实验六 CPU 综合设计

一、实验目的

1 掌握复杂系统设计方法。

2 深刻理解计算机系统硬件原理。

二、实验内容

1）设计一个基于 MIPS 指令集的 CPU，支持以下指令：{add, sub, addi, lw, sw, beq, j, nop}；

2）CPU 需要包含寄存器组、RAM 模块、ALU 模块、指令译码模块；

3）该 CPU 能运行基本的汇编指令；（D~C+）

以下为可选内容：

4）实现多周期 CPU（B-~B+）；

5）实现以下高级功能之一（A-~A+）：

(1)实现 5 级流水线 CPU；

(2)实现超标量；

(3)实现 4 路组相联缓存；

可基于 RISC V 、ARM 指令集实现。

如发现代码为抄袭代码，成绩一律按不及格处理。

三、实验要求

编写相应测试程序，完成所有指令测试。

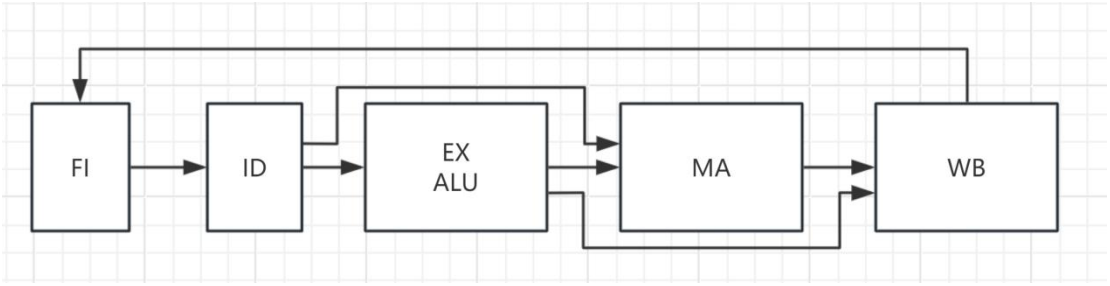
四、实验代码及结果

本实验报告包含两个 CPU 设计，包含 1 个简易的单周期 CPU 和 1 个高级功能 CPU，单周期 CPU 设计所使用的部件为先前所有实验中出现过的 ALU CU RegFile IMem 本报告中主要围绕高级功能 CPU 的设计进行阐述。

1. 高级功能 CPU 设计目标

实现 5 级流水线，实现双发射超标量，支持每条流水线单寄存器旁路数据转发，使用静态分支预测的总是不采用策略。

5 级流水线数据通路基本示意图：

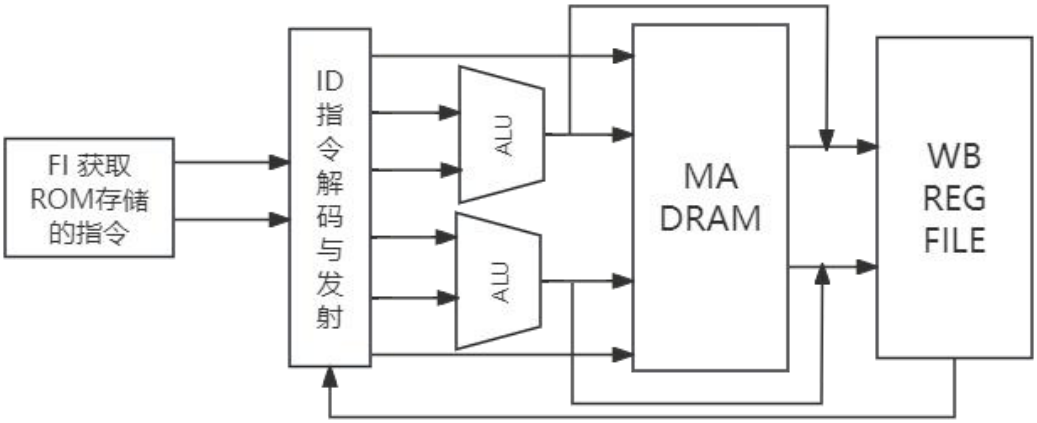


为支持双发射超标量，对于各阶段器件处理性能要求有更改：

1. 对于 FI 指令获取阶段，要求 ROM 支持同时获取并向下级交付连续两条指令。
2. 对于 ID 阶段，需要双发射单元。
3. 对于 EX 阶段，需要使用双 ALU，需注意同级两个流水线之间的信号传递。
4. 对于 MA 阶段，需要在 1 个时钟周期内完成两次读-写操作，并封装，本次设计中采用 DRAM 设计思想，在时钟上升沿和下降沿同时触发操作，RAM 内部使用异步操作，使用 inout 类型实现数据复用，更符合实际场景。
5. 对于 WB 阶段，对于 RegFile 的设计要求是支持 4 读 2 写，其中读寄存器异步，写寄存器时钟同步。

