

# 浙江大学

## 本科实验报告

课程名称: 计算机组成

设计名称: Pipelined CPU

姓 名: 曾帅

学 号: 3190105729

学 院: 计算机学院

专 业: 计算机科学与技术

班 级: 图灵 1901

指导老师: 姜晓红

2021 年 6 月 10 日

# 目录

1 实验内容 . . . . .	3
2 实验原理 . . . . .	3
2.1 指令格式 . . . . .	3
2.2 各部件设计 . . . . .	4
2.3 数据通路 . . . . .	8
2.4 latch 寄存器 . . . . .	9
2.5 Hazard . . . . .	10
3 实验步骤与调试 . . . . .	11
3.1 IMem 与 DMem 设计 . . . . .	11
3.2 立即数生成模块 . . . . .	12
3.3 寄存器模块 . . . . .	13
3.4 ALU 与 Comperator 模块 . . . . .	14
3.5 PC 模块 . . . . .	15
3.6 控制信号产生模块 . . . . .	16
3.7 Hazard 模块 . . . . .	20
3.8 数据通路的设计 . . . . .	23
3.9 后续操作 . . . . .	30
4 实验结果与分析 . . . . .	31
5 讨论与心得 . . . . .	34

# 1 实验内容

实验的基本要求是实现 RISC-V 架构下，指令集为 RV32I 的子集支持 Hazard 的流水线 CPU。

实验要求实现的指令如下：

- lui, auipc, jal, jalr, lw, sw
- beq, bne, blt, bge, bltu, bgeu
- addi, slti, sltiu, xori, ori, andi, slli, srli, srai
- add, sub, sll, slt, sltu, xor, srl, sra, or, and

# 2 实验原理

## 2.1 指令格式

RV32I 的指令格式大致分为一下 6 种，每种指令都是 32 位指令，所以指令在内存中必须以 4 字节对齐。

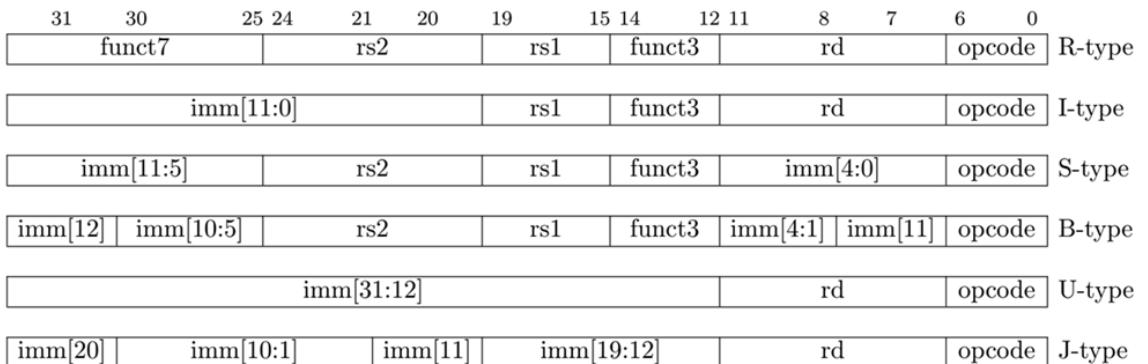


图 2.1: 指令格式

每种指令格式对应者一类或者多类指令，根据每种指令实现需要的条件，对指令格式也有不同的要求，操作码也各不相同

操作	操作码	包含指令
R-type	0110011	add, sub, sll, slt, sltu, xor, srl, sra, or, and
Arithmetic I-type	0010011	addi, slli, srli, srai, andi, ori, xori, slti, sltui
load I-type	0000011	lw
jump I-type	1100111	jalr
S-type	0100011	sw
B-type	1100011	beq, bne, blt, bge, bltu, bgeu
U-type(lui)	0110111	lui
U-type(auipc)	0010111	auipc
J-type	1101111	jal

根据指令格式，我们可以从中知道立即数的存储方式：

操作类型	立即数组织方式
I-type	$\text{Imm} = \{ 21\{\text{Ins}[31]\}, \text{Ins}[30:25], \text{Ins}[24:21], \text{Ins}[20] \}$
S-type	$\text{Imm} = \{ \{21\{\text{Ins}[31]\}\}, \text{Ins}[30:25], \text{Ins}[11:8], \text{Ins}[7] \}$
U-type	$\text{Imm} = \{ \text{Ins}[31], \text{Ins}[30:20], \text{Ins}[19:12], 12'b0 \}$
J-type	$\text{Imm} = \{ \{12\{\text{Ins}[31]\}\}, \text{Ins}[19:12], \text{Ins}[20], \text{Ins}[30:25], \text{Ins}[24:21], 1'b0 \}$
B-type	$\text{Imm} = \{ \{20\{\text{Ins}[31]\}\}, \text{Ins}[7], \text{Ins}[30:25], \text{Ins}[11:8], 1'b0 \}$

需要注意的是，之所以 U-type、J-type、B-type 指令的立即数需要末尾加 0，与 PC 地址恒为 2 的倍数有关，省略恒为 0 的最末尾可以使得立即数表示大一倍，可能我们会注意到 PC 地址实际上恒为 4 的倍数，为什么不直接省略末尾两个 0 呢？是因为还有一种 2Byte 的压缩指令，如果省略两位，那它将无法表示。

## 2.2 各部件设计

设计单周期 CPU 首先应该对整体结构有深刻的理解，之后在根据需求设计各组成部件与部件接口。在我的设计中，我将整个 CPU 分为了，整个单周期 CPU 加其测试框架结构图大致如下：

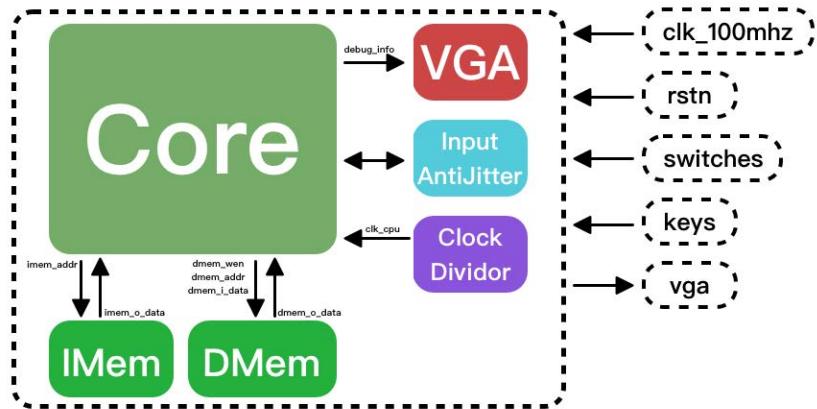
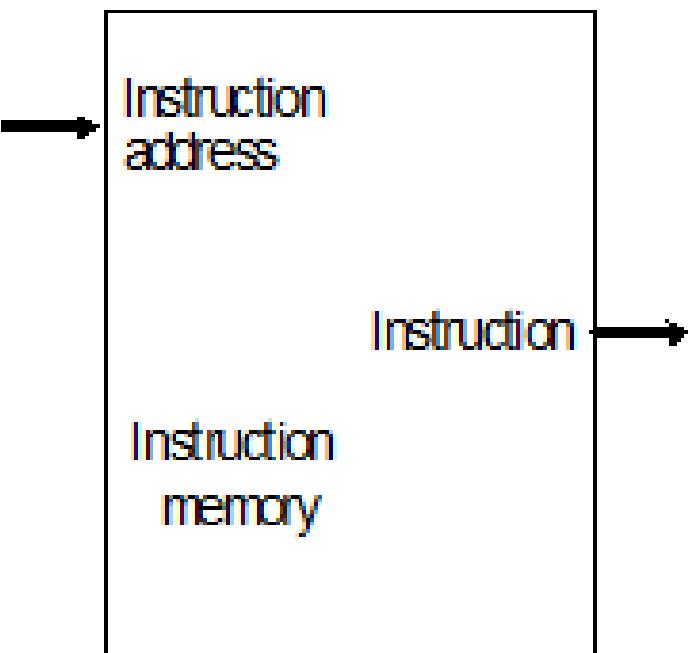


