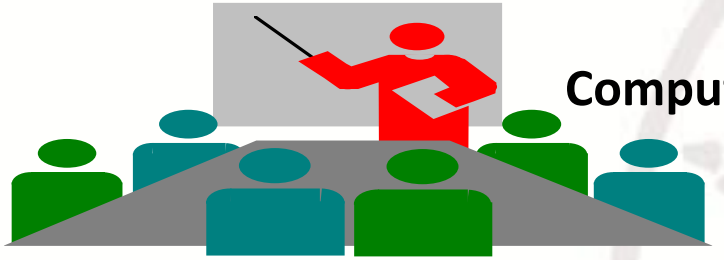


Computer Organization & Design
实验与课程设计

实验十

硬件阻塞流水线处理器

--流水线数据相关性检测与冒险竞争处理之阻塞流水线

施青松
Asso. Prof. Shi Qingsong
College of Computer Science and Technology, Zhejiang University
zjsqs@zju.edu.cn

Course Outline

- 实验目的与实验环境
- 实验任务
- 实验原理
- 实验操作与实现

实验目的

1. **深入理解流水CPU结构**
   - 只能运行固定状态机事件的寄存器传输结构
   - 一种同步控制
2. **深入理解流水优化性能的本质**
   - 流水技术降低了单条指令的执行性能
   - 流水线改善提高了指令执行的吞吐率
3. **深入理解相关性(数据和控制)与冒险竞争的本质**
   - 程序相关性与硬件冒险竞争
4. **学习流水部件同步控制技术**

实验环境

实验设备

1. 计算机（Intel Core i5以上，4GB内存以上）系统

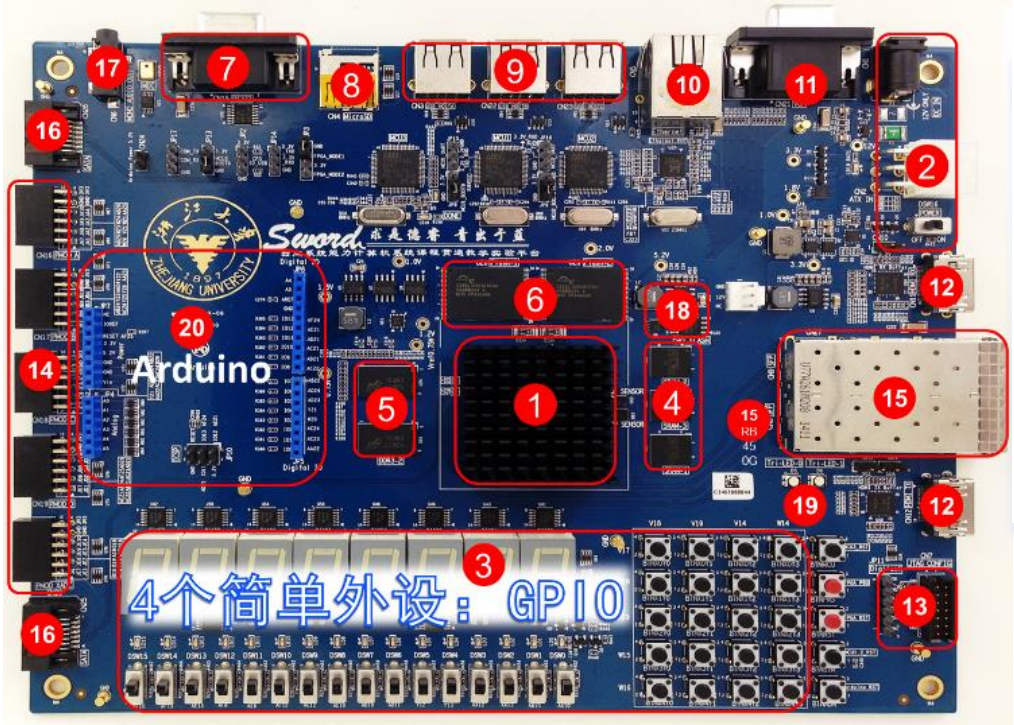
2. 计算机软硬件课程贯通教学实验系统(Sword)

