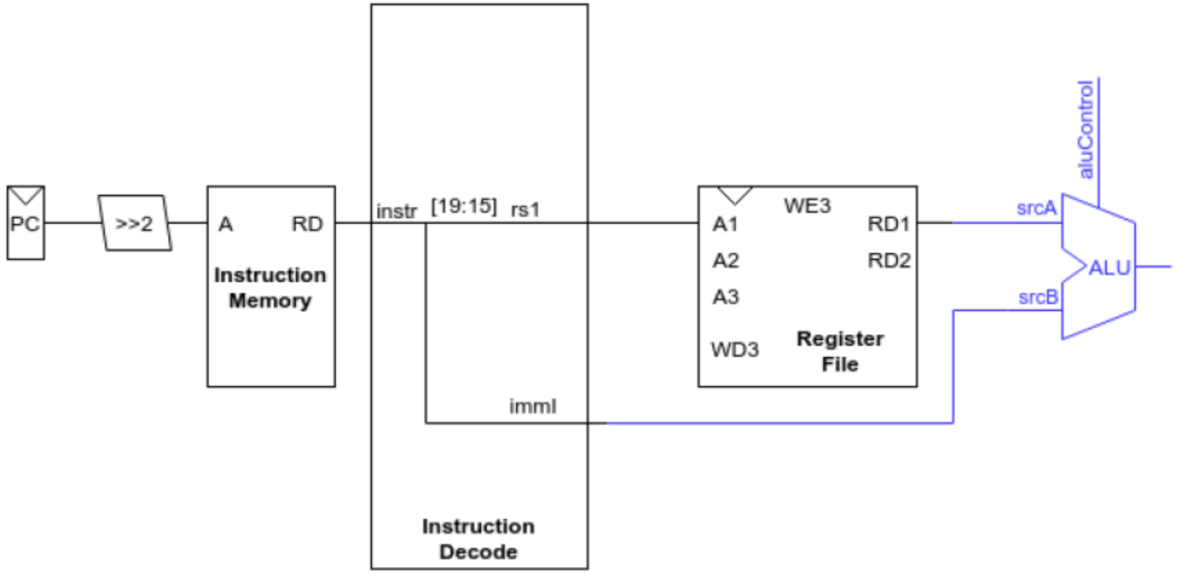


ADDI: **I-type**, adds the sign-extended 12-bit immediate to register rs1: **rd = rs1 + immI**

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
imm[11:0]	rs1	funct3	rd	opcode	**I-type**

31 … 11	10 … 5	4 … 1	0	
--- inst[31] ---	inst[30:25]	inst[24:21]	inst[20	**I-immediate**

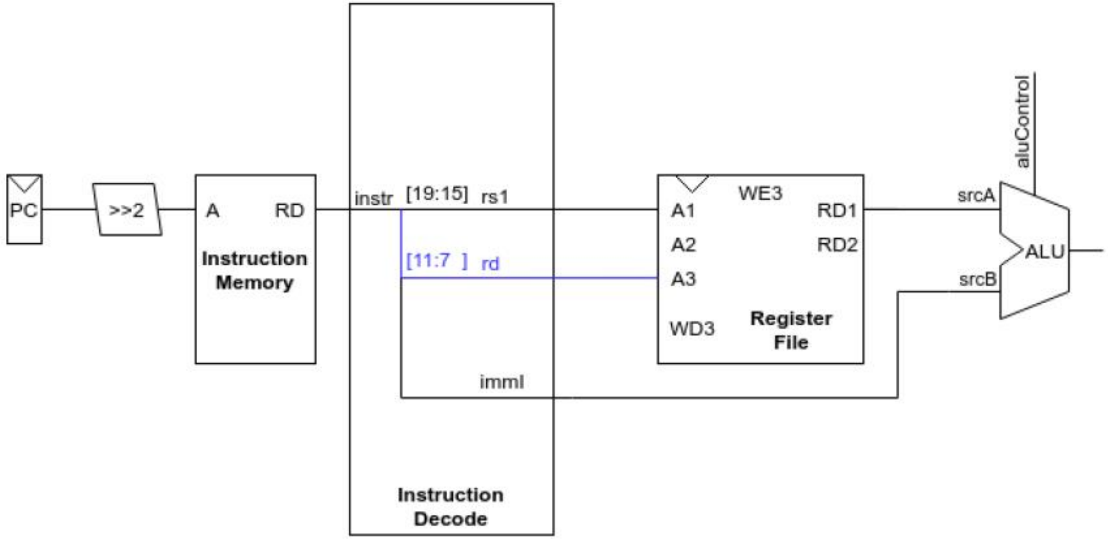
ADDI: calculation the result



ADDI: **I-type**, adds the sign-extended 12-bit immediate to register rs1: **rd = rs1 + immI**