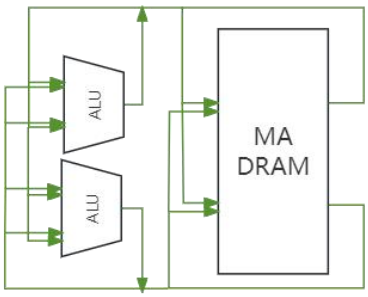
2.高级功能 CPU 设计思路

1. 流水线基本数据通路设计



如图中展示内容，以上本次设计的高级功能 CPU 使用的主要数据通路，整体来看是由两条流水线构成，在代码组织方式上，则是以流水线 5 级方式分别编写，每级流水线内部有两路设计。

2. 流水线旁路数据转发设计

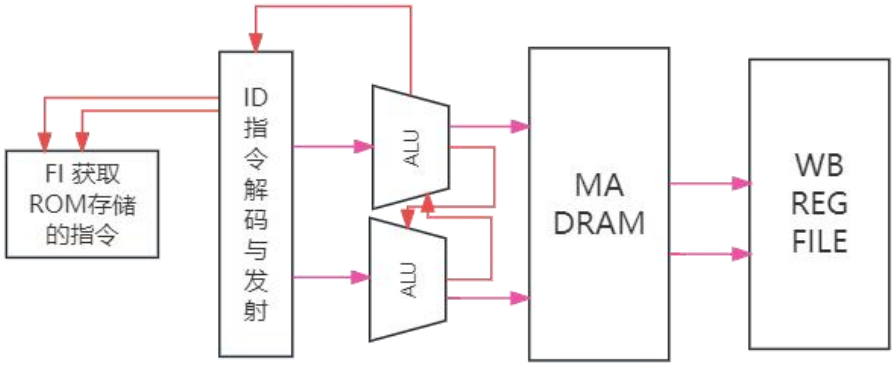


为提高流水线性能和超标量性能，设计了旁路数据转发通道，由于双发射两条流水线的设计比较复杂，所以在转发时，如果一个指令需要读取两个寄存器都正在被写入，则无法进行转发，如果只有 1 个要读的寄存器正在被写入，则会尽力而为进行转发，这种场景只有 1

个情况需要插入 1 个气泡。

3. 流水线级间控制信号传输设计

如图所示，流水线主要信号通过流水线寄存器逐级保存并传输（粉色箭头），级间传输的信号只要有两个类型，1 个是分支预测失败信号（EX 向 ID 传输的信号），另 1 个是更改取指地址的信号，由 ID 根据发射情况，分支预测情况，向 FI 传输。