图 2.2: CPU 架构

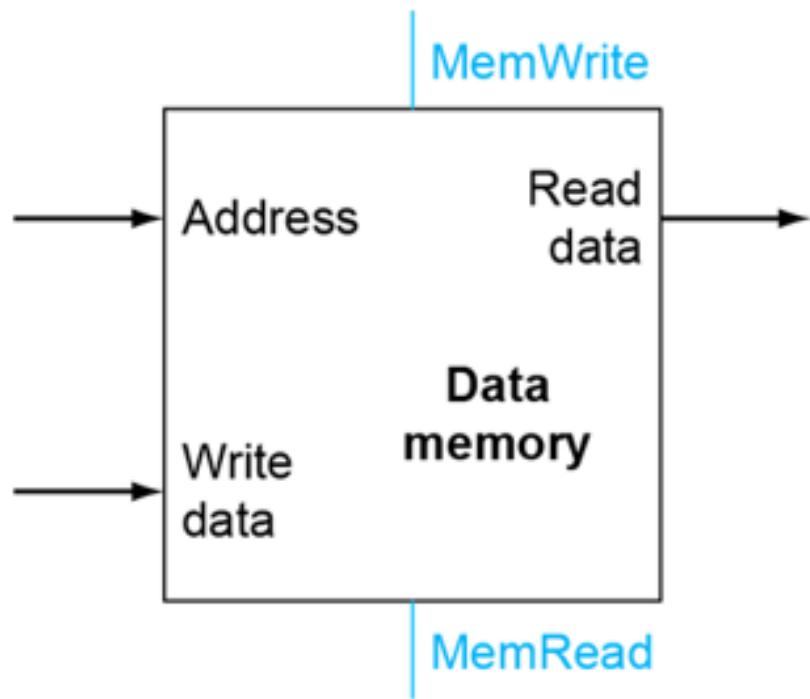
**Instruction Memory** IMem 负责根据 PC 给出的地址取出指令传入 CPU Core，并且它是通过 imem\_data.mem 文件初始化的，同时需要注意的是它是一个不受时钟控制的组合逻辑，也就意味这它的输出是常通的。原理图如下所示



a. Instruction memory

图 2.3: Instruction Memory

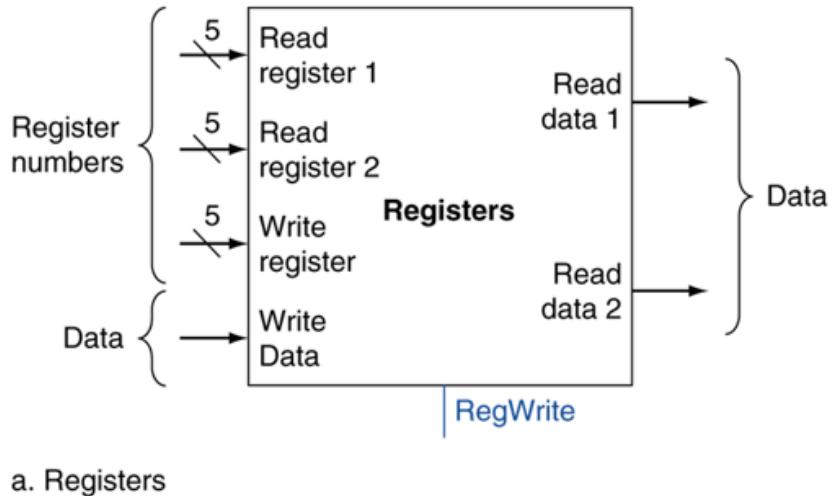
**Data Memory** DMem 复杂内存数据的读取与接入，CPU Core 给出数据地址，DMem 就应该得到对应为值的数据，并放在常通的位置；当控制 DMem 写入的信号置为高位时，DMem 负责将输入数据在时钟下降沿写入对应的内存地址，原理图如下所示



a. Data memory unit

图 2.4: Data Memory

**Registers** Registers 在之前的实验中已经设计过，主要是数据的常通读取与数据在时钟下降沿的写入，需要注意的只有当 RegWrite 信号在高位是才会将数据写入。



a. Registers

图 2.5: Registers

**Control** 在我的实验中 Control 模块分为两个部分，一个部分是接受 Instruction 的 Operation Code 输入的 Control\_Unit 模块，此模块根据 Opcode 识别指令类别，并根据指令类别指定各个信号输出，以及 ALU 需要执行的操作 ALUOp。具体阐述将在控制信号部分阐明

**ALUControl** 此模块接受 Control 模块的 ALUOp 输出、Funct3、Funct7 作为输入，根据它们判断指令类型，再根据指令需求输出需要 ALU 执行何等操作的控制信号。下表说明了基本的控制关系：

指令类型	Funct3	Funct7	ALUOp	ALU 执行操作
lw or sw or jal	***	*	00	0000(add)
branch	***	*	01	0001(sub)
sub	000	1	10	0001(sub)
add	000	0	10	0000(add)
and	111	0	10	1001(and)
or	110	0	10	1000(or)
slt	010	0	10	0011(slt)
sll	001	0	10	0010(sll)
sltu	011	0	10	0100(sl tu)
xor	100	0	10	0101(xor)
srl	101	0	10	0110(srl)

这里需要注意的是由于 Funct7 实际只有起作用，所以这里的 Funct7 特指的是指令的

第 31 位 (Funct7 的第二高位), 同时由于算术 I-type 的除了 Funct3 稍有不同以外, 其他操作都与对应的 R-type 指令基本相同, 这里不再赘述。