3. 3. Vivado 2017.4或Xilinx ISE14.4及以上开发工具

材料

无

计算机软硬件课程贯通教学实验系统



贯通教学实验平台主要参数

▼ **核心芯片**

Xilinx Kintex™-7系列的XC7K160/325资源：

162,240个，Slice：25350，片内存储：11.7Mb

▼ **存储体系** 支持32位存储层次体系结构

6MB SRAM静态存储器：支持32Data，16位TAG

512M BDDR3动态存储：支持32Data

32MB NOR Flash存储 ：支持32位Data

▼ **基本接口** 支持微机原理、SOC或微处理器简单应用

4×5+1矩阵按键、16位滑动开关、16位LED、8位七段数码管

▼ **标准接口** 支持基本计算机系统实现

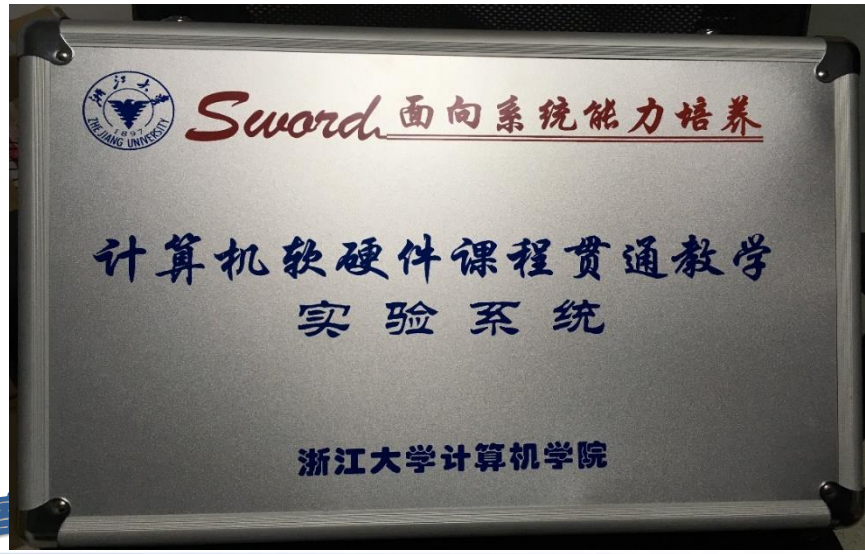
12位VGA接口（RGB656）、USB-HID（键盘）

▼ **通讯接口** 支持数据传输、调试和网络

UART接口、 10M/100M/1000M以太网、 SFP光纤接口

▼ **扩展接口** 支持外存、多媒体和个性化设备

MicroSD(TF）、PMOD、 HDMI、Arduino



Course Outline

- 实验目的与实验环境
- 实验任务
- 实验原理
- 实验操作与实现

实验任务

1. 设计相关性检测电路

- 数据通路之数据相关性检测
- 数据通路之控制相关性检测

2. 设计硬件阻塞流水线电路

- 阻塞流水线设计(Stall)：部分流水线部件等待部分继续
- 冲刷流水线设计(Flush)：清除无效指令

3. 测试硬件阻塞电路

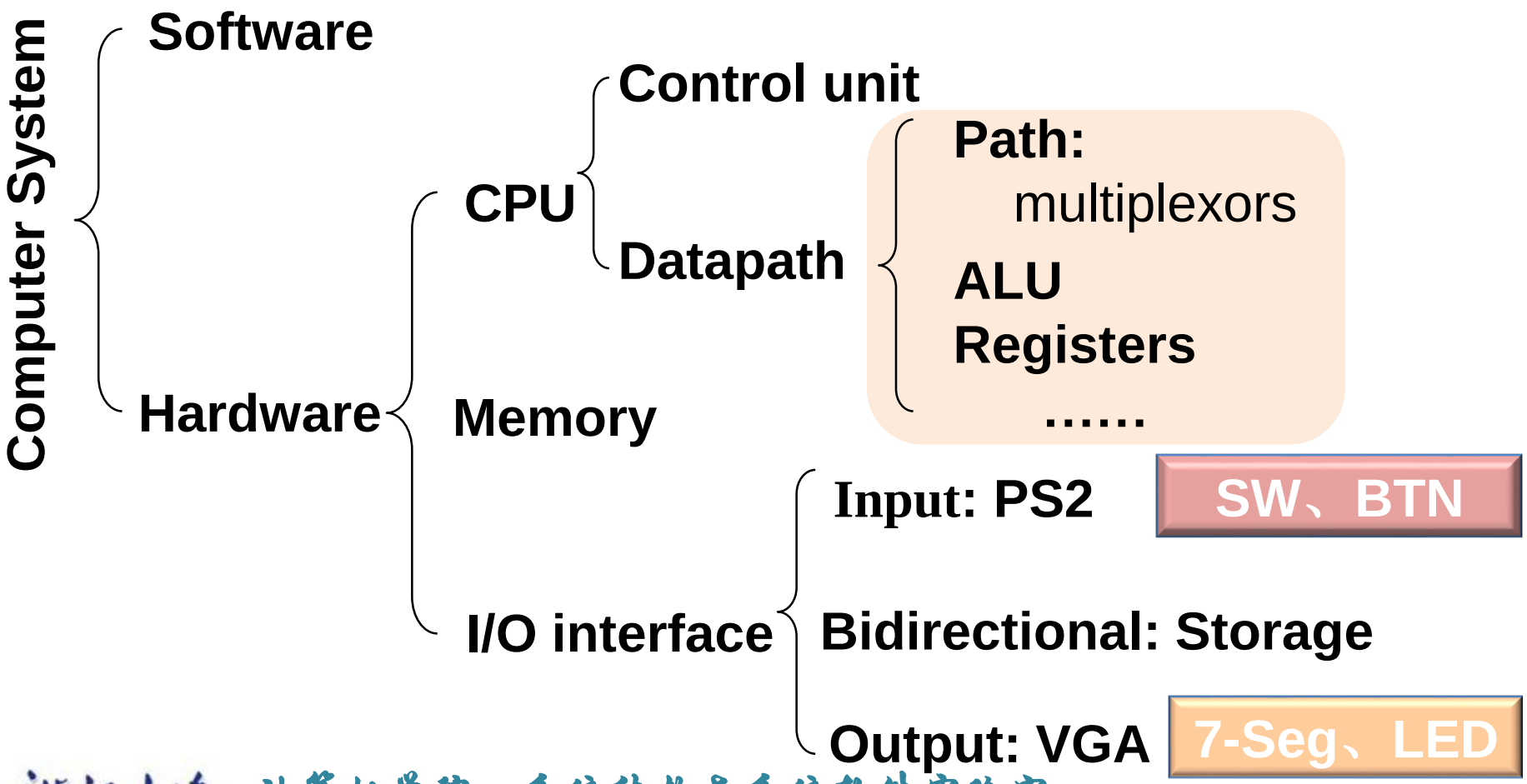