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
imm[11:0]	rs1	**funct3**	rd	**opcode**	**I-type**

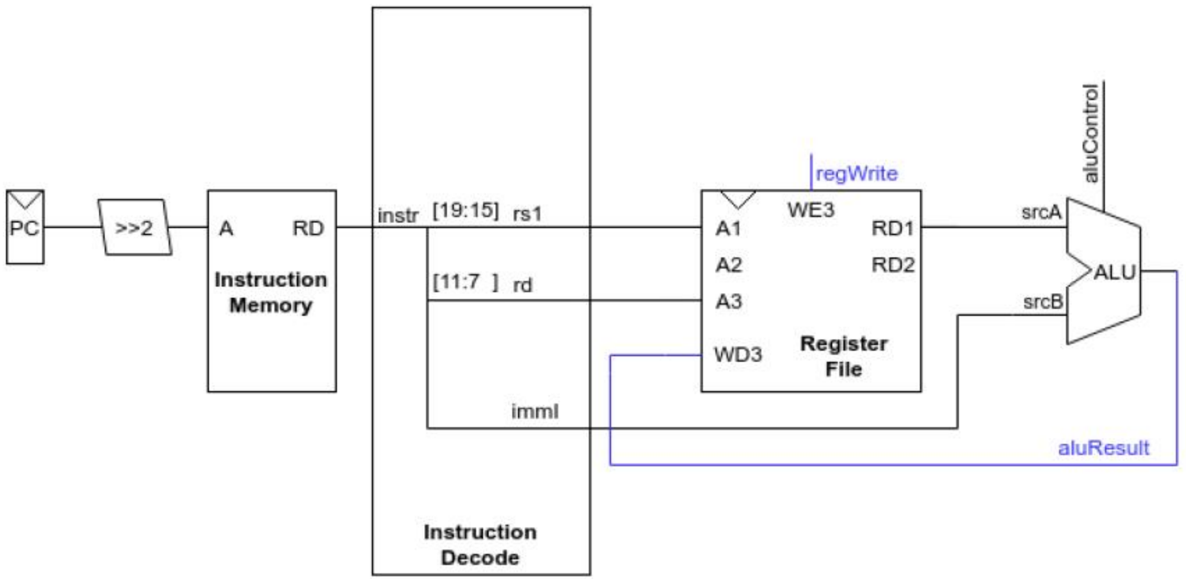
ADDI: decoding the destination address



ADDI: **I-type**, adds the sign-extended 12-bit immediate to register rs1: **rd = rs1 + imml**

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
imm[11:0]		rs1	funct3	**rd**	opcode	**I-type**