**Comperator** 在我的实验中, Comperator 与 ALU 是分开的两个部件, Comperator 是当指令为 Branch 时, 根据指令的类型判断是何等比较操作并输出比较结果 Zero 的部件。

## 2.3 数据通路

完成了个部件的设计, 数据通路就是将各个部分连接起来, 保证各功能正常实现的关键部分, 下面是总体的原理图

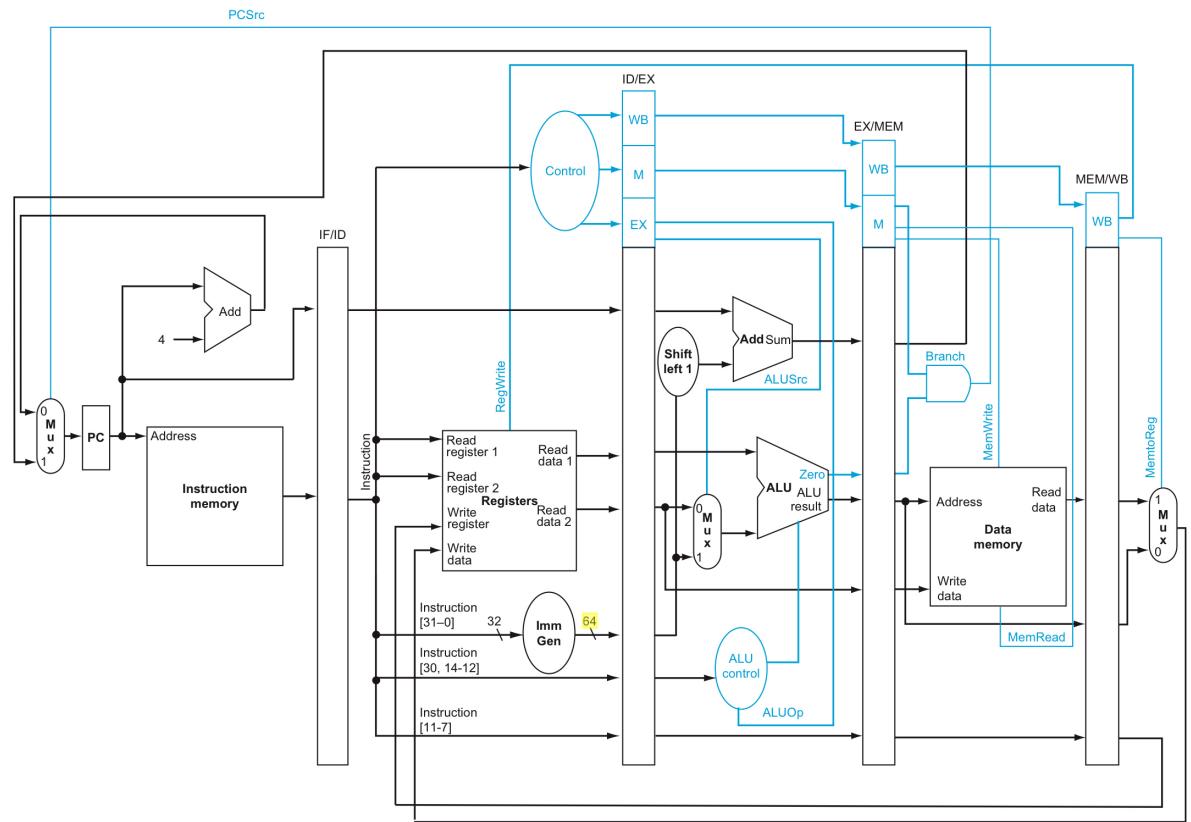


图 2.6: Datapath

**控制信号** 数据通路中最为重要的就是各个控制信号, 解释各个控制信号的作用是 CPU 设计阐述的必要过程

控制信号	控制信号含义
ALUSrc	决定 ALU 的操作数之一是来自寄存器的 rs2_val 还是立即数
MemtoReg	决定是否将 Data Memory 的内容写给寄存器
RegWrite	决定是否对寄存器进行写入
MemRead	决定是否用到 DMem 的数据
BranchANDZero	决定是否进行 Branch 跳转
Jump	决定是否进行 Jump 跳转
ALUOp	决定 ALU 进行何种操作
csr	决定是否进行 csr 类操作
lui	决定是否为 lui 指令

根据各个指令的操作需求，我们可以得到如下对应关系：

指令类型	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	Jump	ALUOp
R-type	0	00	1	0	0	0	0	10
LW	1	01	1	1	0	0	0	00
SW	1	XX	0	0	1	0	0	00
B-type	0	XX	0	0	0	1	0	01
A I-type	1	00	1	0	1	0	0	11
UJ-type	X	10	1	0	0	0	1	XX

## 2.4 latch 寄存器

下面要介绍的是 latch 寄存器的设计，由于整个流水线被分为 IF, ID, EX, MEM, WB 五级，所以我们需要四个段寄存器，并且段寄存器的主要作用是将上一阶段的某些信号值传入下一阶段，并且兼有置零 (Flush) 与保持不变 (Stall) 的作用。

**基本结构** 段寄存器基本是在时钟上升沿决定相应阶段传入的寄存器值，主要设计结构如下所示：

```

1 module latch(
2     input clk ,
3     input stall ,
4     input Flush ,
5     input A_In ,
6     input A_Out
7 );
8
9     always @(*) begin
10         if(~stall) begin

```

```

11      A_Out <= Flush ? 0 : A_In;
12  end
13 end

```

**信号的传递** 每个阶段之间需要传输的信号是 Latch 设计中最为重要的部分，由下表给出

段寄存器	信号
IF/ID	PC, Instruction
ID/EX	All Control Signals, PC, Imm, Instruction
EX/MEM	ALUOut, RegWrite, MemWrite, MemtoReg, rd, lui, PC, Imm, Instruction
MEM/WB	Mem_w_data, RegWrite, MemWrite, MemWrite, PC, Imm, Instruction

## 2.5 Hazard

在此次的实验中，Hazard 由结构竞争、数据冒险和控制竞争三部分组成，简单起见本模块被设计在 ID 阶段，利用段寄存的 Flush 与 Stall 功能实现插入 bubble 或者清空段寄存器数据并且本次实验还设计了 Forward Units 以减少停顿次数，提高流水线效率，同时本此实验中实现了 Branch No Taken 的跳转策略，在 EX 阶段识别并执行跳转语句。具体的设计分几部分阐述如下：

**Forward Units** 前送只涉及到 EX, MEM, WB 三个阶段的数据前送，在此实验中分为 ALU 的两个数据输入口进行讨论，其中 ALU 的 1 号操作数口接受来自：1. MEM 阶段的上一条指令 EX 阶段的 ALU 计算结果 2. WB 阶段将要写回 Memory 的数据结果 3. ID 阶段传入的 rs1\_val(正常值) 所以前送 ALU 操作数 1 口的条件应该是：

- \* EX/MEM.RegisterRd = ID/EX.RegisterRs1

- \* MEM/WB.RegisterRd = ID/EX.RegisterRs1

同样的分析可以得到，前送 ALU 操作数 2 口的条件应该是：

- \* EX/MEM.RegisterRd = ID/EX.RegisterRs2

- \* MEM/WB.RegisterRd = ID/EX.RegisterRs2

这样使得涉及到算数指令的 Read After Write 冲突无需停顿（插入一个 bubble）即可得到正确结果，使得涉及到 Load 指令的 RAW 情形仅需一次停顿即可避免数据冲突。

**Branch No Taken** 对于跳转指令的判断，此实验我均是在 EX 阶段判断并跳转，我采取了 Branch No Taken 的设计方式：若 EX 阶段检测到不满足跳转条件 (Branch 指令不跳转)，则流水线继续执行，若 EX 阶段检测到跳转 (Branch 满足跳转条件或 Jal, Jalr 必定跳转)，则将下一条 PC 地址的值修改为跳转的目标地址，并且将 IF/ID, ID/EX 段寄存器的值全部 Flush 掉 (清空)，这样可以使得无效指令全部被清空，并且跳转指令的剩下部分继续向后流动直至执行完毕 (例如 jal, jalr 指令还需将 PC + 4 的值写入寄存器)

**Data Hazard** 由于拥有了前递控制，所以实验中仅需将涉及到 Load 指令的 RAW 情形在 EX 阶段判断并插入 bubble，在此实验中，插入 bubble 的判断条件就变为了：

- \* EX 阶段执行的语句是 Load 指令
- \* WB 阶段的 RegWrite(寄存器写入使能信号) 为高位 (要写入寄存器)
- \* EX.RegisterRd = ID.RegisterRs1 或者 EX.RegisterRd = ID.RegisterRs2

并且插入 bubble 的方式就是将 EX/MEM 段寄存器 Flush 掉，并使得 IF/ID 与 ID/EX 段寄存器 Stal(停顿) 一个周期

### 3 实验步骤与调试

#### 3.1 IMem 与 DMem 设计

IMem 与 DMem 均使用了系统函数进行初始化，将值存入 32 位寄存器中进行读写，需要注意的是本实验采取的策略是读取常通，时钟上升沿写入。

**IMem 设计** IMem 需要注意的是输入的 PC 地址需要除以 4，才能获得正确的指令

```
1 module IMem(
2     input [31:0] addr,
3     output reg [31:0] data
4 );
5
6 (* ram_style = "block" *) reg [31:0] mem [0:4095];
7 initial $readmemh("imem_data.mem", mem);
8
9 always @(*) begin
10    data <= mem[addr>>2];
11 end
12
```

13 | **endmodule**

### DMem Dmem 仅在时钟下降沿写入

```
1 module DMem(
2     input clk ,
3     input wen,
4     input [31:0] addr ,
5     input [31:0] i_data ,
6     output reg [31:0] o_data
7 );
8
9     (* ram_style = "block" *) reg [31:0] mem [0:4095];
10    initial $readmemh("dmem_data.mem", mem);
11
12    always @(*) begin
13        o_data <= mem[addr];
14    end
15
16    always @ (posedge clk) begin
17        if(wen == 1)
18            mem[addr] <= i_data;
19    end
20
21 endmodule
```

## 3.2 立即数生成模块

ImmGen 接受指令 Instruction，并根据实验原理部分的指令格式，译码出对应指令的立即数 Imm，在此模块中 lui, auipc, jal 指令已经直接在末尾补零

```
1 module ImmGen(
2     input wire [31:0] Ins ,
3     output reg [31:0] Imm
4 );
5
6 always @(*) begin
7     case (Ins [6:0])
8         7'b0010011: Imm = { {21{Ins [31]}}, Ins [30:25], Ins [24:21], Ins [20]
9                         };      //arith_I type: addi, slti, sltiu, xori, ori, andi
10        7'b0100011: Imm = { {21{Ins [31]}}, Ins [30:25], Ins [11:8], Ins [7]
11                         };      //store
```

```

10      7'b0110111: Imm = { Ins[31], Ins[30:20], Ins[19:12], 12'b0 };
11          //lui
12      7'b0010111: Imm = { Ins[31], Ins[30:20], Ins[19:12], 12'b0 };
13          //auipc
14      7'b1101111: Imm = { {12{Ins[31]}}, Ins[19:12], Ins[20], Ins
15          [30:25], Ins[24:21], 1'b0 }; //jal
16      7'b1100111: Imm = { {21{Ins[31]}}, Ins[30:25], Ins[24:21], Ins[20]
17          }; //jalr
18      7'b1100011: Imm = { {20{Ins[31]}}, Ins[7], Ins[30:25], Ins[11:8],
19          1'b0 }; //branch
20      default:     Imm = { {21{Ins[31]}}, Ins[30:25], Ins[24:21], Ins
21          [20] }; // IMM_I
22      endcase
23  end
24 endmodule