- 设计测试程序遍历测试所有可能的相关性
- 此项需要完备性测试

Course Outline

- 实验目的与实验环境
- 实验任务
- 实验原理
- 实验操作与实现

Computer Organization

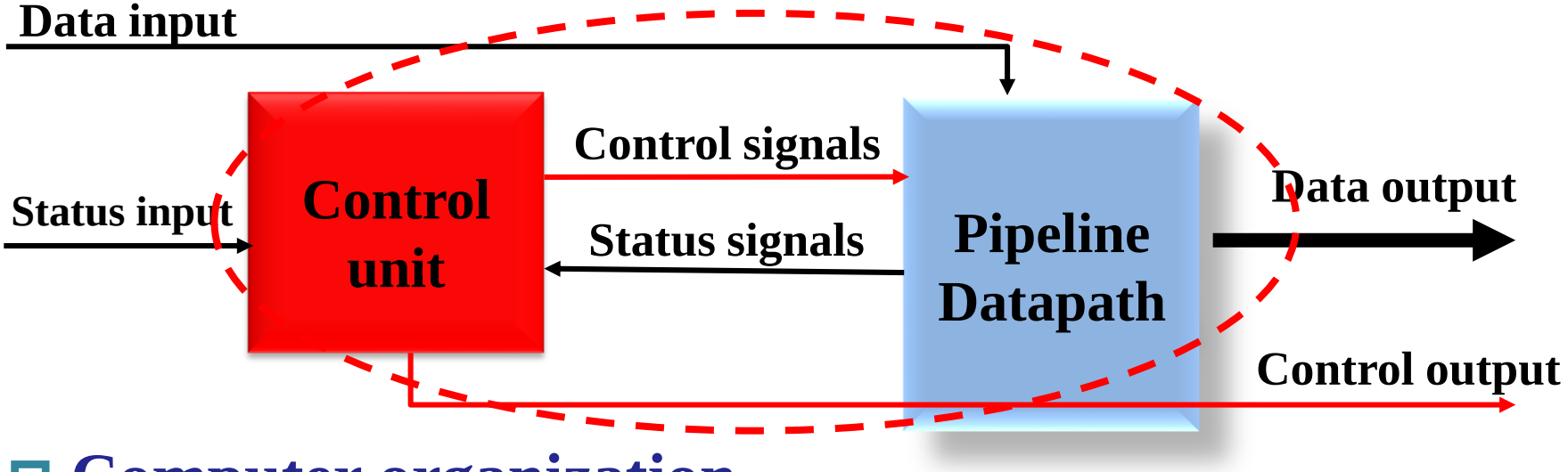
- Decomposability of computer systems



CPU organization

Digital circuit

- General circuits that controls logical event with logical gates --Hardware



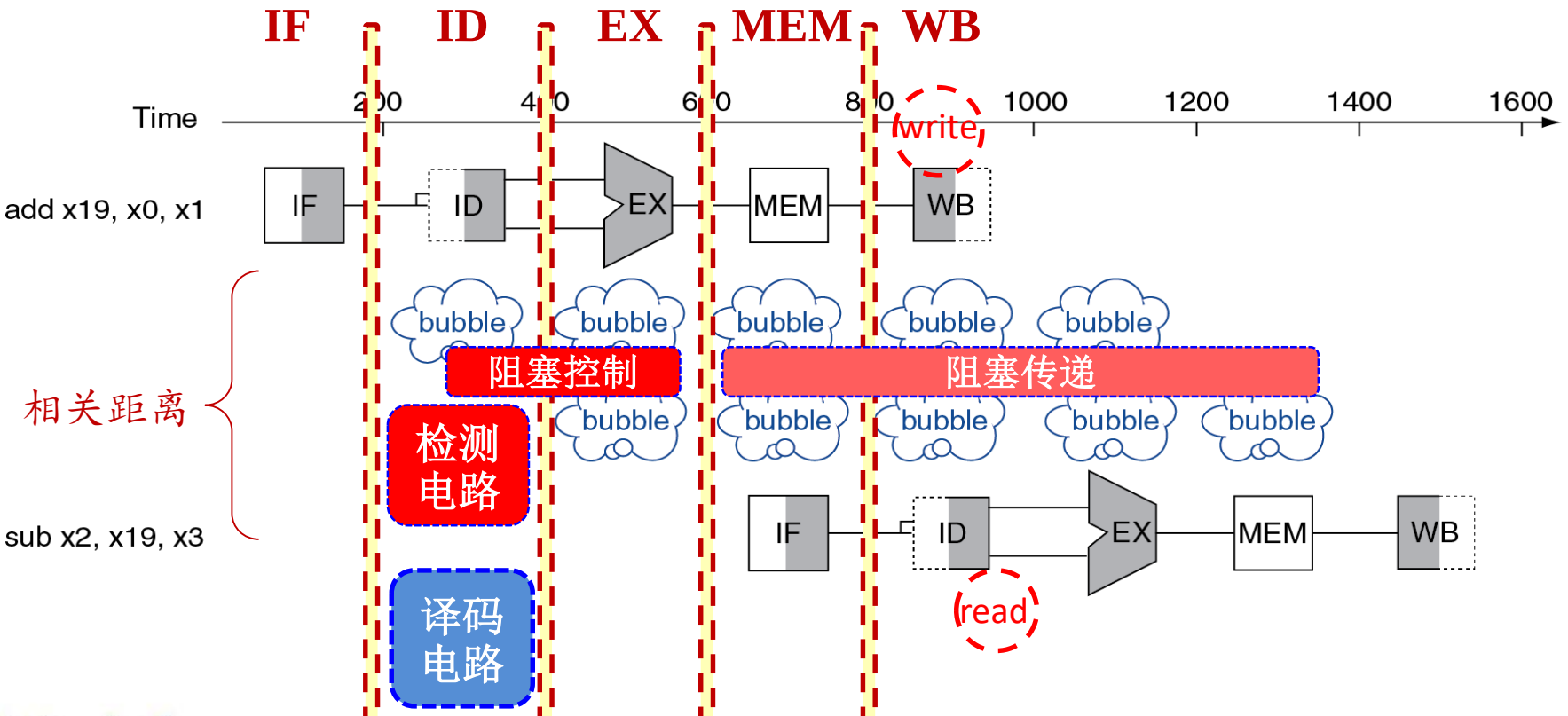
Computer organization

- Special circuits that processes logical action with instructions -Software

相关性检测位置

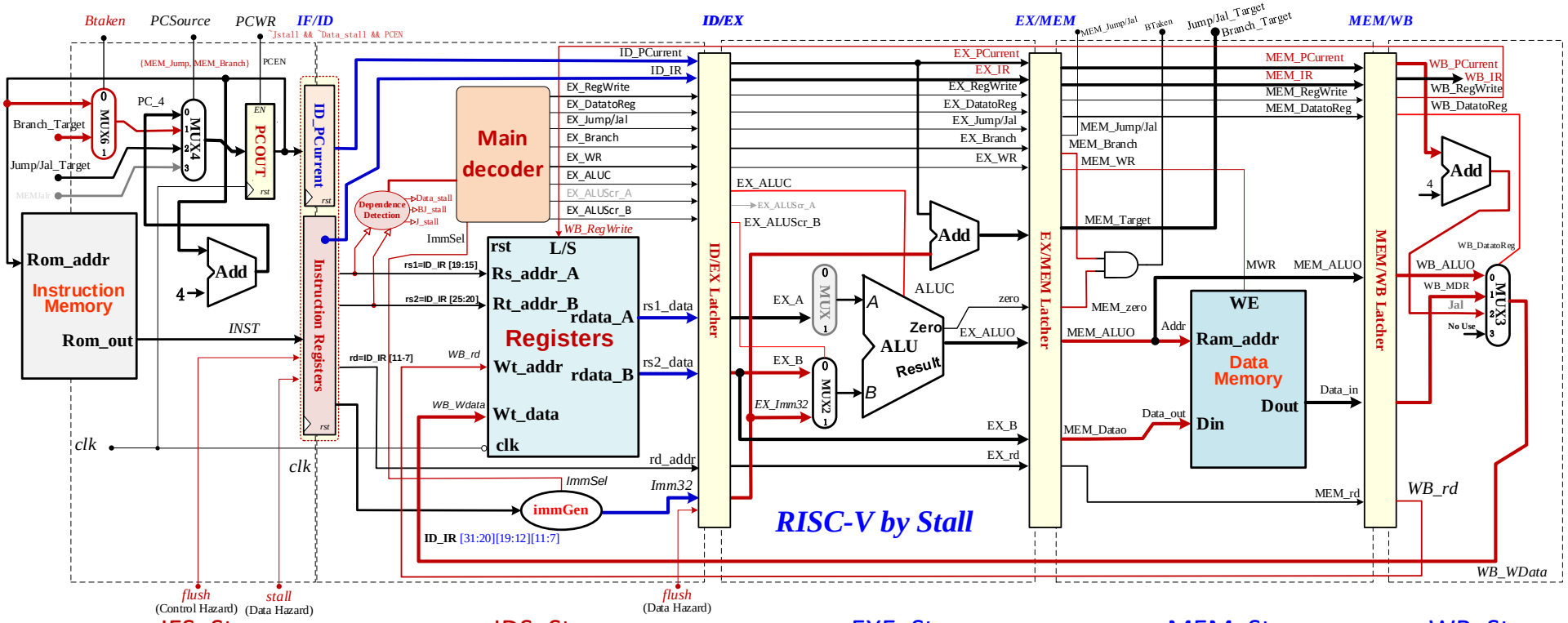
此处以实验九为例

- 定位本质：尽早检测，清除方便
  - 当前指令ID级检测相关性
    - 此时指令还没有破坏CPU状态，便于消除影响



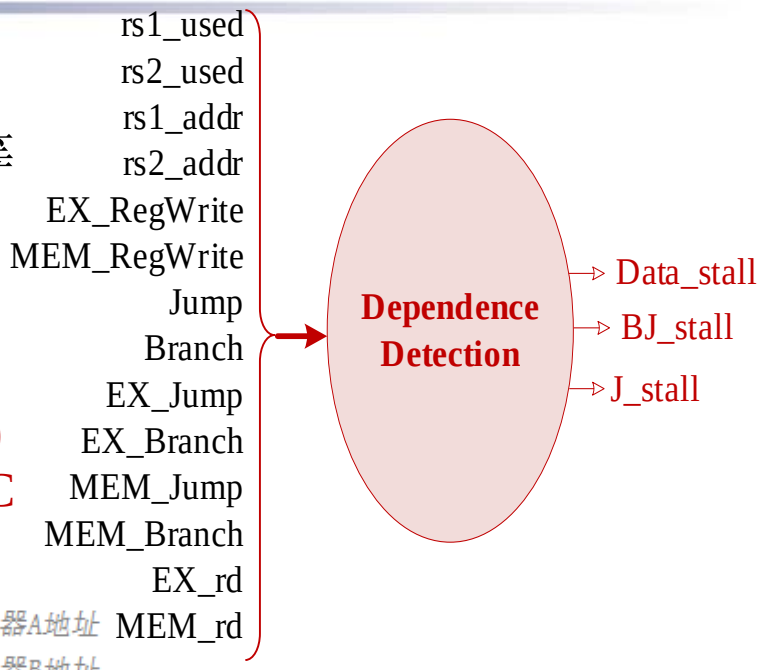
数据相关性检测与阻塞流水线

- 基于实验九增加相关性检测
  - ID增加检测电路Dependence Detection
    - Data_stall
    - BJ_stall
    - J_stall
  - IF级阻塞流水线，ID级冲刷流水线： *flush*、*Stall*