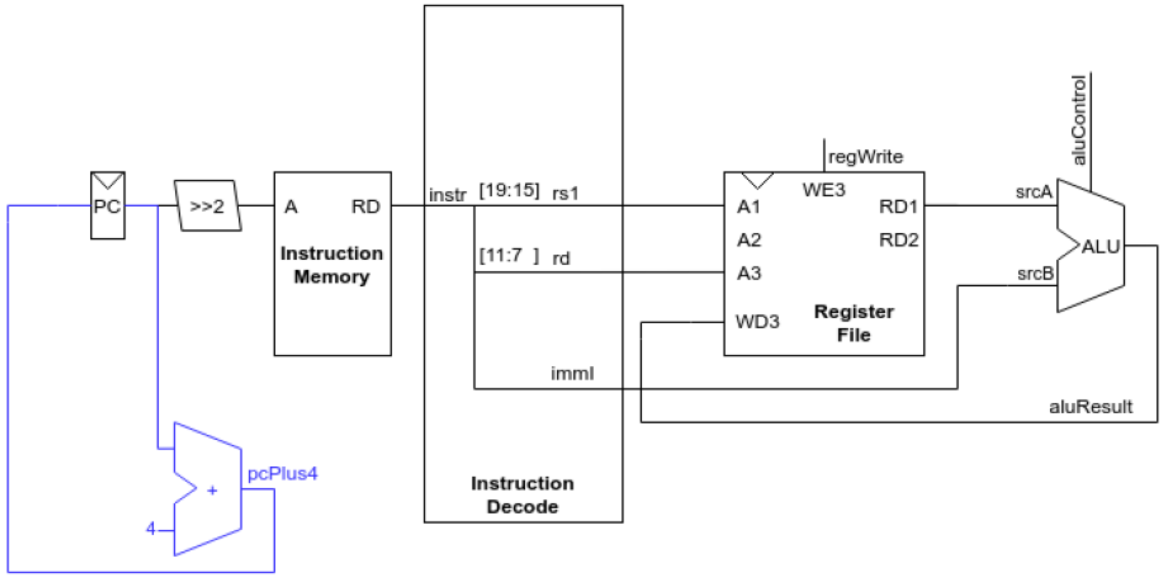
ADDI: writing the result



ADDI: **I-type**, adds the sign-extended 12-bit immediate to register rs1: **rd = rs1 + imml**

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
imm[11:0]		rs1	**funct3**	rd	**opcode**	**I-type**

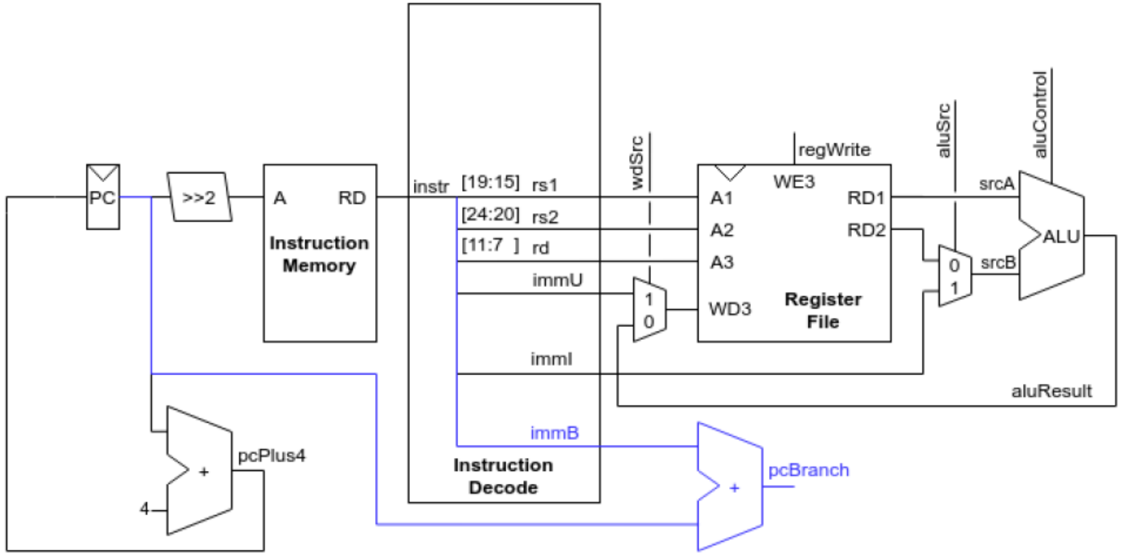
ADDI: calculation the next instruction address



ADDI: **I-type**, adds the sign-extended 12-bit immediate to register rs1: **rd = rs1 + imml**

31 ... 25 24 ... 20	19 ... 15	14 ... 12	11 ... 7	6 ... 0	
imm[11:0]	rs1	funct3	rd	opcode	**I-type**