4. 流水线控制信号格式设计

如下图所示是该 CPU 主要控制信号设计，控制信号含义在备注或名称中展示，这些信号直接由每级流水线内部使用，具体使用规则请见每级流水线内部设计。

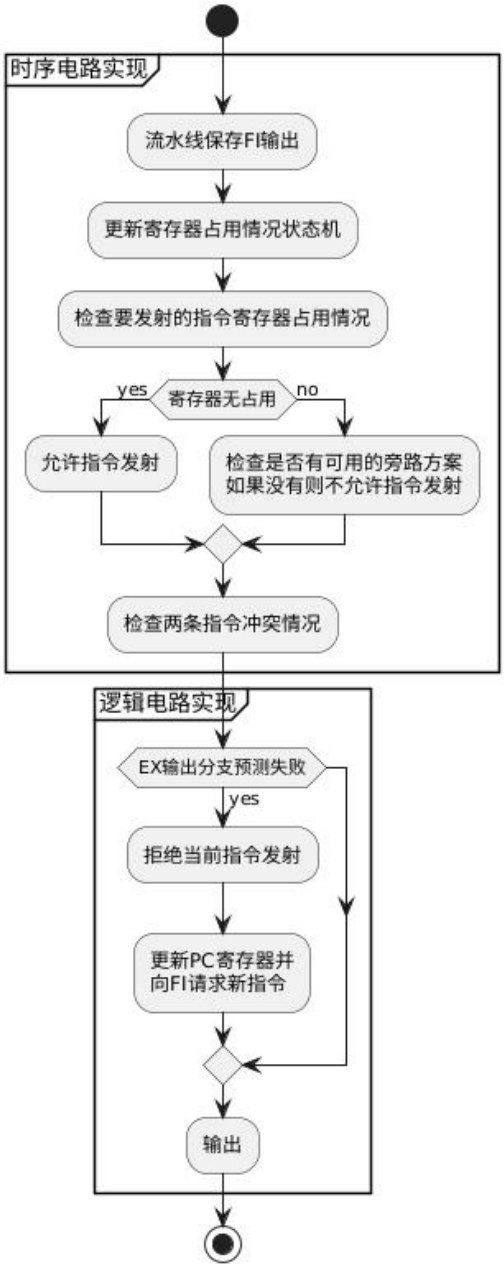
位置码	31-30	29	28-27	26-25	24-21	20	19	18
内容	PCSRC	exsign	ALUSRCA	ALUSRCB	ALUOP	CS	DSRC	R_W
长度	2	1	2	2	4	1	1	1
无状态计算			x	x			x	
备注		符号扩展				DRAM使能	DRAM使用旁路	DRAM读/写

17	16-12	11-8	7-0
WE	WTARGET	旁路方案	保留
1	5	4	8
		x	
寄存器写使能	目标写入寄存器		

下图是旁路数据前推的方案设计，整体来说存在基本的 6 种情况，同时由于有两条流水线，这俩爹旁路方案有 12 种情况，所以使用 4 位编码进行表示，这里有一些可以改进的地方，实际上可以针对两个寄存器的占用情况进行分开编码，但是问题就是方案和代码的复杂程度会进一步提升。

			EX旁路线1 SRCA	EX旁路线2 SRCB	DRAM旁路线 DSRC
旁路方案	x000	不使用旁路	x	x	x
	x001	MA->sw_data	x	x	x
	x010	MA->ALU1(by_src0)	11	x	x
	x011	MA->ALU2(by_src1)	x	11	x
	x100	ALU->ALU1	11	x	x
	x101	ALU->ALU2	x	11	x
	x110	MA->MA	x	x	1