相关性检测： Dependence Detection

- 数据相关
  - ID级rs1和rs2与EX和MEM级rd相等
    - Data Stall = 1： 阻塞1~2clk 插入1~2nop(bubble)
- 控制相关
  - ID级指令译码为Control类指令
    - BJ Stall = 1：清除3条指令 (bubble/clk)
    - J Stall = 1： 阻塞2clk，MEM级使能PC




```
/*Hazards Detection by Data dependence: 相关性检测*/
//    wire[4:0] rs1_addr = ID_IR[19:15];                  //ID级读寄存器A地址
//    wire[4:0] rs2_addr = ID_IR[24:20];                  //ID级读寄存器B地址
    wire Hazards = (EX_RegWrite && EX_rd !=0 ||          //前续1指令写且不为0
                    MEM_RegWrite && MEM_rd !=0 );        //前续2指令写且不为0
    assign Data_stall =(rs1_used  &&  rs1_addr != 0 && Hazards &&      //rs1使用且不为"0", 前续rd存在写
                        (rs1_addr == EX_rd || rs1_addr == MEM_rd)) ||  //存在数据冲突
                        (rs2_used  &&  rs2_addr != 0 && Hazards &&     //rs2使用且不为"0", 前续rd存在写
                        (rs2_addr == EX_rd || rs2_addr == MEM_rd));    //存在数据冲突
/*Hazards Detection by Branch Dependence*/
    assign BJ_stall = Branch || Jump || EX_Branch || EX_Jump || MEM_Branch || MEM_Jump;    //前续指令为转移指令
    assign Jstall =   Branch || Jump || EX_Branch || EX_Jump;  //无论taken or not MEM级时必须使能PC
```

冒险清除之一：IF 级禁止取指

- 阻塞流水线
  - 禁止PC，停止取指
    - 清除控制相关指令
    - 阻塞数据相关指令

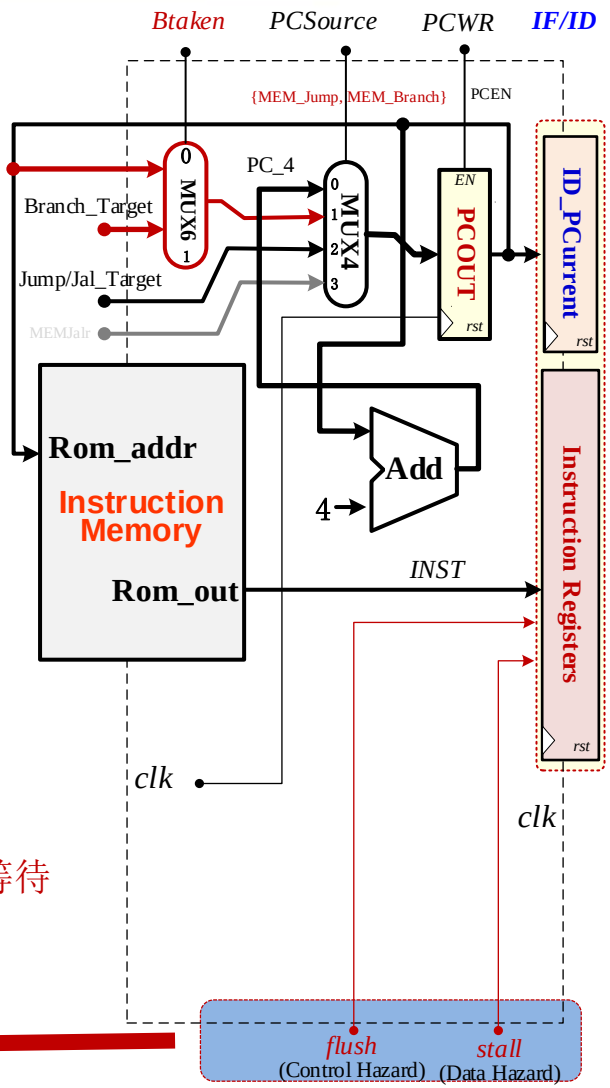
```
/*IF stage-------------------------------------------------------------*/
reg[31:0] PCNEXT;
    wire[31:0] PC_4      = PCOUT + 4;                  //顺序执行PC程序地址
    wire[31:0] PC_JUMP   = MEM_Target;                 //jump伪指令目标地址
    wire[31:0] PC_Branch = Btaken ? MEM_Target : PCOUT; //Beq指令目标地址：Btaken = MEM_zero && MEM_Branch
    assign  PCSource = {MEM_Jump, MEM_Branch};         //下一指令地址选择，taken or not：PCSource = 01
    wire PCWR = ~Jstall && ~Data_stall && PCEN;        //Data & Branch Hazards 暂停取指

    always @* begin
        case(PCSource)                                 //MUX4/5
            2'b00: PCNEXT = PC_4;                      //Sequential address
            2'b01: PCNEXT = PC_Branch;                 //Branch Target address
            2'b10: PCNEXT = PC_JUMP;                   //jal Target address
            2'b11: PCNEXT = PC_4;                      //No Use
        endcase
    end
    REG32      IMPC(.clk(clk),                         //PC指针修改
                    .rst(rst),
                    .CE(PCWR),
                    .D(PCNEXT),
                    .Q(PCOUT));                        //当前PC address，在时钟上升沿更新
```

禁止PC

阻塞相关指令等待

```
//IF/ID Latch   ID级锁存变量：ID_IR, ID_PCurrent
    REG_IF_ID  IFID(.clk(clk),.rst(rst), .EN(1'b1),.Data_stall(Data_stall), //锁存输入
                    .flush(flush), .PCOUT(PCOUT),    .IR(inst_field),
                    .ID_IR(ID_IR), .ID_PCurrent(ID_PCurrent) );          //锁存输出
```

冲刷流水线



IF/ID 流水锁存寄存器冲刷和阻塞等待

```
module    REG_IF_ID(input clk,                                    //IF/ID Latch
                    input rst,
                    input EN,                                     //流水寄存器使能
                    input Data_stall,                             //数据竞争禁止PC更新
                    input flush,                                  //控制竞争Flush清除3条指令
                    input [31:0] PCOUT,                           //指令存储器指针
                    input [31:0] IR,                              //指令存储器输出
                    output reg[31:0] ID_IR,                       //取指锁存
                    output reg[31:0] ID_PCurrent  );              //当前存在指令地址

//reg[31:0]ID_PCurrent,ID_IR;
    always @(posedge clk) begin
    if(rst) begin
         ID_IR <= 32'h00000000;                          //复位清零
         ID_PCurrent <= 32'h00000000;                    //复位清零
         end
    else if(EN)begin
          if(Data_stall)begin                            //优先处理数据冒险竞争：此处仅禁止PC更新
              ID_IR <= ID_IR;                            //IR waiting for Data Hazards 暂停取指
              ID_PCurrent <= ID_PCurrent;   end          //保存对应PC指针
          else if(flush)begin                            //若Branch flusch 3条指令
                ID_IR <= 32'h00002003;                   //IR waiting for Control Hazards 清除指令并暂停取指
                ID_PCurrent <= ID_PCurrent;   end        //保存清除指令的指针(测试)
               else begin
                ID_IR <= IR;                             //正常取指,传送下一流水级译码
                ID_PCurrent <= PCOUT;   end              //当前取指PC地址，Branch/Junp指令计算目标地址用(非PC+4)
        end
        else begin
            ID_IR <= ID_IR;
            ID_PCurrent <= ID_PCurrent;
        end
end
endmodule
```