BEQ: branch address calculation

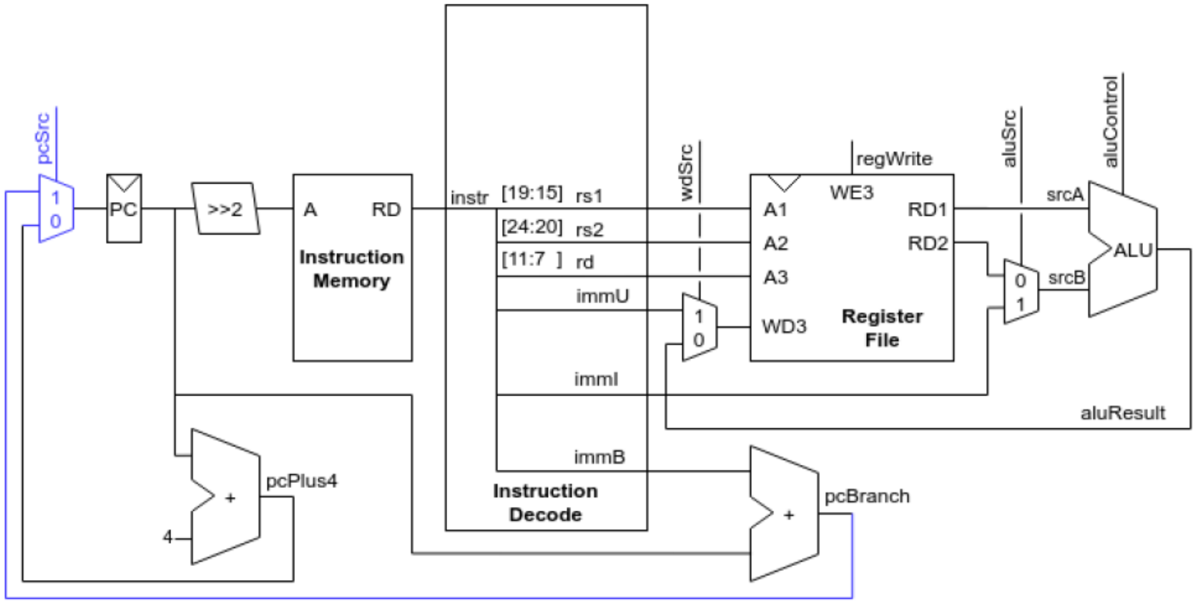


BEQ: **B-type**, branch on equal: **if (rs1 == rs2) PC = PC + immB**

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0	
imm	rs2	rs1	funct3	imm	opcode	**B-type**

31 30 – 20 19 – 12	11	10 – 5	4 – 1	0	
--- inst[31] ---	inst[7]	inst[30:25]	inst[11:8]	0	**B-immediate**

