```
`timescale 1ns / 1ps
module data_path(clock,pc,reset,IR,op_code,Data1,Data2);
input clock,reset;
output [31:0] pc,IR,Data1,Data2;
output [6:0] op_code;

wire [31:0] pc_current;
reg [31:0] pc_next;
reg flag;
wire [2:0] alu_branch;
///////////////Register wires/////////////
wire [31:0] readata1,readata2,memtoreg;
////////// immediate values //////////
wire [31:0] ImmI,ImmB,ImmS;
wire [31:0] instr_reg;
wire [19:0] sign_extend;
reg  [31:0] imm_value;
wire [4:0] ramaddress,rom_address;
/////////////// Control unit wires/////////////////
wire wr_rd,r_rs1,r_rs2,alu_cont,branchop,jumpop,mem_read,mem_write,alu_im,aluop;
wire [1:0] imm_sel;
wire [31:0] Alu_out;
//////////////////ALU Mapping//////////////////////////
ALU Alu_dut
(.A(readata1),
.B(readata2),
.ALU_SEL(instr_reg[14:12]),
.OUT(Alu_out),
.imm(imm_value),
.alu_immed(alu_im),
.alu_op(aluop));
////////////////////////////RAM Mapping////////////////////
ram dut_ram(.address(ramaddress),
.clk(clock),
.write(mem_write),
.data_in(readata2),
.data_out(memtoreg),
.read(mem_read));
//////////////////////////// Rom Mapping  ////////////////////////////
rom rom_dut(.romaddress(rom_address),.instr_reg(instr_reg));
//////////////////////////// Sign extend///////////////
assign ImmB = {sign_extend,instr_reg[31:25],instr_reg[11:7]};
assign ImmI = {sign_extend,instr_reg[31:20]};
assign ImmS = {sign_extend,instr_reg[31:25],instr_reg[11:7]};
assign sign_extend = 20'd0;
////////////////////////////Address Reduction/////////////////////
```

```
redaddress dut1(.alu_address(Alu_out),.ram_address(ramaddress));
redaddress dut2(.alu_address(pc_current),.ram_address(rom_address));
////////////////////Register Files  Mapping ////////////
registerfile
regfile_dut(.clock(clock),
.reset(reset),
.read_rs1(instr_reg[19:15]),
.read_rs2(instr_reg[24:20]),
.read_rd(instr_reg[11:7]),
.write_data(memtoreg),
.w_reg(wr_rd),
.readsig1(r_rs1),
.readsig2(r_rs2),
.read_data1(readata1),
.read_data2(readata2));
////////////////////Control unit Mapping/////////////////////
controlunit dut(.ALU_OP(aluop),
.read_reg1(r_rs1),
.read_reg2(r_rs2),
.write_reg(wr_rd),
.branch(branchop),
.jump(jumpop),
.ALU_IMM(alu_im),
.sel_imm(imm_sel),
.write_data(mem_write),
.read_data(mem_read),
.opcode(instr_reg[6:0]));
////////////////////Mux IMM/////////////////////
always@(*)
begin
case(imm_sel)
2'b00: imm_value = 32'h0000;
2'b01: imm_value = ImmI;
2'b10: imm_value = ImmB;
2'b11: imm_value = ImmS;
default: imm_value = 32'hZZZZ;
endcase
end
////////////////// clock increment /////////////////
always@ (posedge clock or negedge reset)
begin
if(reset == 0)
pc_next <= 32'h0000;
else if( flag == 1)
pc_next <= pc_current + ImmB;
else if(pc_current == 32)
```

```
pc_next <= 32'h0000;
else
pc_next <= pc_current + 32'h0004;
end
assign pc_current = pc_next;

/////////////// Branch instruction ////////////////////
assign alu_branch = branchop?  instr_reg[14:12] : 3'bZZZ;
always@(*)
begin
if( alu_branch == 000 & ((readata1 == readata2)))
flag = 1'b1;
else if (alu_branch == 001 & ((readata1 > readata2)))
flag = 1'b1;
else if (alu_branch == 010 & ((readata1 < readata2)))
flag = 1'b1;
else if (alu_branch == 011 & ((readata1 != readata2)))
flag = 1'b1;
else
flag = 1'b0;
end
/////////////// Branch instruction ////////////////////

assign pc = pc_current;
assign op_code = instr_reg[6:0];
assign IR = instr_reg;
assign Data1 = readata1;
assign Data2 = readata2;
endmodule
```