```

### 3.3 寄存器模块

寄存器的设计在之前的实验中已经阐明，在此不再赘述

```

1 `include "Defines.vh"
2
3 module RegFile(
4     `VGA_DBG_RegFile_Outputs
5     input clk, rst,
6     input wen, // 写使能
7     input [4:0] rs1, // 源寄存器1的编号
8     input [4:0] rs2, // 源寄存器2的编号
9     input [4:0] rd, // 目的寄存器的编号
10    input [31:0] i_data, // 写入的数据
11    output [31:0] rs1_val, // 源寄存器1的输出数据
12    output [31:0] rs2_val // 源寄存器2输出的数据
13 );
14
15    integer i;
16    reg [31:0] regs [1:31];
17    `VGA_DBG_RegFile_Assignments
18
19    assign rs1_val = (rs1 == 0) ? 0 : regs[rs1];
20    assign rs2_val = (rs2 == 0) ? 0 : regs[rs2];
21
22    always @(posedge clk or posedge rst) begin

```

```

23      if( rst == 1) begin
24          for( i = 1; i < 32; i = i + 1)
25              regs[ i ] <= 0;
26      end
27      else if( (rd != 0) && (wen == 1) )
28          regs[ rd ] <= i_data;
29      end
30 endmodule

```

### 3.4 ALU 与 Comperator 模块

ALU 与 Comperator 模块复杂计算与逻辑比较部分，在之前的设计中已经阐明，在此不再赘述。

```

1 module Alu(
2     input [31:0] a_val,
3     input [31:0] b_val,
4     input [3:0] ctrl,
5     output reg [31:0] result
6 );
7
8     always @ (a_val or b_val or ctrl) begin
9         case( ctrl )
10             4'd0: result <= a_val + b_val;
11             4'd1: result <= a_val - b_val;
12             4'd2: result <= (a_val << b_val); // SLL
13             4'd3: result <= ($signed(a_val) < $signed(b_val)) ? 1 : 0; // LT
14             4'd4: result <= (a_val < b_val) ? 1 : 0; // LTU
15             4'd5: result <= a_val ^ b_val; // XOR
16             4'd6: result <= (a_val >> b_val); // SRL
17             4'd7: result <= (( $signed(a_val) ) >>> b_val[4:0]); // SRA
18             4'd8: result <= a_val | b_val; // OR
19             4'd9: result <= a_val & b_val; // AND
20             default: result <= a_val + b_val;

```

```

          ADD
21      endcase
22  end
23 endmodule

```

为方便译码，Comperator 模块的控制码稍有改动

```

1 module Comperator(
2     input [31:0] a_val,
3     input [31:0] b_val,
4     input [2:0] ctrl,
5     output reg result
6 );
7
8     always @(a_val or b_val or ctrl) begin
9         case(ctrl)
10            3'd0: result = (a_val == b_val) ? 1 : 0;
11            3'd1: result = (a_val == b_val) ? 0 : 1;
12            3'd4: result = ($signed(a_val) < $signed(b_val)) ? 1 : 0;
13            3'd5: result = ($signed(a_val) >= $signed(b_val)) ? 1 : 0;
14            3'd6: result = (a_val < b_val) ? 1 : 0;
15            3'd7: result = (a_val >= b_val) ? 1 : 0;
16        default: result = 0;
17     endcase
18 end
19 endmodule

```

### 3.5 PC 模块

PC 模块是较为重要的模块，将它单独列出来作为一个模块是为了方便设计过程中指令的拓展与纠错，PC 模块的读取也是常通的，但只有当时钟上升沿时才会进行下一条 PC 地址的写入，具体的设计理念在代码注释中阐明。

```

1 module PC(
2     input [31:0] PC, PCD, PCE,
3     input rst,
4     input [1:0] JumpE,      //EX阶段的Jal与Jalr合信号
5     input Branch,          //EX阶段的Branch控制信号
6     input [31:0] ImmE,      //EX阶段的立即数
7     input Zero,             //EX阶段的Comperator比较结果
8     input [31:0] JalTarge, JalrTarge,
9     output reg [31:0] nextpc
10    );

```

```

11   wire [31:0] PCplus4, BranchTarge;
12   wire BranchAND;
13
14   assign BranchTarge = PCE + ImmE;
15   assign PCplus4      = PC  + 32'd4;
16
17   assign BranchAND = Branch & Zero;
18
19   always @(*) begin
20     if(rst == 1) nextpc <= 0;
21     else begin
22       if(BranchAND)           nextpc <= BranchTarge; //branch
23       else if(JumpE == 2'b01) nextpc <= JalTarge;    //jal
24       else if(JumpE == 2'b10) nextpc <= JalrTarge;   //auipc and
25                     jalr
26       else                   nextpc <= PCplus4;      //PC+4
27     end
28   end
endmodule

```

### 3.6 控制信号产生模块

控制模块的产生由实验原理部分的指令译码为蓝本，结合 if...else 判断设计而成，基本是对指令结构的一种判断，值得注意的是此 ControlUnit 包含两个二级译码模块，一个是传入 ALUOp 信号进行进一步译码的 ALU 操作控制模块 ALU\_Control 模块，另一个是传入 csr 与 insill(指令不存在信号) 的中断寄存器操作指令译码器 CSR\_Control 模块。具体设计如下所示：

```

1 module Control_Unit(
2   input [6:0] OpCode,          //instruction [6:0]
3   output reg RegWrite,        //寄存器写入使能
4   output reg ALUSrc,         //ALU选择立即数还是rs2_val
5   output reg Branch,         //Branch指令
6   output reg [1:0] Jump,     //jal, jalr控制
7   output reg MemRead,        //内存的读取
8   output reg MemWrite,        //内存的写入
9   output reg [1:0] MemtoReg, //选择写入寄存器的值的来源
10  output reg [1:0] ALUOp,     //ALU操作控制
11  output reg lui,            //lui指令与否
12 );
13
14  always @ (*) begin

```

```

15   if(OpCode == 7'b0110011) begin      // R-Type-AND, OR, SUB, ADD
16     ALUSrc <= 0;
17     MemtoReg <= 2'b00;
18     RegWrite <= 1;
19     MemRead <= 0;
20     MemWrite <= 0;
21     Branch <= 0;
22     Jump <= 2'b00;
23     ALUOp <= 2'b10;
24     lui <= 0;
25   end
26   else if(OpCode == 7'b0000011) begin // Type Load - LW
27     ALUSrc <= 1;
28     MemtoReg <= 2'b01;
29     RegWrite <= 1;
30     MemRead <= 1;
31     MemWrite <= 0;
32     Branch <= 0;
33     Jump <= 2'b00;
34     ALUOp <= 2'b00;
35     lui <= 0;
36   end
37   else if(OpCode == 7'b0100011) begin // Type Store - SW
38     ALUSrc <= 1;
39     MemtoReg <= 2'b00;                  // don't care
40     RegWrite <= 0;
41     MemRead <= 0;
42     MemWrite <= 1;
43     Branch <= 0;
44     Jump <= 2'b00;
45     ALUOp <= 2'b00;
46     lui <= 0;
47   end
48   else if(OpCode == 7'b1100011) begin // Type Branch
49     ALUSrc <= 0;
50     MemtoReg <= 2'b00;                  // don't care
51     RegWrite <= 0;
52     MemRead <= 0;
53     MemWrite <= 0;
54     Branch <= 1;
55     Jump <= 2'b00;
56     ALUOp <= 2'b01;