BEQ: address fetching

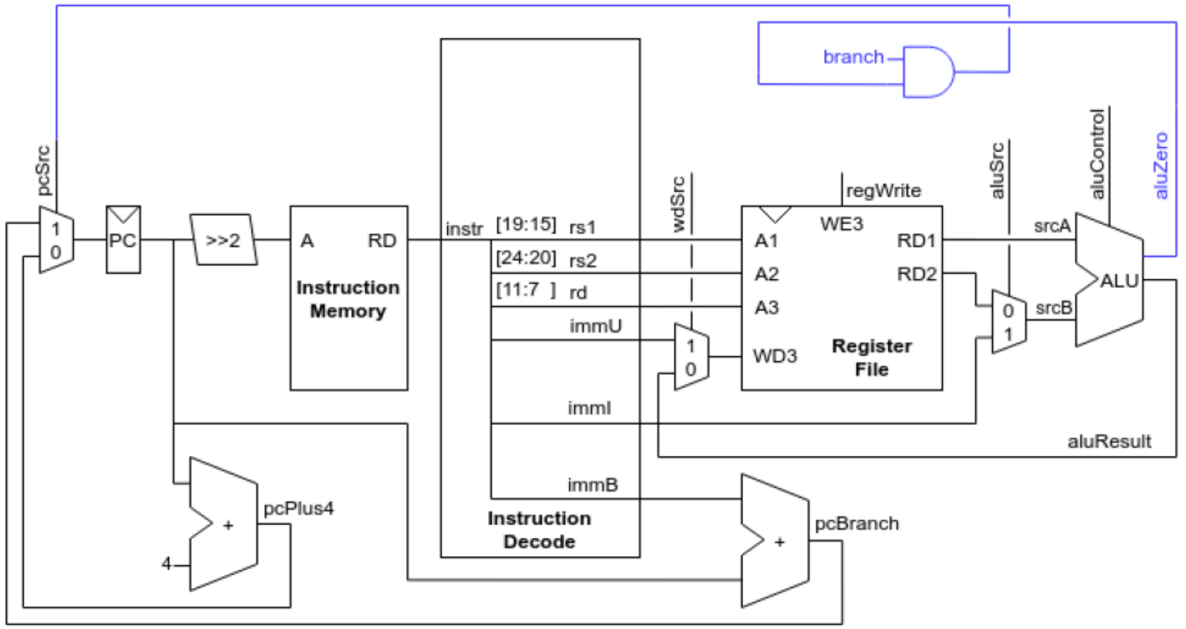


BEQ: **B-type**, branch on equal: **if (rs1 == rs2) PC = PC + immB**

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0	
imm	rs2	rs1	**funct3**	imm	**opcode**	**B-type**

31 30 – 20 19 – 12	11	10 – 5	4 – 1	0	
--- inst[31] ---	inst[7]	inst[30:25]	inst[11:8]	0	**B-immediate**

BEQ: check branch condition



BEQ: **B-type**, branch on equal: **if (rs1 == rs2) PC = PC + immB**

31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0	
imm	rs2	rs1	**funct3**	imm	**opcode**	**B-type**

31	30 — 20 19 — 12	11	10 — 5	4 — 1	0	
--- inst[31] ---		inst[7]	inst[30:25]	inst[11:8]	0	**B-immediate**

PC, adders and addresses multiplexors

```
// sm_register.v
module sm_register (
    input                  clk,
    input                  rst,
    input      [ 31 : 0 ] d,
    output reg [ 31 : 0 ] q
);
    always @ (posedge clk or negedge rst)
        if(~rst) q <= 32'b0;
        else     q <= d;
endmodule
```

```
// sr_cpu.v
    wire [31:0] pc;
    wire [31:0] pcBranch = pc + immB;
    wire [31:0] pcPlus4  = pc + 4;
    wire [31:0] pcNext   = pcSrc ? pcBranch : pcPlus4;
    sm_register r_pc(clk ,rst_n, pcNext, pc);
```

Instruction memory

```
// sm_rom.v
module sm_rom
#(
    parameter SIZE = 64
)
(
    input  [31:0] a,
    output [31:0] rd
);
    reg [31:0] rom [SIZE - 1:0];
    assign rd = rom [a];

    initial begin
        $readmemh ("program.hex", rom);
    end

endmodule
```

```
// sm_top.v
sm_rom reset_rom(imAddr, imData);
```

HDU-ITMO Joint Institute
杭州电子科技大学 圣光机联合学院

Instruction decoder (1)

```
// sr_cpu.v
module sr_decode
(
    input       [31:0] instr,
    output      [ 6:0] cmdOp,
    output      [ 4:0] rd,
    output      [ 2:0] cmdF3,
    output      [ 4:0] rs1,
    output      [ 4:0] rs2,
    output      [ 6:0] cmdF7,
    output reg [31:0] immI,
    output reg [31:0] immB,
    output reg [31:0] immU
);
    assign cmdOp = instr[ 6: 0];
    assign rd    = instr[11: 7];
    assign cmdF3 = instr[14:12];
    assign rs1   = instr[19:15];
    assign rs2   = instr[24:20];
    assign cmdF7 = instr[31:25];
```

Instruction decoder (2)

```
    // I-immediate
    always @ (*) begin
        immI[10: 0] = instr[30:20];
        immI[31:11] = { 21 {instr[31]} };
    end

    // B-immediate
    always @ (*) begin
        immB[     0] = 1'b0;
        immB[ 4: 1] = instr[11:8];
        immB[10: 5] = instr[30:25];
        immB[31:11] = { 21 {instr[31]} };
    end

    // U-immediate
    always @ (*) begin
        immU[11: 0] = 12'b0;
        immU[31:12] = instr[31:12];
    end
endmodule
```