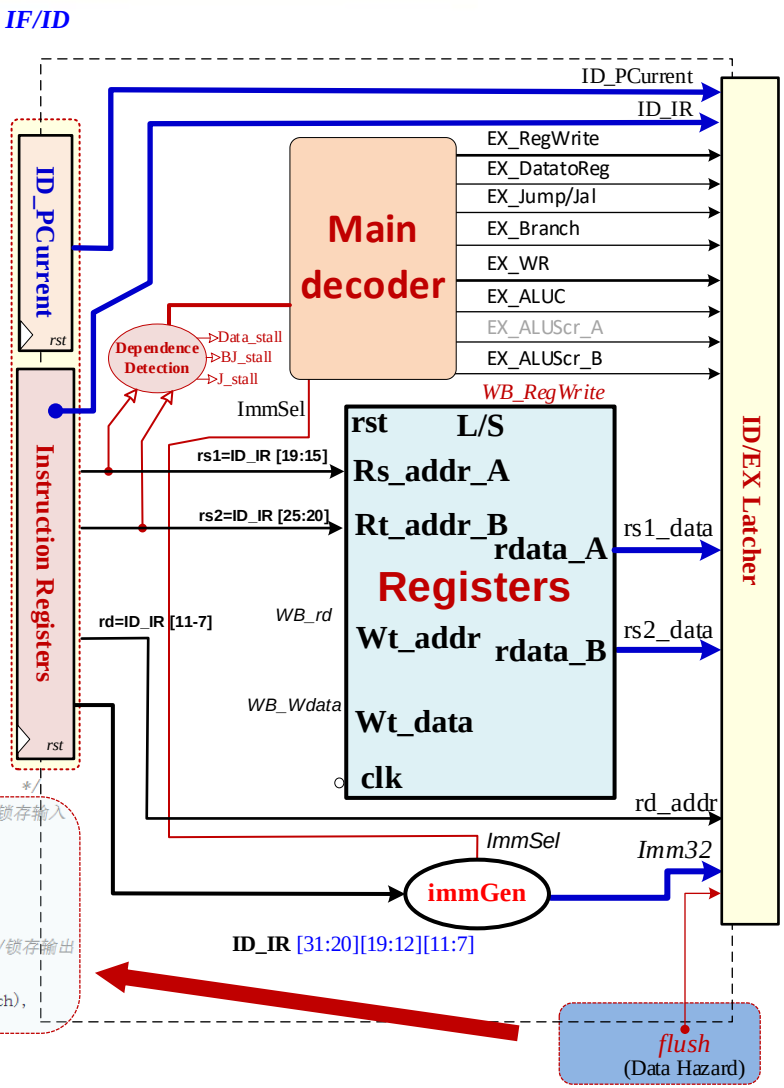
数据相关：阻塞等待

控制相关插入nop：
lw zero, 0(zero)

冒险清除之二：ID 级冲刷流水线

```
module  IDS_Stage(input clk,
                  input rst,
                  input flush, //若指令数据竞争则插入两个气泡  IF已禁止PC更新
                  output[31:0]rs2_data,                  //rs2寄存器输出
                  output[31:0]Imm32,                     ////ID级读出并扩展后立即数
                  input [4:0]Debug_addr,                 //测试定位
                  output [31:0]Debug_regs                //测试、调试信号
                  );

    wire[4:0] rs1_addr = ID_IR[19:15];          //ID级读寄存器A地址
    wire[4:0] rs2_addr = ID_IR[24:20];          //ID级读寄存器B地址
    wire[4:0] rd_addr  = ID_IR[11:7];           //ID级写寄存器A地址，需传输至WB级

    Regs    DU2(.clk(clk),
                .rst(rst),
                .L_S(WB_RegWrite),              //WB级写寄存器使能
                .R_addr_A(rs1_addr),            //ID级读寄存器A地址
                .R_addr_B(rs2_addr),            //ID级读寄存器B地址
                .rdata_A(rs1_data),             //ID寄存器A读出数据
                .rdata_B(rs2_data),             //ID寄存器B读出数据
                .Wt_addr(WB_rd),                //WB级写寄存器地址
                .Wt_data(Wt_data),                //WB级写数据
                .Debug_addr(Debug_addr[4:0]),   // debug address
                .Debug_regs(Debug_regs)         // debug data
                );

    MUX4T1_32   MUX0(.s(ImmSel),                //ID级立即数读取：ImmGen                                    //addi\lw(I): I-Type
                     .I0({{20{ID_IR[31]}},ID_IR[31:20]}),                                                   //sw(S)     : S-Type
                     .I1({{20{ID_IR[31]}},ID_IR[31:25],ID_IR[11:7]}),                                       //beq(B)    : B-Type
                     .I2({{19{ID_IR[31]}},ID_IR[31],ID_IR[7],ID_IR[30:25],ID_IR[11:8],1'b0}),               //jal(J)    : J-Type
                     .I3({{11{ID_IR[31]}},ID_IR[31],ID_IR[19:12],ID_IR[20],ID_IR[30:21],1'b0}),
                     .o(Imm32));                //another imm: U_imm = {inst [31:12], 12'h0};  U-Type

/*ID/EX Latch   EX级锁存变量：EX_IR, EX_PCurrent, EX_A, EX_B, EX_Imm32, EX_rd, EX_ALUSrc_A, EX_ALUSrc_B, EX_ALUC,
                EX_DatatoReg, EX_RegWrite, EX_Jump, EX_Branch, EX_WR, EX_Sign                          */
    REG_ID_EX   IDEX(.clk(clk), .rst(rst), .EN(1'b1), .flush(flush), .ID_IR(ID_IR), .ID_PCurrent(ID_PCurrent), //锁存输入
                     .rs1_data(rs1_data),  .rs2_data(rs2_data), .Imm32(Imm32), .rd_addr(rd_addr),
                     .ALUSrc_A(ALUSrc_A), .ALUSrc_B(ALUSrc_B), .ALUC(ALUC), .DatatoReg(DatatoReg),
                     .RegWrite(RegWrite), .Jump(Jump), .Branch(Branch), .WR(WR), .MIO(MIO), .Sign(Sign),

                     .EX_PCurrent(EX_PCurrent), .EX_IR(EX_IR), .EX_A(EX_A), .EX_B(EX_B), .EX_Imm32(EX_Imm32),     //锁存输出
                     .EX_rd(EX_rd), .EX_ALUSrc_A(EX_ALUSrc_A), .EX_ALUSrc_B(EX_ALUSrc_B), .EX_ALUC(EX_ALUC),
                     .EX_DatatoReg(EX_DatatoReg), .EX_RegWrite(EX_RegWrite), .EX_Jump(EX_Jump), .EX_Branch(EX_Branch),
                     .EX_WR(EX_WR), .EX_MIO(EX_MIO), .EX_Sign(EX_Sign));
endmodule
```