```

```

57     lui <= 0;
58 end
59 else if(OpCode == 7'b0010011) begin //andi ori ...
60     ALUSrc <= 1;
61     MemtoReg <= 2'b00;
62     RegWrite <= 1;
63     MemRead <= 0;
64     MemWrite <= 0; //修改
65     Branch <= 0;
66     Jump <= 2'b00;
67     ALUOp <= 2'b11;
68     lui <= 0;
69 end
70 else if(OpCode == 7'b1101111) begin //Jump jal
71     ALUSrc <= 0; //don't care
72     MemtoReg <= 2'b10;
73     RegWrite <= 1;
74     MemRead <= 0;
75     MemWrite <= 0;
76     Branch <= 0;
77     Jump <= 2'b01;
78     ALUOp <= 2'b00;
79     lui <= 0;
80 end
81 else if(OpCode == 7'b1100111) begin //jalr
82     ALUSrc <= 1;
83     MemtoReg <= 2'b10;
84     RegWrite <= 1;
85     MemRead <= 0;
86     MemWrite <= 0;
87     Branch <= 0;
88     Jump <= 2'b10;
89     ALUOp <= 2'b00;
90     lui <= 0;
91 end
92 else if(OpCode == 7'b0010111) begin //auipc
93     ALUSrc <= 1;
94     MemtoReg <= 2'b11;
95     RegWrite <= 1;
96     MemRead <= 0;
97     MemWrite <= 0;
98     Branch <= 0;

```

```

99      Jump <= 2'b00;
100     ALUOp <= 2'b00;
101     lui <= 0;
102   end
103   else if(OpCode == 7'b0110111) begin //lui
104     ALUSrc <= 1;
105     MemtoReg <= 2'b00;
106     RegWrite <= 1;
107     MemRead <= 0;
108     MemWrite <= 0;
109     Branch <= 0;
110     Jump <= 0;
111     ALUOp <= 2'b00;
112     lui <= 1;
113   end
114   else begin //instruction illegal
115     ALUSrc <= 0;
116     MemtoReg <= 2'b00;
117     RegWrite <= 0;
118     MemRead <= 0;
119     MemWrite <= 0;
120     Branch <= 0;
121     Jump <= 0;
122     ALUOp <= 2'b00;
123     lui <= 0;
124     csr <= 0;
125     insill <= 1;
126   end
127 end
128 endmodule

```

根据最高级控制信号产生模块设计的次级控制信号产生器如下所示: 其中 ALUControl 是根据 ALUOp 信号与指令的 Funct3, Funct7(实际只有一位有效) 产生 ALU 的控制信号 Ctrl

```

1 module ALUControl(
2   input [1:0] ALUOp,
3   input Funct7,           //actually only one bit of Funct7 makes sense
4   input [2:0] Funct3,
5   output reg [3:0] ctrl
6 );
7
8 always @(*) begin

```

```

9   case(ALUOp)
10    2'b00: ctrl <= 4'b0000; //lw or sw : add
11    2'b01: ctrl <= 4'b0001; //branch : sub
12    2'b10:                      //R-type
13    begin
14      if(Funct7 == 1) begin
15        if(Funct3 == 3'b101) ctrl <= 4'b0111; //sra
16        else ctrl <= 4'b0001;                  //sub
17      end
18      else if(Funct3 == 3'b000) ctrl <= 4'b0000; //add
19      else if(Funct3 == 3'b111) ctrl <= 4'b1001; //and
20      else if(Funct3 == 3'b110) ctrl <= 4'b1000; //or
21      else if(Funct3 == 3'b010) ctrl <= 4'b0011; //slt
22      else if(Funct3 == 3'b001) ctrl <= 4'b0010; //sll
23      else if(Funct3 == 3'b011) ctrl <= 4'b0100; //sltu
24      else if(Funct3 == 3'b100) ctrl <= 4'b0101; //xor
25      else /*if(Funct3 == 3'b101)*/ ctrl <= 4'b0110; //srl
26    end
27    2'b11:
28    begin
29      if(Funct3 == 3'b101) begin
30        if(Funct7 == 1) ctrl <= 4'b0111; //srai
31        else ctrl <= 4'b0110;          //srli
32      end
33      else if(Funct3 == 3'b000) ctrl <= 4'b0000; //addi
34      else if(Funct3 == 3'b111) ctrl <= 4'b1001; //andi
35      else if(Funct3 == 3'b110) ctrl <= 4'b1000; //ori
36      else if(Funct3 == 3'b010) ctrl <= 4'b0011; //slti
37      else if(Funct3 == 3'b001) ctrl <= 4'b0010; //slli
38      else if(Funct3 == 3'b011) ctrl <= 4'b0100; //sltiu
39      else /*if(Funct3 == 3'b100)*/ ctrl <= 4'b0101; //xori
40    end
41  endcase
42 end
43 endmodule

```

### 3.7 Hazard 模块

在此实验中我选择将 Hazard 模块作为一个组合逻辑放置在 ID 和 EX 两阶段之间，并且此模块将会实现插入 bubble，前递和 Branch No Taken。

---

```
1 module Stall(
```

```

2   input rst ,
3   input BranchE ,
4   input [31:0] PCE, ImmE,
5   input [1:0] JumpD,
6   input [1:0] JumpE,
7   input [4:0] rs1D, rs2D, rs1E, rs2E, rdE, rdM, rdW,
8   input [31:0] ResultM, DataWB, rs1_valE, rs2_valE ,
9   input [1:0] MemtoRegE,
10  input ALUSrcE,
11  input RegWriteM, RegWriteW,
12  output reg [31:0] mem_w_data,
13  output reg StallF, FlushF, StallD, FlushD, StallE, FlushE, StallM,
14    FlushM, StallW, FlushW,
15  output reg [31:0] op1, op2      //ALU的两个操作数
16 );
17
18 always @(*) begin
19   if(rst) begin
20     FlushF <= 1;
21     FlushD <= 1;
22     FlushE <= 1;
23     FlushM <= 1;
24     FlushW <= 1;
25
26     StallF <= 0;
27     StallD <= 0;
28     StallE <= 0;
29     StallM <= 0;
30     StallW <= 0;
31
32     op1 <= rs1_valE;
33     op2 <= rs2_valE;
34     mem_w_data <= rs2_valE;
35   end
36   else begin
37     if(MemtoRegE == 2'b01 && (RegWriteW && rdE != 0 && ((rdE ==
38       rs1D) || (rdE == rs2D)))) begin //Load of RAW
39       FlushF <= 0;
40       FlushD <= 0;
41       FlushE <= 1;
42       FlushM <= 0;
43       FlushW <= 0;

```

```

42
43         StallF <= 1;
44         StallD <= 1;
45         StallE <= 0;
46         StallM <= 0;
47         StallW <= 0;
48     end
49     else if (JumpE == 2'b01 || JumpE == 2'b10 || BranchE == 1)
50         begin //jal , jalr , Branch in EX
51             FlushF <= 0;
52             FlushD <= 1;
53             FlushE <= 1;
54             FlushM <= 0;
55             FlushW <= 0;
56
57             StallF <= 0;
58             StallD <= 0;
59             StallE <= 0;
60             StallM <= 0;
61             StallW <= 0;
62         end
63     else begin
64         FlushF <= 0;
65         FlushD <= 0;
66         FlushE <= 0;
67         FlushM <= 0;
68         FlushW <= 0;
69
70         StallF <= 0;
71         StallD <= 0;
72         StallE <= 0;
73         StallM <= 0;
74         StallW <= 0;
75     end
76
77     /*前递模块*/
78     //ALU的一号操作数
79     if      (rdM && rdM == rs1E && RegWriteM) op1 <= ResultM;
80     else if (rdW && rdW == rs1E && RegWriteW) op1 <= DataWB;
81     else
82         op1 <= rs1_valE;
83
84     //else op1 <= PCE - 4; //ALU 1号操作数来自PC, auipc 指令

```

```

83
84      //ALU的二号操作数    来自 rs2 / 立即数
85      if          (ALUSrcE)                      op2 <= ImmE;
86      else if   (rdM && rdM == rs2E && RegWriteM) op2 <= ResultM ;
87      else if   (rdW && rs2E == rdW && RegWriteW) op2 <= DataWB;
88      else                                op2 <= rs2_valE ;
89
90      //Memmory将要写入的数据前送
91      if  (rdM && rdM == rs2E && RegWriteM)      mem_w_data <=
92          ResultM ;
93      else if (rdW && rs2E == rdW && RegWriteW) mem_w_data <= DataWB
94          ;
95      else                                mem_w_data <=
96          rs2_valE ;
      end
    end
endmodule