HDU-ITMO Joint Institute
杭州电子科技大学 圣光机联合学院

Register file

```
// sr_cpu.v
module sm_register_file
(
    input           clk,
    input   [ 4:0] a0,
    input   [ 4:0] a1,
    input   [ 4:0] a2,
    input   [ 4:0] a3,
    output  [31:0] rd0,
    output  [31:0] rd1,
    output  [31:0] rd2,
    input   [31:0] wd3,
    input          we3
);
    reg [31:0] rf [31:0];

    assign rd0 = (a0 != 0) ? rf [a0] : 32'b0;
    assign rd1 = (a1 != 0) ? rf [a1] : 32'b0;
    assign rd2 = (a2 != 0) ? rf [a2] : 32'b0;

    always @ (posedge clk)
        if(we3) rf [a3] <= wd3;
endmodule
```

ALU operation

```
// sr_cpu.vh

`define ALU_ADD     3'b000  // A + B

`define ALU_OR      3'b001  // A | B

`define ALU_SRL     3'b010  // A >> B

`define ALU_SLTU    3'b011  // A < B ? 1 : 0

`define ALU_SUB     3'b100  // A - B
```

HDU-ITMO Joint Institute
杭州电子科技大学 圣光机联合学院

ALU implementation

```
// sr_cpu.v
module sr_alu
(
    input  [31:0] srcA,
    input  [31:0] srcB,
    input  [ 2:0] oper,
    output        zero,
    output reg [31:0] result
);
    always @ (*) begin
        case (oper)
            default   : result = srcA + srcB;
            `ALU_ADD  : result = srcA + srcB;
            `ALU_OR   : result = srcA | srcB;
            `ALU_SRL  : result = srcA >> srcB [4:0];
            `ALU_SLTU : result = (srcA < srcB) ? 1 : 0;
            `ALU_SUB : result = srcA - srcB;
        endcase
    end

    assign zero   = (result == 0);
endmodule
```

Data multiplexors

```verilog
// sr_cpu.v
wire [31:0] srcB = aluSrc ? immI : rd2;
```

```verilog
// sr_cpu.v
assign wd3 = wdSrc ? immU : aluResult;
```

Control unit (1)

```
// sr_cpu.v
module sr_control
(
    input       [ 6:0] cmdOp,
    input       [ 2:0] cmdF3,
    input       [ 6:0] cmdF7,
    input              aluZero,
    output             pcSrc,
    output reg         regWrite,
    output reg         aluSrc,
    output reg         wdSrc,
    output reg  [2:0]  aluControl
);
    reg            branch;
    reg            condZero;
    assign pcSrc = branch & (aluZero == condZero);
```

Control unit (2)

```
// sr_cpu.v
    always @ (*) begin
        branch      = 1'b0;
        condZero    = 1'b0;
        regWrite    = 1'b0;
        aluSrc      = 1'b0;
        wdSrc       = 1'b0;
        aluControl  = `ALU_ADD;

        casez( {cmdF7, cmdF3, cmdOp} )
            { `RVF7_ADD,  `RVF3_ADD,  `RVOP_ADD  } : begin regWrite = 1'b1; aluControl = `ALU_ADD;  end
            { `RVF7_OR,   `RVF3_OR,   `RVOP_OR   } : begin regWrite = 1'b1; aluControl = `ALU_OR;   end
            { `RVF7_SRL,  `RVF3_SRL,  `RVOP_SRL  } : begin regWrite = 1'b1; aluControl = `ALU_SRL;  end
            { `RVF7_SLTU, `RVF3_SLTU, `RVOP_SLTU } : begin regWrite = 1'b1; aluControl = `ALU_SLTU; end
            { `RVF7_SUB,  `RVF3_SUB,  `RVOP_SUB  } : begin regWrite = 1'b1; aluControl = `ALU_SUB;  end

            { `RVF7_ANY,  `RVF3_ADDI, `RVOP_ADDI } : begin regWrite = 1'b1; aluSrc = 1'b1; aluControl = `ALU_ADD; end
            { `RVF7_ANY,  `RVF3_ANY,  `RVOP_LUI  } : begin regWrite = 1'b1; wdSrc  = 1'b1; end

            { `RVF7_ANY,  `RVF3_BEQ,  `RVOP_BEQ  } : begin branch = 1'b1; condZero = 1'b1; aluControl = `ALU_SUB; end
            { `RVF7_ANY,  `RVF3_BNE,  `RVOP_BNE  } : begin branch = 1'b1; aluControl = `ALU_SUB; end
        endcase
    end
```