5. 双发射超标量使用的 ID 阶段主要流程



特别的，对于控制信号输出也使用逻辑电路实现，这里省略，特别的，ALUSRCA ALUSRCB

DSRC 这 3 部分信号需要同时根据指令和前部分时序电路确定的旁路方案进行判断，在图中已省略。

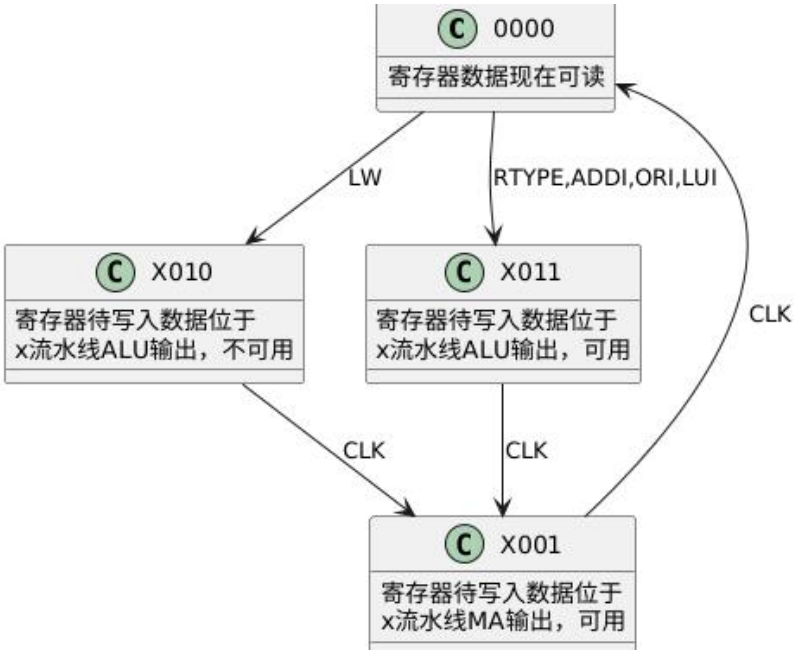
6. 寄存器占用情况状态机

为了判断每个寄存器的可用情况，在 ID 流水线内部和维护了一组寄存器状态机寄存器，由于指示当前寄存器是否可用，状态机的状态含义以及状态转移方法由图中所示。

特别的，状态定义的最高位 X 代表当前寄存器正在被写入的数据在哪一条流水线上，0 代表流水线 1，1 代表流水线 2。

reg_w_status

定义	x000	可用
	x010	当前数据在ALU，不可用
	x011	当前数据在ALU，可用
	x001	当前数据在MA，可用



7. ALU 设计

ALU 使用了先前实验中所使用的 ALU，同时满足了加法单元使用基本逻辑门（特别地，ADD 使用 generate 块完成）。

```
module ADD (F,CF,A,B,EN);
    input EN;
    parameter SIZE = 32;
    output reg [SIZE-1:0] F;
    input wire [SIZE-1:0] A,B;
    output reg CF;
    wire [SIZE-1:0] c;
    wire [SIZE-1:0] Fw;
    wire CFw;
    genvar  i;

    always @(*) begin
        if(EN==1) begin
            {CF,F} <= A+B;
        end
        else begin F<=32'bz;
            CF <= 1'bz;
        end
    end

    generate
        for (i=0;i<=31 ;i=i+1 ) begin
            case (i)
                0: fulladder_bit fa0(Fw[0], c[0],A[0],B[0],0);
                SIZE-1 :fulladder_bit fah(Fw[SIZE-1], CFw ,A[SIZE-1],B[SIZE-1],c[SIZE-2]);
                default: fulladder_bit fa(Fw[i],c[i],A[i],B[i],c[i-1]);
            endcase
        end
    endgenerate

endmodule
```

8. MA 模块设计

```
always @(posedge CLK or negedge CLK or posedge RST) begin
    ram_RW = 1;
    ram_cs = 0;

assign ram_data_inout = (~ram_RW) ? ram_in_data : 32'bz;
assign Data2out = ram_data_inout;
        if(CLK) begin
            //时钟上升沿

            //非阻塞式保存下级时钟流水线ASAP
            if(DSRC2)   tmp_in_data2 <= bypass2;
            else        tmp_in_data2 <= Data2in;
            //tmp_in_data2 <= (DSRC2)  bypass2 : Data2in;
            tmp_in_CS2 <= CS2;
            tmp_R_W2 <= R_W2;
            tmp_addr2 <= Addr2;
            //end

            //保存并输入来自上级流水线内容
            if(~R_W1)
            if(DSRC1)   ram_in_data <= bypass1; //如果写入
            else        ram_in_data <= Data1in; //如果写入
            else ram_in_data <= 0;

            ram_addr = Addr1;   //注意数据竞争，使用阻塞式，防止错误写入数据
            ram_cs = CS1;
            ram_RW = R_W1;
            //end
        end
        else begin
            //时钟下降沿
            Data1out <= ram_data_inout;

            ram_addr = tmp_addr2;

            ram_RW = tmp_R_W2;
            ram_cs = tmp_in_CS2;
            if(~R_W2) ram_in_data <= tmp_in_data2; //如果写入
            else ram_in_data = 0;
        end
    end
end
```

这里使用了 DRAM 的设计思路，主要是一个时钟内在上升沿和下降沿都能传输数据，

避免了接入一个 2x 时钟的麻烦，同时，这里实际上可以通过流水线停顿信号，加入 Cache 模块，就可以针对 RAM 实现更现实一些的读写方式。

3. CPU 仿真记录与分析

仿真程序记录：

```
`timescale 1ns/1ps

module sim_cpu ();



    reg CLK,RST;
    wire [5:0] ROM_A1,ROM_A2;
    wire [31:0] ROM_RD1,ROM_RD2;


    initial begin
        CLK=0;
        RST=1;
        fork
            repeat(2000) begin
                #10 CLK=~CLK;
            end
            begin
                #52 RST=0;
            end
        join
    end
    MIPS32_CPU cpu(
        CLK,RST,ROM_A1,ROM_A2,ROM_RD1,ROM_RD2
    );

    IMem ROM(
        ROM_A1,ROM_A2,ROM_RD1,ROM_RD2
    );
    single_cycle_CPU cpu2(CLK,RST);

endmodule
```