```

### 3.8 数据通路的设计

数据通路在此实验中即为 Core.v 文件，是将之前的部分与整个测试框架连接起来的文件，基本设计参照实验原理部分架构图。

```

1 `include "Defines.vh"
2 module Core(
3     `VGA_DBG_Core_Outputs
4     input clk , rst ,
5     input [31:0] imem_o_data ,
6     input [31:0] dmem_o_data ,
7     output [31:0] imem_addr ,
8     output dmem_wen ,
9     output [31:0] dmem_addr ,
10    output [31:0] dmem_i_data
11 );
12
13    wire MemRead, MemReadE;
14    wire [1:0] MemtoReg, MemtoRegE, MemtoRegM, MemtoRegW;
15    wire ZeroE ;
16    wire [1:0] ALUOp;
17    wire MemWrite, MemWriteE, MemWriteM, MemWriteW;
18    wire [31:0] Op1, Op2;
19    wire Branch, BranchE;

```

```

20   wire [1:0] Jump, JumpE;
21
22   wire [3:0] CtrlD, CtrlE;
23   wire [31:0] imm, ImmE, ImmM, ImmW;
24   wire Load, LoadE, LoadM, LoadW;
25   wire rs1_used, rs2_used;
26   wire [1:0] rsuse;
27   wire [4:0] rs1D, rs2D, rd, rs1E, rs2E, rdE, rs1M, rs2M, rdM, rdW;
28   wire [31:0] rs1_valD, rs2_valD, rs1_valE, rs2_valE, rs2_valM;
29   wire [31:0] ALUOutE, ALUOutM;
30   wire [31:0] PCF, PCE, PCD, PCM, PCW, PC_In;
31   wire [31:0] Ins, InsE, InsM, InsW;
32   wire StallF, FlushF, FlushD, StallD, FlushE, StallE, FlushM, StallM,
      FlushW, StallW;
33   reg [31:0] ResultE, ResultM;
34   wire lui, luiE, luiM;
35   wire [31:0] ResultW; //DataMem写入的数据
36   wire RegWrite, RegWriteE, RegWriteM, RegWriteW;
37   wire ALUSrc, ALUSrcE;
38   wire [31:0] mem_w_data, mem_w_dataM;
39   wire [31:0] JalTarge, JalrTarge;
40 //*****IF部分*****
41 //计算下一条指令的PC值的模块，常通
42 PC pc(
43     .PC(PCF), //IF阶段的PC值
44     .PCD(PCD),
45     .PCE(PCE),
46     .rst(rst), //清零
47     .JumpE(JumpE), //jalr指令跳转标志EX阶段
48     .Branch(BranchE), //Branch跳转标志EX阶段
49     .ImmE(ImmE), //Branch跳转到PCE + ImmE
50     .Zero(ZeroE), //Branch是否要跳转的必要条件之一
51     .nextpc(PC_In), //此周期的下一条指令的PC地址
52     .JalTarge(JalTarge),
53     .JalrTarge(JalrTarge)
54 );
55
56 IF IF_ID1(
57     .clk(clk),
58     .en(~StallF),
59     .clc(FlushF),
60     .PC_In(PC_In), //下一条预备好写入的PC地址

```

```

61     .PCF(PCF)           // 此周期的PC地址
62   );
63
64   assign imem_addr = PCF;      // 从IMem中取得指令
65
66 // ***** ID部分 *****
67   ID IF_ID2(
68     .clk(clk),
69     .clc(FlushD),
70     .en(~StallD),
71     .PCF(PCF),
72     .IMem_o_data(imem_o_data), // IF阶段读取到的原始指令
73     .IMemData(Ins),          // 经过选择后的指令
74     .PCD(PCD)               // 将PCF保存到ID模块
75   );
76
77 ImmGen(.Ins(Ins), .Imm(imm)); // 这里的imm为ImmD(ID阶段的指令生成的立即数)
78
79 // 生成的均是ID阶段的控制指令
80 Control_Unit con(
81   .OpCode(Ins[6:0]),
82   .RegWrite(RegWrite),
83   .ALUSrc(ALUSrc),
84   .Branch/Branch,
85   .Jump/Jump,
86   .MemRead(MemRead),
87   .MemWrite(MemWrite),
88   .MemtoReg(MemtoReg),
89   .ALUOp(ALUOp),
90   .lui(lui),
91   .rsuse(rsuse)
92 );
93
94 ALUControl ALUcon(
95   .ALUOp(ALUOp),
96   .Funct7(Ins[30]),
97   .Funct3(Ins[14:12]),
98   .ctrl(CtrlD)
99 );
100
101 assign rs1D = Ins[19:15];

```

```

102 assign rs2D = Ins[24:20];
103 assign rd = Ins[11:7];
104
105 RegFile regfile(
106     `VGA_DBG_RegFile_Arguments
107     .clk(~clk), //下降沿写入double bump
108     .rst(rst),
109     .wen(RegWriteW), //WB阶段的数据
110     .rs1(Ins[19:15]),
111     .rs2(Ins[24:20]),
112     .rd(rdW), //WB阶段才能决定的数据
113     .i_data(ResultW), //WB阶段的返回数据
114     .rs1_val(rs1_valD),
115     .rs2_val(rs2_valD)
116 );
117
118 //*****EX部分*****
119 EX ex(
120     .clk(clk),
121     .en(~StallE),
122     .clc(FlushE),
123     .PCD(PCD),
124     .InsD(Ins),
125     .ImmD(imm),
126     .rdD(rd),
127     .rs1D(rs1D),
128     .rs2D(rs2D),
129     .rs1_valD(rs1_valD),
130     .rs2_valD(rs2_valD),
131     .RegWriteD(RegWrite),
132     .ALUSrcD(ALUSrc),
133     .BranchD(Branch),
134     .JumpD(Jump),
135     .MemReadD(MemRead),
136     .MemWriteD(MemWrite),
137     .MemtoRegD(MemtoReg),
138     .CtrlD(CtrlD),
139     .luiD(lui),
140     .LoadD(Load),
141
142     .PCE(PCE),
143     .InsE(InsE),

```

```

144     .ImmE(ImmE) ,
145     .rdE(rdE) ,
146     .rs1E(rs1E) ,
147     .rs2E(rs2E) ,
148     .rs1_valE(rs1_valE) ,
149     .rs2_valE(rs2_valE) ,
150     .RegWriteE(RegWriteE) ,
151     .ALUSrcE(ALUSrcE) ,
152     .BranchE(BranchE) ,
153     .JumpE(JumpE) ,
154     .MemReadE(MemReadE) ,
155     .MemWriteE(MemWriteE) ,
156     .MemtoRegE(MemtoRegE) ,
157     .CtrlE(CtrlE) ,
158     .luiE(luiE) ,
159     .LoadE(LoadE)
160 );
161 Alu ALU1(
162     .a_val(PCE) ,
163     .b_val(ImmE) ,
164     .ctrl(0) ,
165     .result(JalTarge)
166 );
167 /*ALU的1号操作数：
168 1. 前送的MEM阶段ALU计算结果
169 2. 来自Load阶段前送的将要WB的内存值(已加bubble)
170 3. 来自EX阶段的rs1_valD*/
171 /*ALU的2号操作数：
172 1. 前送的MEM阶段ALU计算结果
173 2. 来自Load阶段前送的将要WB的内存值(已加bubble)
174 3. 来自EX阶段的rs2_valD
175 4.ImmE*/
176 Alu ALU(
177     .a_val(Op1) ,
178     .b_val(Op2) ,
179     .ctrl(CtrlE),      //操作类型
180     .result(ALUOutE) //ALU计算结果
181 );
182
183 assign JalrTarge = ImmE + Op1;
184
185 //判断Branch是否满足条件，返回ZeroE