数据竞争冲刷流水线 Data_stall



IDS_Stage

ID/EXE 流水锁存寄存器 冲刷流水线

```
module    REG_ID_EX(input clk,                              //ID/EX Latch
                    input rst,
                    input EN,                               //流水寄存器使能
//                  input flush,    //数据竞争清除指令插入2个气泡：Data_stall
                    input flush,                            //数据竞争清除指令插入2个气泡：Data_s
                    input [31:0] ID_IR,                     //当前译码指令(测试)
                    input [31:0] ID_PCurrent,               //当前译码指令存储器指针
                    input [31:0] rs1_data,                  //当前指令读出寄存器A数据
                    input [31:0] rs2_data,                  //当前指令读出寄存器A数据
                    input [31:0] Imm32,                     //当前指令读出并扩展32位立即数据
                    input [4:0]  rd_addr,                   //当前指令读出目的操作数地址
                    input ALUSrc_A,                       //当前指令译码：ALU A通道控制
                    input ALUSrc_B,                       //当前指令译码：ALU B通道控制
                    input [2:0]  ALUC,                      //当前指令译码：ALU操作控制
                    input [1:0]  DatatoReg,                 //当前指令译码：REG写数据通道选择
                    input RegWrite,                         //当前指令译码：寄存器写信号
                    input Jump,                             //当前指令译码：UJ跳转控制
                    input Branch,                           //当前指令译码：SB跳转控制
                    input WR,                               //当前指令译码：存储器读写信号
                    input MIO,
                    input Sign,                             //当前指令译码：符号标志(保留)
                    output reg[31:0] EX_PCurrent,           //锁存当前译码指令地址
                    output reg[31:0] EX_IR,                 //锁存当前译码指令(测试)
                    output reg[31:0] EX_A,                  //锁存当前译码指令读出寄存器A数据
                    output reg[31:0] EX_B,                  //锁存当前译码指令读出寄存器B数据
                    output reg[31:0] EX_Imm32,              //锁存当前译码指令32位立即数据
                    output reg[4:0]  EX_rd,                 //锁存当前译码指令写目的寄存器地址
                    output reg       EX_ALUSrc_A,           //锁存当前译码指令ALU A通道控制
                    output reg       EX_ALUSrc_B,           //锁存当前译码指令ALU B通道控制(保留)
                    output reg[2:0]  EX_ALUC,               //锁存当前译码指令ALU操作功能控制
                    output reg[1:0]  EX_DatatoReg,          //锁存当前译码指令REG写数据通道选择
                    output reg       EX_RegWrite,           //锁存当前译码指令寄存器写信号
                    output reg       EX_Jump,               //锁存当前译码指令UJ跳转控制
                    output reg       EX_Branch,             //锁存当前译码指令SB跳转控制
                    output reg       EX_WR,                 //锁存当前译码指令存储器读写信号
                    output reg       EX_MIO,
                    output reg       EX_Sign                //锁存当前译码指令符号标志（保留）
                    );
  always @(posedge clk ) begin                      //ID/EX Latch
  if(rst) begin
      EX_rd        <= 0;
      EX_RegWrite  <= 0;
      EX_Jump      <= 0;
      EX_Branch    <= 0;
      EX_WR        <= 0;
      EX_IR        <= 32'h00000000;
      EX_PCurrent  <= 32'h00000000 ;
   end
   else if(EN)begin
      if(flush)begin                                //数据冲突时冲刷流水线禁止改变CPU状态
          EX_IR        <= 32'h00000000;             //nop,废弃当前取指 : Stall插入32'h00000000区别编译
          EX_rd        <= 0;                        //cancel Instruction write address
          EX_RegWrite  <= 0;                        //寄存器写信号：禁止寄存器写
          EX_Jump      <= 0;                        //cancel JUMP instruction
          EX_Branch    <= 0;                        //cancel Branch instruction
          EX_WR        <= 0;                        //cancel write memory
          EX_MIO       <= MIO;                      //清除存储IO访问标志
          EX_PCurrent  <= ID_PCurrent;              //传递PC(测试)
      end
      else begin                                    //无数据冲突正常传输到EX级
          EX_PCurrent  <= ID_PCurrent;              //传递当前指令地址
          EX_IR        <= ID_IR;                    //传递当前指令地址(测试)
          EX_A         <= rs1_data;                 //传递寄存器A读出数据
          EX_B         <= rs2_data;                 //传递寄存器B读出数据
          EX_Imm32     <= Imm32;                    //传递扩展后立即数
          EX_rd        <= rd_addr;                  //传递写目的寄存器地址
          EX_ALUSrc_A  <= ALUSrc_A;                 //传递ALU A通道控制信号
          EX_ALUSrc_B  <= ALUSrc_B;                 //传递ALU B通道控制信号
          EX_ALUC      <= ALUC;                     //传递ALU操作功能控制信号
          EX_DatatoReg <= DatatoReg;                //传递REG写数据通道选择
          EX_Jump      <= Jump;                     //传递UJ跳转控制信号
          EX_Branch    <= Branch;                   //传递SB跳转控制信号
          EX_RegWrite  <= RegWrite;                 //传递寄存器写信号
          EX_WR        <= WR;                       //传递存储器读写信号
          EX_MIO       <= MIO;                      //存储IO访问标志
          EX_Sign      <= Sign;                     //传递符号标志（保留）
          end
      end
      //else 不变                                   //锁存器禁止
  end

endmodule
```

硬件插入nop：HDbubble

流水控制器译码参考：方案不唯一

```
reg [1:0] ALUop;
wire[3:0] Fun;

    assign ALE = ~clk;
    assign PCEN = 1;

    always @* begin
        ALUSrc_A  =0;
        ALUSrc_B  =0;
        DatatoReg =0;
        RegWrite  =0;
        Branch    =0;
        Jump      =0;
        WR        =0;
        CPU_MIO   =0;
        ALUop     =2'b10;
        rs1_used  = 0;
        rs2_used  = 0;

      case(OPcode)
        5'b01100: begin ALUop=2'b10;RegWrite=1;                ALUSrc_B=0;Branch=0;Jump=0;DatatoReg=2'b00;      //ALU(R)
                        rs1_used = 1; rs2_used = 1;                                                          end
        5'b00000: begin ALUop=2'b00;RegWrite=1;ImmSel=00;ALUSrc_B=1;Branch=0;Jump=0;DatatoReg=2'b01;      //load
                        rs1_used = 1;                                                WR=0;CPU_MIO = 1; end
        5'b01000: begin ALUop=2'b00;RegWrite=0;ImmSel=01;ALUSrc_B=1;Branch=0;Jump=0;WR=1;CPU_MIO = 1;     //store
                        rs1_used = 1; rs2_used = 1;                                                          end
        5'b11000: begin ALUop=2'b01;RegWrite=0;ImmSel=10;ALUSrc_B=0;Branch=1;Jump=0;                      //beq
                        rs1_used = 1; rs2_used = 1;                                                          end
        5'b11011: begin              RegWrite=1;ImmSel=11;                          Jump=1;DatatoReg=2'b10; end //jump
        5'b00100: begin ALUop=2'b11;RegWrite=1;ImmSel=00;ALUSrc_B=1;Branch=0;Jump=0;DatatoReg=2'b00;      //ALU(I)
                        rs1_used = 1;                                                                        end
        default:         ALUop=2'b00;
      endcase
    end
```

```
    assign Fun = {Fun3,Fun7};
    always @* begin
        case(ALUop)
            2'b00: ALUC=3'b010;                 //load/store
            2'b01: ALUC =3'b110;                //sub: beq
            2'b10:
                case(Fun)
                    4'b0000: ALUC =3'b010;    //add
                    4'b0001: ALUC =3'b110;    //sub
                    4'b1110: ALUC =3'b000;    //and
                    4'b1100: ALUC =3'b001;    //or
                    4'b0100: ALUC =3'b111;    //slt
                    4'b1010: ALUC =3'b101;    //srl
                    4'b1000: ALUC =3'b011;    //xor
                    default: ALUC =3'bx;
                endcase
            2'b11:
                case(Fun3)
                    3'b000: ALUC =3'b010;         //addi
                    3'b111: ALUC =3'b000;         //andi
                    3'b110: ALUC =3'b001;         //ori
                    3'b010: ALUC =3'b111;         //slti
                    3'b101: ALUC =3'b101;         //srli
                    3'b100: ALUC =3'b011;         //xori
                    default:   ALUC =3'bx;
                endcase
            default:   ALUC =3'bx;
        endcase
    end

endmodule
```