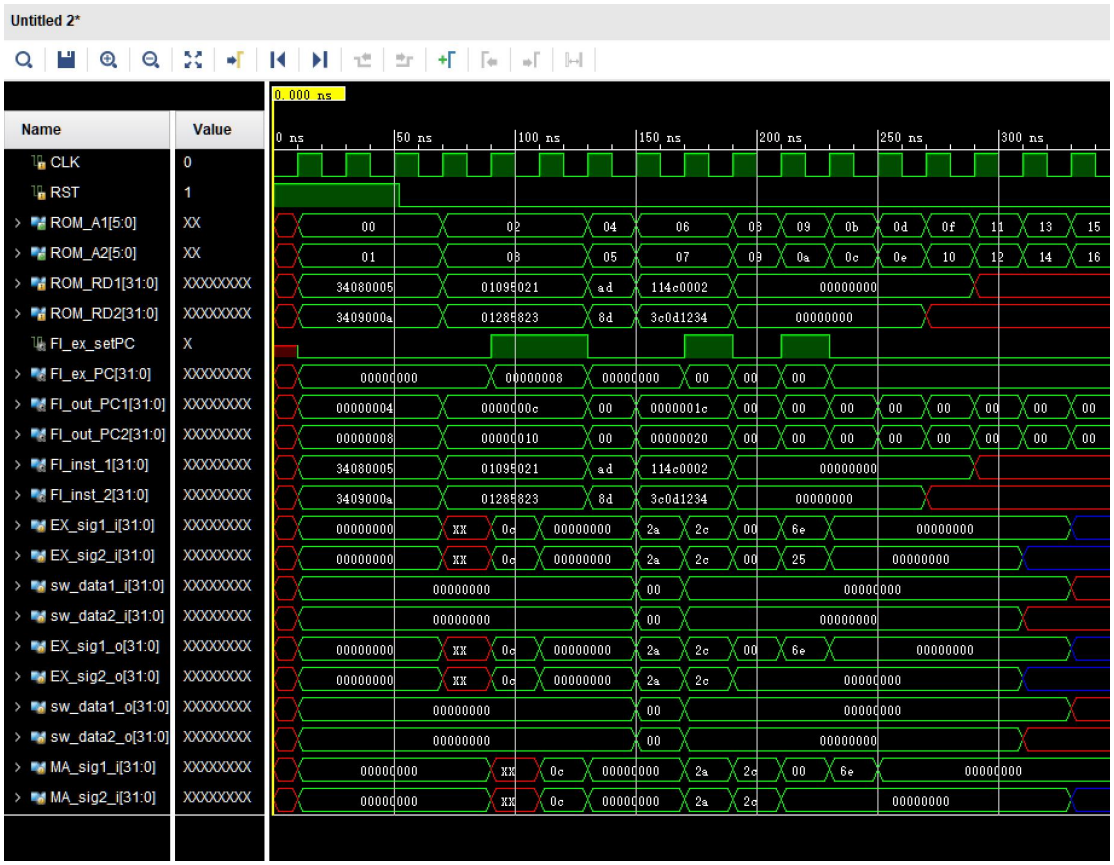
同时进行了高级功能 CPU 和单周期 CPU 的仿真，将 ROM 放在 CPU 外面的主要原因是：如果 ROM 写在了 CPU 里面，那么 CPU 在实现这一步中，opt_design 会认为这个 CPU 没有实际有效的输出和变化，于是将所有元器件清空。为了能正常测试这个高级功能 CPU 能不能通过 vivado 的综合仿真来排除所有的语法错误，所以我将 ROM 设计在了 CPU 的外面。而对于单周期 CPU 来说并不是本次实验的重点，仅进行了仿真测试。