```

```

186     Comperator cmp(
187         .a_val(Op1) ,
188         .b_val(Op2) ,
189         .ctrl(InsE[14:12]) ,
190         .result(ZeroE)
191     );
192
193 // ****MEM 分 ****
194 MEM Mem(
195     .clk(clk) ,
196     .en(~StallM) ,
197     .clc(FlushM) ,
198     .PCE(PCE) ,
199     .ALUOutE(ALUOutE) ,
200     .ImmE(ImmE) ,
201     .rdE(rdE) ,
202     .RegWriteE(RegWriteE) ,
203     .MemWriteE(MemWriteE) ,
204     .MemtoRegE(MemtoRegE) ,
205     .luiE(luiE) ,
206     .InsE(InsE) ,
207     .rs2_valE(rs2_valE) ,
208     .mem_w_data(mem_w_data) ,
209
210     .PCM(PCM) ,
211     .ALUOutM(ALUOutM) ,
212     .ImmM(ImmM) ,
213     .rdM(rdM) ,
214     .RegWriteM(RegWriteM) ,
215     .MemWriteM(MemWriteM) ,
216     .MemtoRegM(MemtoRegM) ,
217     .luiM(luiM) ,
218     .InsM(InsM) ,
219     .rs2_valM(rs2_valM) ,
220     .mem_w_dataM(mem_w_dataM)
221 );
222
223 always @(*) begin
224     if(luiM == 1)           ResultM <= ImmM;          // lui
225     else if(MemtoRegM == 2'b00) ResultM <= ALUOutM;    // ALU结果
226     else if(MemtoRegM == 2'b01) ResultM <= dmem_o_data; // Load
227     else if(MemtoRegM == 2'b10) ResultM <= PCM + 4;    // jal , jalr

```

```

228     else                         ResultM <= PCM;           // auipc
229   end
230
231   assign dmem_wen    = MemWriteM;    // 写入使能
232   assign dmem_i_data = mem_w_dataM; // Store Data
233   assign dmem_addr   = ALUOutM;      // 读取与写入地址均是ALUOutM
234 /* *****WB部分***** */
235 WB(
236   .clk ( clk ) ,
237   .en (~StallW) ,
238   .clc (FlushW) ,
239   .ResultM (ResultM) ,
240   .rdM(rdM) ,
241   .RegWriteM(RegWriteM) ,
242   .MemtoRegM(MemtoRegM) ,
243   .PCM(PCM) ,
244   .InsM(InsM) ,
245
246   .ResultW (ResultW) ,
247   .rdW(rdW) ,
248   .RegWriteW(RegWriteW) ,
249   .MemtoRegW(MemtoRegW) ,
250   .PCW(PCW) ,
251   .InsW(InsW)
252 );
253
254
255 /* *****Hazard部分***** */
256 Stall stall(
257   .rst (rst) ,
258   .BranchE (BranchE & ZeroE) ,
259   .JumpD(JumpD) ,
260   .JumpE(JumpE) ,
261   .rs1D(rs1D) ,
262   .rs2D(rs2D) ,
263   .rs1E(rs1E) ,
264   .rs2E(rs2E) ,
265   .rdE(rdE) ,
266   .rdM(rdM) ,
267   .rdW(rdW) ,
268   .MemtoRegE(MemtoRegE) ,
269   .RegWriteM(RegWriteM) ,

```

```

270     .RegWriteW(RegWriteW) ,
271     .StallF(StallF) ,
272     .StallD(StallD) ,
273     .StallE(StallE) ,
274     .StallM(StallM) ,
275     .StallW(StallW) ,
276     .FlushF(FlushF) ,
277     .FlushD(FlushD) ,
278     .FlushE(FlushE) ,
279     .FlushM(FlushM) ,
280     .FlushW(FlushW) ,
281
282     .PCE(PCE) ,
283     .ImmE(ImmE) ,
284     .ResultM(ResultM) ,
285     .rs1_valE(rs1_valE) ,
286     .rs2_valE(rs2_valE) ,
287     .DataWB(ResultW) ,
288     .ALUSrcE(ALUSrcE) ,
289     .op1(Op1) ,
290     .op2(Op2) ,
291     .mem_w_data(mem_w_data)
292 );
293
294 // *****VGA Assignment*****
295 `VGA_DBG_Core_Assignments
296 endmodule

```

### 3.9 后续操作

**综合** 选择左侧面板的 Run Synthesis 或者点击上方的绿色小三角，选择 Synthesis

**实现** 选择左侧面板的 Run Implementation 或者点击上方的绿色小三角，选择 Implementation。值得注意的是执行 implementation 之前应该确保引脚约束存在且正确，同时之前已经综合过最新的代码。

**生成二进制文件** 选择左侧面板的 Generate Bitstream 或者点击上方的绿色二进制标志。同时生成 Bitstream 前要确保：之前已经综合、实现过最新的代码。如没有，直接运行会默认从综合、实现开始。此过程还要注意生成的 bit 文件默认存放在.runs 下相应的 implementation 文件夹中

**烧写上板** 点击左侧的 Open Hardware Manager → 点击 Open Target → Auto Connect → 点击 Program Device → 选择 bistream 路径，烧写。验证结果见实验结果部分。

## 4 实验结果与分析

经历综合实现，我们可以在开发板上看到测试的结果我们分以下几个部分验证实验结果的正确性

**完整 CPU 执行路径验证** 在上板过程中，整个 CPU 的执行路径如下所示，根据测试代码可以判断跳转执行行为完全正确

```
1 0x000 - 0x070
2 0x078 - 0x080
3 0x088 - 0x090
4 0x098 - 0x0a4
5 0x0ac - 0x0b0
6 0xa8 - 0xa4
7 loop
```

**最终寄存器值的验证** 将根据测试代码得到的寄存器值与实际上板的最终结果进行对比，发现完全一致，基本认定加入部分执行正常



The screenshot shows the RV32I Pipelined CPU debugger interface. It displays a register dump and pipeline status. The register dump includes fields for PC, Inst, Valid, Ra, Sp, Gp, Tp, S0, S1, A1, A2, A3, A4, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, T3, T4, T5, and T6. The pipeline status shows stages If, Id, Ex, Ma, and Mb with their respective PC, Inst, Valid, and control signals (rd, rs1, rs2, rs1\_val, rs2\_val, reg\_wen, mem\_wen, mem\_ren, is\_branch, is\_jal, is\_jalr, is\_lui, alu\_ctrl, cmp\_ctrl).

图 4.7: 最终寄存器结果

**Bubble 插入验证** 在测试代码中，会产生 RAW 情形的 bubble 的两条代码为例

```

1 lw      s0 , 0(zero)      # s0 = 0x1234_5678 0x48
2 slli    s0 , s0 , 1       # s0 = 0x2468_acf0 0x4c

```

当 0x48(也即 Load) 指令到达 EX 阶段, 如果不产生 bubble 的话, 下一条 slli 指令就不能得到正确的结果, 所以此时会让 ID, IF 阶段 Stall 一个周期, 并且清空 EX 阶段以产生一个 bubble

```

RV32I Pipelined CPU

=====
pc: 00000050 inst: 00002223
=====
pc: 0000004c inst: 00141413 valid: 0
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 0000000c
a0: 00000007 a1: 00000001 a2: 00000001 a3: 0000000f a4: 00000007
a5: 00000006 a6: 0000001c a7: 00000007 s2: 00000015 s3: 00000000
s4: 00000001 s5: 00000008 s6: 00000003 s7: 00000007 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000
=====
pc: 00000048 inst: 00002403 valid: 1
rd: 08 rs1: 00 rs2: 00 rs1_val: 00000000 rs2_val: 00000000 reg_wen: 1
is_imm: 1 imm: 00000000
mem_wen: 0 mem_ren: 1 is_branch: 0 is_jal: 0 is_jalr: 0
is_auipc: 0 is_lui: 0 alu_ctrl: 0 cmp_ctrl: 0
=====
pc: 00000044 inst: 00c57c33 valid: 1
rd: 18 reg_wen: 1 mem_w_data: 00000001 alu_res: 00000001
mem_wen: 0 mem_ren: 0 is_jal: 0 is_jalr: 0
=====
pc: 00000040 inst: 00c56bb3 valid: 1
rd: 17 reg_wen: 1 reg_w_data: 00000007

```

图 4.8: RAW 的判断阶段

插入 bubble 后的结果 (也即下一周期的结果如下所示)

```

RV32I Pipelined CPU

=====
pc: 00000050 inst: 00002223
=====
pc: 0000004c inst: 00141413 valid: 1
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 0000000c
a0: 00000007 a1: 00000001 a2: 00000001 a3: 0000000f a4: 00000007
a5: 00000006 a6: 0000001c a7: 00000007 s2: 00000015 s3: 00000000
s4: 00000001 s5: 00000008 s6: 00000003 s7: 00000007 s8: 00000001
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000
=====
pc: 00000000 inst: 00000000 valid: 1
rd: 00 rs1: 00 rs2: 00 rs1_val: 00000000 rs2_val: 00000000 reg_wen: 0
is_imm: 0 imm: 00000000
mem_wen: 0 mem_ren: 0 is_branch: 0 is_jal: 0 is_jalr: 0
is_auipc: 0 is_lui: 0 alu_ctrl: 0 cmp_ctrl: 0
=====
pc: 00000048 inst: 00002403 valid: 1
rd: 08 reg_wen: 1 mem_w_data: 00000000 alu_res: 00000000
mem_wen: 0 mem_ren: 0 is_jal: 0 is_jalr: 0
=====
pc: 00000044 inst: 00c57c33 valid: 1
rd: 18 reg_wen: 1 reg_w_data: 00000001