实验六或七插入源操作数使用标志

功能测试程序

- Exp07的功能测试程度手工调度
  - 本实验仅建立流水结构没有处理竞争
- 请将实验7测试程序手工插入nop
  - 数据相关插入2个nop
  - 转移指令后手入3个nop

详细参考pdf：

实验10：硬件stall去掉数据相关nop

实验11：硬件flush去掉控制相关nop

本实验硬件插入气泡，恢复实验六或实验七功能测试程序：

软件插入 nop(32'h00000013) = addi zero, zero, 0

数据相关硬件插入气泡(32'h00000000) = flush zero

控制相关硬件插入气泡(32'h00002003) = lw zero, 0(zero)

实验四测试程序优化片段

```
loop:
    lw a1, 0x0(s1)          # 读GPIO端口F0000000状态: {out0, out1, out2, D28-D21, BTN3-BTN0, SW15-SW0}
    nop
    nop
    add a1, a1, a1
    nop
    nop
    add a1, a1, a1
    nop
    nop
    sw a1, 0x0(s1)          # x11输出到GPIO端口F0000000，计数器通道counter_set=00端口不变、LED=SW:  {GPIOf0
    lw a1, 0x0(s1)          # 再读GPIO端口F0000000状态
    nop                     Data 2 nop
    nop
    and s8, a1, t0          # 取最高位=out0，屏蔽其余位送x14
    add s6, s6, t1          # 程序计数延时
    nop
    nop
    #beq    s8, t0, C_init  # 若硬件计数启用：C0=0，Counter通道0溢出，转计数器初始化，修改7段码显示
    beq s6, zero, C_init    # 程序计数x22=0，转计数器初始化，修改7段码显示：C_init
    nop
    nop                     Branch 3 nop
    nop
```

Course Outline

- 实验目的与实验环境
- 实验任务
- 实验原理
- 实验操作与实现

设计工程：OExp10-RCSTE-RV32IPS

- **基于实验九数据通路扩展阻塞流水线功能**
  - 设计相关性检测电路
  - 修改IF、ID级增加阻塞与消除功能
- **调试、测试和应用环境**
  - 顶层用HDL实现
    - 模块名： CSTE_RV32IP_Stall
  - 测试程序： RISCV-DEMO9.coe

设计要点

注意：本实验在Exp09基础上增加检测、阻塞和清除，极为简单

- 设计相关性检测电路
  - 注意检测的边界条件
    - 本实验工作量很少，但知识点(原理)很重要
    - 注意信号有时序区别
- 设计流水线阻塞和冲刷清除电路
  - 以实验六指令仅需修改IF Stage和 ID Stage
    - 取指阻塞：封锁PC停止取指
    - 数据相关冒险阻塞：采用状态等待
    - 数据冒险清除：插入气泡(32h'00000000/清零，也可用其他区分复位)
    - 控制冒险清除：插入气泡(32h' 00002003/lw zero, 0(zero))
- 流水级命名(实现方法一用)
  - IFS_Stage、IDS_Stage、EXE_Stage、MEM_Stage、WB_Stage
  - 锁存器命名
    - IF/ID： REGS_IF2ID， 对应锁存变量不变
    - ID/EX：REGS_ID2EX，对应锁存变量不变

设计要点：功能测试代码. coe

- **32位指令存储器：**
  - SWORD实验平台 ROM用Distributed Memory
- **ROM初始化文件(RISCV-DEMO9.coe)**
  - 功能测试程序，其功能与实验四-六完全相同
    - 相关性非常强，用于测试流水竞争非常适用

```
memory_initialization_radix=16;
memory_initialization_vector=
0200006F,00000033,00000033,00000033,00000033,00000033,00000033,00000033,00C02283,00502333,
006303B3,00638E33,00738733,01CE02B3,005282B3,01C28EB3,01DE8F33,01EF0F33,01CF0433,01EF0F33,
01EF0F33,01DF0FB3,01EF0F33,01EF0F33,01EF0F33,01EF0F33,01EF0F33,01EF0F33,01EF0F33,01EF0F33,
01EF0F33,01EF0F33,01EF0F33,01EF0F33,01EF0F33,01EF0F33,01EF0F33,01EF0F33,01EF0F33,01EF0F33,
01EF0F33,01EF0F33,01EF0F33,01EF0F33,01EF0F33,01EF04B3,01E4E633,00948933,012902B3,005282B3,
406006B3,00C4A223,0004A583,00B585B3,00B585B3,00B4A023,006A8AB3,01592023,01402B03,0004A583,
00B585B3,00B585B3,00B4A023,0004A583,0055FC33,006B0B33,040B0E63,0004A583,00E70BB3,017B8CB3,
019B8BB3,0175FC33,000C0C63,037C0463,00E70BB3,037C0663,01592023,FB9FF06F,00D78463,0080006F,
00D687B3,00F92023,FA5FF06F,0609AA83,01592023,F99FF06F,0209AA83,01592023,F8DFF06F,01402B03,
00F787B3,0067E7B3,00E989B3,0089F9B3,006A8AB3,00DA8463,00C0006F,00E00AB3,006A8AB3,0004A583,
00B58C33,018C0C33,0184A023,00C4A223,F6DFF06F;
```