仿真程序：

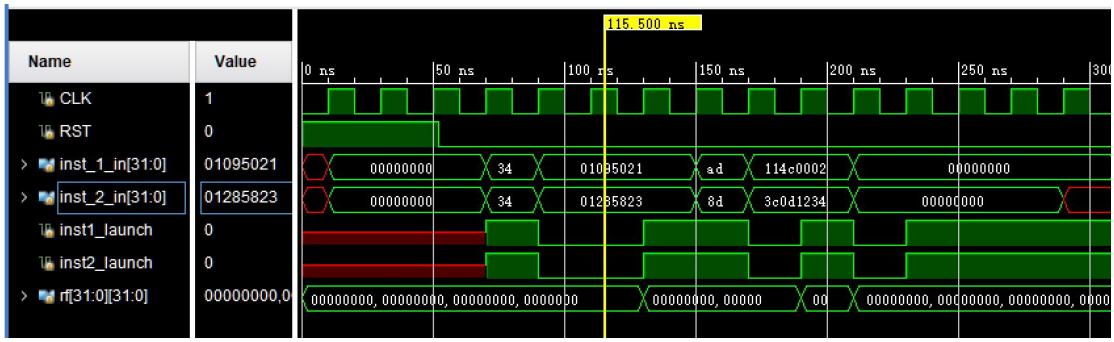
汇编程序以及对应的机器码如图所示：

```
ROM[0] = 32'h34080005;  // ori $t0, $zero, 5
ROM[1] = 32'h3409000A;  // ori $t1, $zero, 10
ROM[2] = 32'h01095021;  // addu $t2, $t0, $t1
ROM[3] = 32'h01285823;  // subu $t3, $t1, $t0
ROM[4] = 32'hAD0A0000;  // sw $t2, 0($t0)
ROM[5] = 32'h8D0C0000;  // lw $t4, 0($t0)
ROM[6] = 32'h114C0002;  // beq $t2, $t4, label
ROM[7] = 32'h3C0D1234;  // lui $t5, 0x1234
ROM[8] = 32'h00000000;  // nop
ROM[9] = 32'h00000000;  // nop (label 后续指令)
```

仿真波形运行结果：

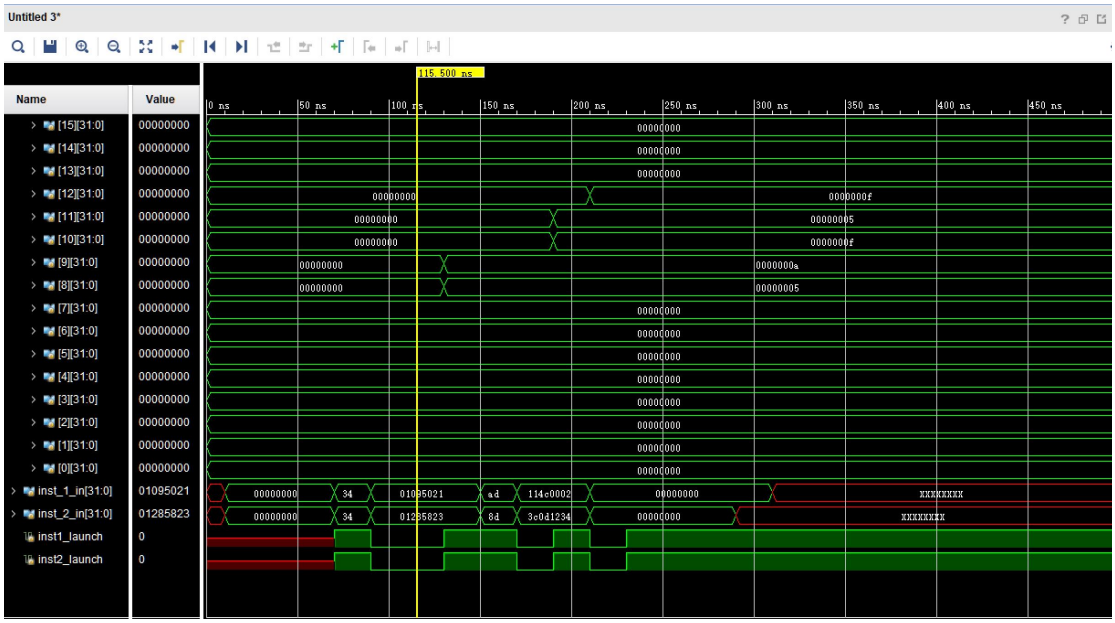


高级功能 CPU 中的信号和波形图过于复杂，我们重点分析指令发射情况和寄存器变化。



上图是指令发射情况，其中，inst1_in 是第一条指令内容，inst2_in 是第二条指令内容。

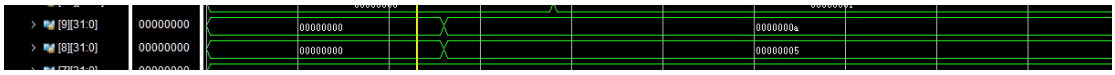
Inst1_launch 为 1 表示指令 1 成功发射，inst2_launch 为 1 则表示指令 2 成功发射，否则表示插入了空泡。