```

图 4.9: RAW 插入 bubble

可见插入 bubble 的功能实现正常。

**Branch No Taken 验证** 我们以很典型的一段代码的执行情况为例来说明此功能的实现

```

1 blt      a0 , a1 , blt_target #0x8c
2 beq      zero , zero , end_b #0x90

```

其中 0x8c 指令是不会跳转的，而 0x90 会发生跳转。当 0x8c 指令到达 EX 阶段时，会让它及后面的指令正常流过，不会产生清除操作

```

RV32I Pipelined CPU
=====
pc: 00000094 inst: 00168693
===== If =====
pc: 00000090 inst: 00000463 valid: 1
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 2468acf0 s1: 2468acf2
a0: 2468acf0 a1: db975310 a2: 00000001 a3: 00000001 a4: 00000007
a5: 00000006 a6: 0000001c a7: 00000007 s2: 00000015 s3: 000e0000
s4: 00000001 s5: 00000008 s6: 00000003 s7: 00000007 s8: 00000001
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000
===== Id =====
pc: 0000008c inst: 00b54463 valid: 1
rd: 08 rs1: 0a rs2: 0b rs1_val: 2468acf0 rs2_val: db975310 reg_wen: 0
is_imm: 0 imm: 00000008
mem_wen: 0 mem_ren: 0 is_branch: 0 is_jal: 0 is_jalr: 0
is_auipc: 0 is_lui: 0 alu_ctrl: 1 cmp_ctrl: 0
===== Ex =====
pc: 00000088 inst: 00168693 valid: 1
pc: 00000090 inst: 00000463 valid: 1
rd: 0d reg_wen: 1 mem_w_data: 00000000 alu_res: 00000002
mem_wen: 0 mem_ren: 0 is_jal: 0 is_jalr: 0
===== Ma =====
pc: 00000088 inst: 00168693 valid: 1
rd: 0d reg_wen: 1 mem_w_data: 00000002
===== Wb =====
pc: 00000000 inst: 00000000 valid: 1
rd: 00 reg_wen: 0 reg_w_data: 00000000
=====

RV32I Pipelined CPU
=====
pc: 00000098 inst: 00000013
===== If =====
pc: 00000094 inst: 00168693 valid: 1
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 2468acf0 s1: 2468acf2
a0: 2468acf0 a1: db975310 a2: 00000001 a3: 00000001 a4: 00000007
a5: 00000006 a6: 0000001c a7: 00000007 s2: 00000015 s3: 000e0000
s4: 00000001 s5: 00000008 s6: 00000003 s7: 00000007 s8: 00000001
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000
===== Id =====
pc: 00000090 inst: 00000463 valid: 1
rd: 08 rs1: 00 rs2: 00 rs1_val: 00000000 rs2_val: 00000000 reg_wen: 0
is_imm: 0 imm: 00000008
mem_wen: 0 mem_ren: 0 is_branch: 1 is_jal: 0 is_jalr: 0
is_auipc: 0 is_lui: 0 alu_ctrl: 1 cmp_ctrl: 0
===== Ex =====
pc: 00000090 inst: 00000463 valid: 1
rd: 08 rs1: 00 rs2: 00 rs1_val: 00000000 rs2_val: 00000000 reg_wen: 0
is_imm: 0 imm: 00000008
mem_wen: 0 mem_ren: 0 is_branch: 1 is_jal: 0 is_jalr: 0
is_auipc: 0 is_lui: 0 alu_ctrl: 1 cmp_ctrl: 0
===== Ma =====
pc: 0000009c inst: 00b54463 valid: 1
rd: 08 reg_wen: 0 mem_w_data: db975310 alu_res: 48d159e0
mem_wen: 0 mem_ren: 0 is_jal: 0 is_jalr: 0
===== Wb =====
pc: 00000088 inst: 00168693 valid: 1
rd: 0d reg_wen: 1 reg_w_data: 00000002
=====
```

图 4.10: 不发生跳转

当 0x90 指令到达 EX 阶段时会发生跳转，此时下一时钟周期 ID，EX 阶段均会被清空，并且 IF 阶段 PC 地址变为跳转的目标地址。

```

RV32I Pipelined CPU
=====
pc: 00000098 inst: 00000013
===== If =====
pc: 00000094 inst: 00168693 valid: 1
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 2468acf0 s1: 2468acf2
a0: 2468acf0 a1: db975310 a2: 00000001 a3: 00000001 a4: 00000007
a5: 00000006 a6: 0000001c a7: 00000007 s2: 00000015 s3: 00000000
s4: 00000001 s5: 00000008 s6: 00000003 s7: 00000007 s8: 00000001
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000
===== Id =====
pc: 00000090 inst: 00000463 valid: 1
rd: 08 rs1: 00 rs2: 00 rs1_val: 00000000 rs2_val: 00000000 reg_wen: 0
is_imm: 0 imm: 00000008
mem_wen: 0 mem_ren: 0 is_branch: 1 is_jal: 0 is_jalr: 0
is_auipc: 0 is_lui: 0 alu_ctrl: 1 cmp_ctrl: 0
===== Ex =====
pc: 0000008c inst: 00b54463 valid: 1
rd: 08 reg_wen: 0 mem_w_data: db975310 alu_res: 40d159e0
mem_wen: 0 mem_ren: 0 is_jal: 0 is_jalr: 0
===== Ma =====
pc: 00000088 inst: 00168693 valid: 1
rd: 0d reg_wen: 1 reg_w_data: 00000002
===== Wb =====

```

图 4.11: 发生跳转

## 5 讨论与心得

在本次实验中，我遇到了许多设计上的问题，或者设计上值得优化的地方。

**Double Bump 问题** 在此次实验初期，我只考虑到了数据冒险与控制冒险，没有考虑到结构竞争。因为我一直没有注意到在同一时钟沿同时写入与读取一个寄存器，读取的值并不会是写入的新值，而是之前的旧值。我采用了 double bump 的方式解决结构竞争，而不是插入一个 bubble 来降低效率：在时钟的下降沿写入数据。

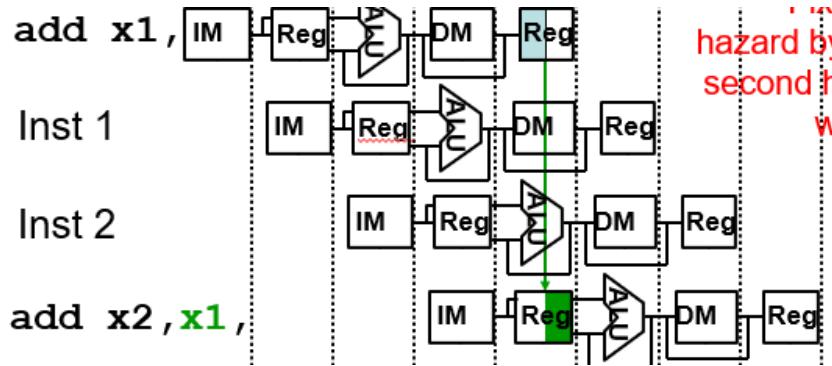


图 5.12: Double Bump

**Forward 条件问题** 在 Forward 模块中，我认为单纯判断目的寄存器是否是两个值寄存器之一这样的判断条件是不完善的，因为有些指令并没有实际的 rs1 或者 rs2 部分，解码（指令的对应位）出来的 rs1 与 rs2 很有可能正好与 rd 相同，从而产生错误的前送。在此次实验中，store 指令就会产生这样的问题，但我采用了遇到 Imm 的使用就跳过前送的策略规避了错误，可是更好的做法应该是设置两个信号，分别表示 rs1, rs2 是否被用到，只有用到了才会开始判断前送。