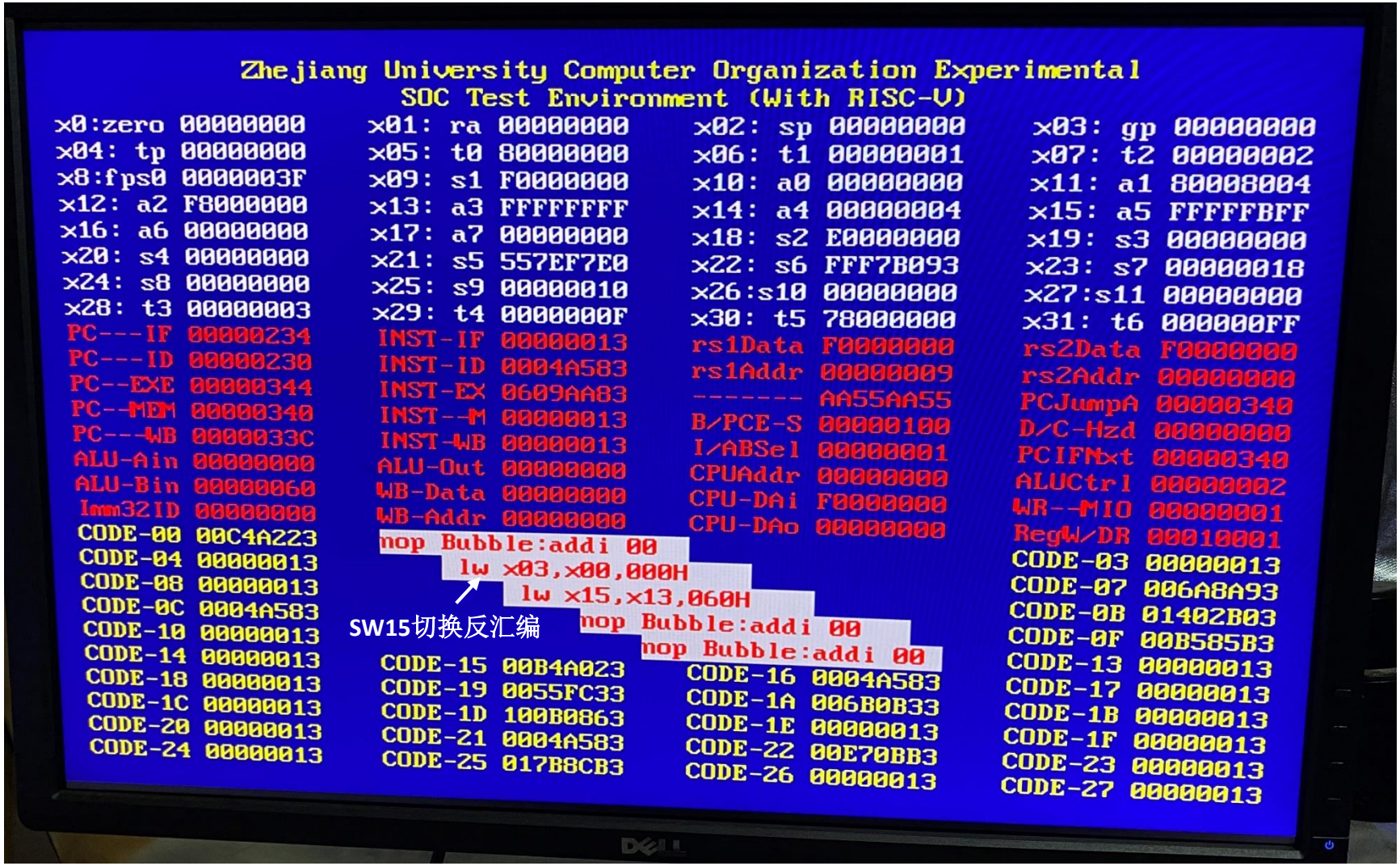
设计要点：数据存储器模块测试

- **32位数据存储器模块**
  - 7段码显示器的地址是E0000000/FFFFFFE0
  - LED显示地址是F0000000/FFFFFF00
  - 请设计存储器模块测试程序
    - 测试结果在7段显示器上指示
- **RAM初始化数据同OExp05/06**

```
memory_initialization_radix=16;
memory_initialization_vector=
f0000000, 000002AB, 80000000, 0000003F, 00000001, FFF70000, 0000FFFF, 80000000,
00000000, 11111111, 22222222, 33333333, 44444444, 55555555, 66666666, 77777777,
88888888, 99999999, aaaaaaaa, bbbbbbbb, cccccccc, dddddddd, eeeeeeee, ffffffff,
557EF7E0, D7BDFBD9, D7DBFDB9, DFCFFCFB, DFCFBFFF, F7F3DFFF, FFFFDF3D, FFFF9DB9,
FFFFBCFB, DFCFFCFB, DFCFBFFF, D7DB9FFF, D7DBFDB9, D7BDFBD9, FFFF07E0, 007E0FFF,
03bdf020, 03def820, 08002300;
```

RAM初始化数据。红色数据为七段LED图形

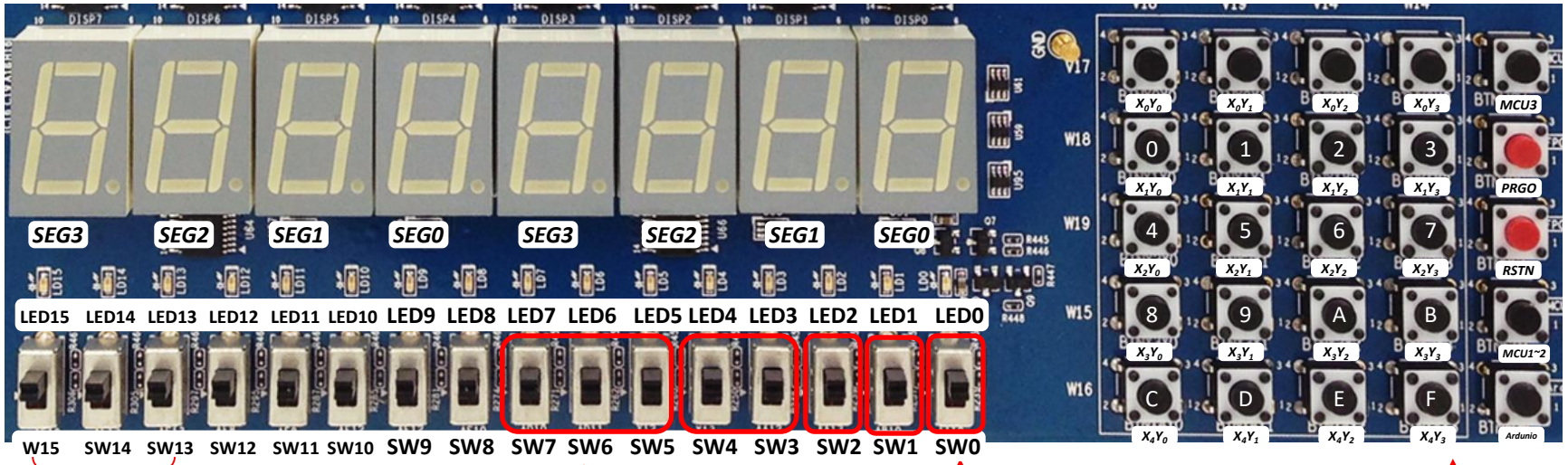
VGA_TESTP增加功能



物理验证

- 使用**RISCV-DEMO9**测数据通路功能
  - DEMO单步调试数据通路
    - 建议增加手动单步功能方便相关性调试
    - 保留原有自动单步，在Clkdiv接入Pulse信号(Arraykeys)
      - 选用SW8切换或控制
    - 结合SW15切换连续5条指令的反汇编
  - DEMO接口功能
    - SW[7:5]=000， SW[2]=0(全速运行)
      - SW[4:3]=00， SW[0]=0，点阵显示程序：跑马灯
      - SW[4:3]=00， SW[0]=0，点阵显示程序：矩形变幻
      - SW[4:3]=01， SW[0]=1，内存数据显示程序：0～F
      - SW[4:3]=10， SW[0]=1，当前寄存器s5(x21)+1显示

设计要点：物理验证接口（详细参见实验二）



SW[13]=0选择测试ROM

SW[]13=1选择测试RAM

SW[14]=0/1测试数据翻页

SW[15]=1显示5条反汇编

SW[7:5]=显示通道选择
SW[7:5]=000：CPU程序运行输出
SW[7:5]=001：测试PC字地址
SW[7:5]=010：测试指令字
SW[7:5]=011：测试计数器
SW[7:5]=100：测试RAM地址
SW[7:5]=101：测试CPU数据输出
SW[7:5]=110：测试CPU数据输入

SW[0]=文本图形选择

SW[1]=高低16位选择

没有使用

SW[2]=CPU单步时钟选择

SW[4:3]=00，点阵显示程序：跑马灯
SW[4:3]=00，点阵显示程序：矩形变幻
SW[4:3]=01，内存数据显示程序：0～F
SW[4:3]=10，当前寄存器+1显示(用户扩展保留)

下载验证流水处理器

- **非IP核仿真**
  - 对自己设计的模块做时序仿真(单周期时仿真过的略)
  - 第三方IP核不做仿真(固核无法做仿真)
  - 流水结构不难，但时序紧凑仿真可以减少大量时间
- **SOC物理验证**
  - 下载流文件.bit
  - 验证调试SOC功能
    - 功能不正确时排查错误
  - 定性观测SOC关键信号
    - 本实验只要求定性观测，功能执行正确

思考题

- 扩展下列指令，相关检测和冒险消除有什么不同：

R-Type：	sra, sll, sltu；
I-Type：	addi, andi, ori, xori, lui, slti , srai, slli, sltiu
B-Type：	bne, blt；
UJ-Type：	Jal；
U-Type：	lui；

- 流水结构增加stall消除数据冒险竞争和flush消除控制冒险竞争，你认为应该先解决那一个更好？
- RISC-V用not taken 预测优化应如何修改本实验
- Stall(flush)和forward相关性检测位置相同吗？


END