结合分析寄存器变化，我们可以确认运行结果的正确行，由于测试程序的依赖关系正好满足双发射，即不存在只发射 1 条指令的情况，所以寄存器变化基本也是成对的。

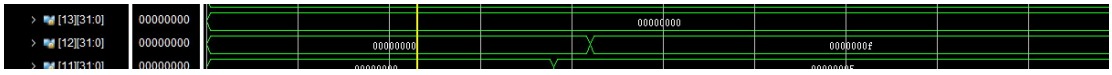
```
ROM[0] = 32'h34080005;  // ori $t0, $zero, 5
ROM[1] = 32'h3409000A;  // ori $t1, $zero, 10
ROM[2] = 32'h01095021;  // addu $t2, $t0, $t1
ROM[3] = 32'h01285823;  // subu $t3, $t1, $t0
ROM[4] = 32'hAD0A0000;  // sw $t2, 0($t0)
ROM[5] = 32'h8D0C0000;  // lw $t4, 0($t0)
ROM[6] = 32'h114C0002;  // beq $t2, $t4, label
ROM[7] = 32'h3C0D1234;  // lui $t5, 0x1234
ROM[8] = 32'h00000000;  // nop
ROM[9] = 32'h00000000;  // nop (label 后续指令)
```

观察仿真波形图，第一条指令运行结果是$t0 变为 5，我们可以看到对应的寄存器文件 rf[8] (代表$t0)成功变为了 5，第二条则是$t1 变为 10，我们可以看到对应的 rf[9]变为了 0xa（即 10 进制 10）。

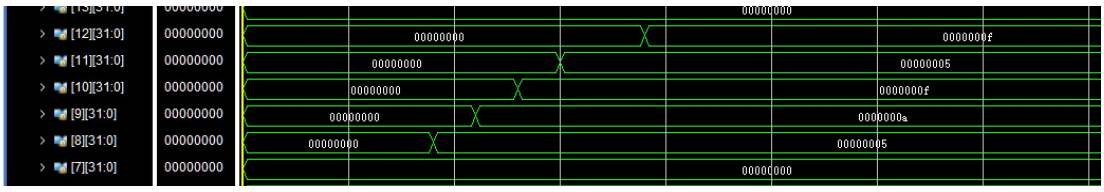


同理，第 3 条指令使得 t2 为 15，第 4 条指令使得 t3 为 5，可以看到波形图上都有对应变化。

后两条则是存入 RAM 并读出，使得 t4 变为了 15（与 t2 相同），和也是正确的。最后是一条分支指令，如果指令正确执行则跳过了 lui，可以看到这里的 t5 并没有执行，说明了 CPU 设计基本正确性。



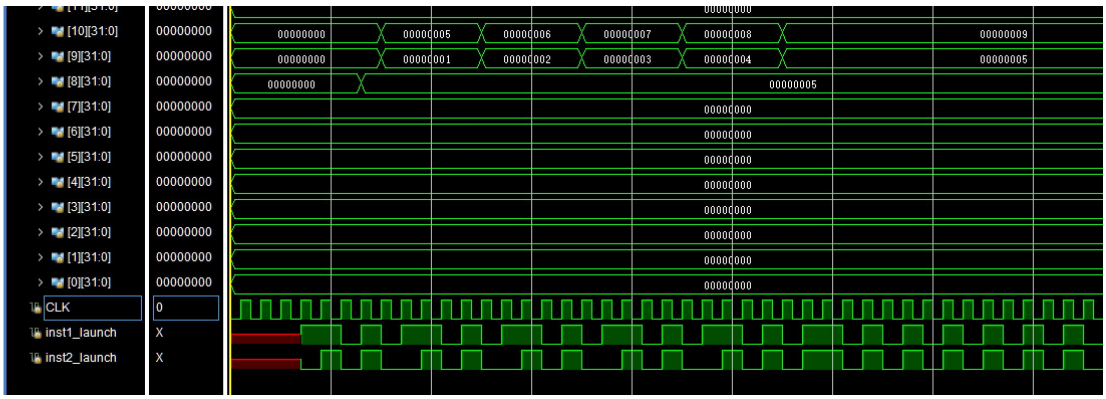
同理，对于单周期 CPU，也有类似的波形，说明实验中设计的简易单周期 CPU 正确性。



第二条测试程序也比较简单，是对 t1 进行循环累加，从 0 累加到 5（与 t0 相同），t2 则是记录每次 t0+t1（累加前的值）的结果，最终应该是从 5 累加到 9，程序结束后，使用

beq -1 的方式进入死循环。

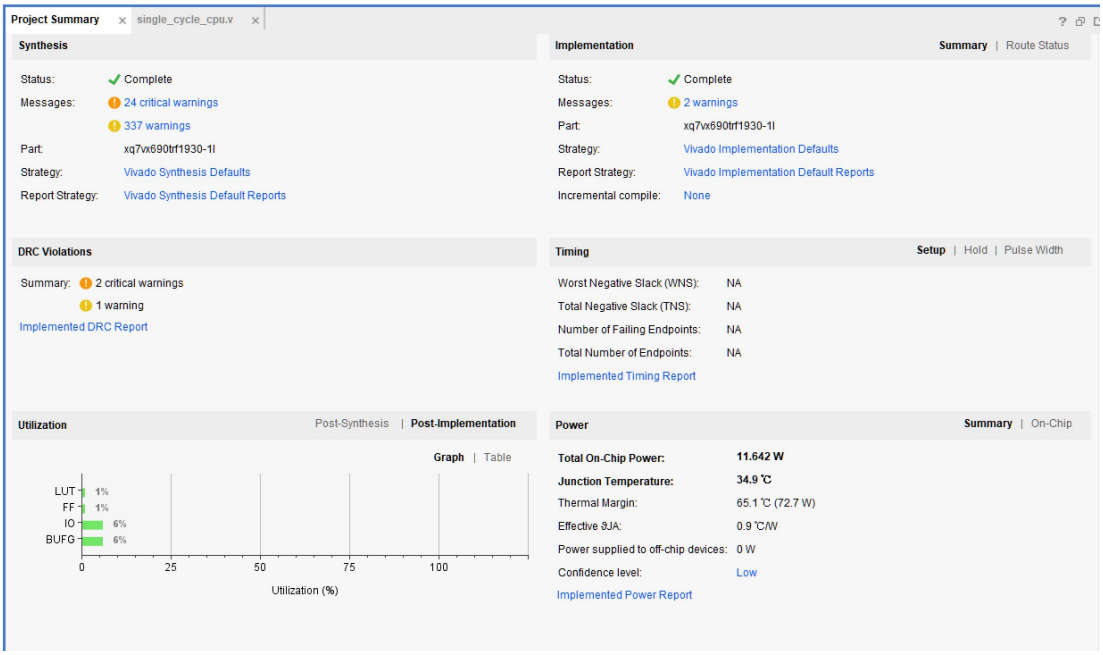
```
ROM[0] = 32'h20080005;
ROM[1] = 32'h01095020;
ROM[2] = 32'h21290001;
ROM[3] = 32'h11090002;
ROM[4] = 32'h08000000;
ROM[5] = 32'h00000000;
ROM[6] = 32'h00000000;
ROM[7] = 32'h00000000;
ROM[8] = 32'h00000000;
ROM[9] = 32'h1109FFFF;
ROM[10] = 32'h00000000;
ROM[11] = 32'h00000000;
ROM[12] = 32'h00000000;
// addi $t0  $zero 5 //循环次数
// addu $t2  $t0  $t1
// addi $t1 $t1 1
// beq $t0,$t1,2
// nop
// j 0
// beq $t0 $t1 -1
```



可以看到，高级功能 CPU 同样正确执行了这个测试程序，累加到 5 以后，进入死循环。虽然这里的指令一直不变，由于流水线性质，以及每次都分支预测失败，所以会有 1 周期的惩罚，所以可以看到，在进入死循环之后，都是隔 1 个周期发射两条指令（指令 2 是 nop）

4. CPU vivado RTL 综合与实现测试

本次设计的高级功能 CPU 可以正确通过综合与实现测试，但是使用了其他的开发板型号进行，主要是由于我们先前使用的 Ego1 的可编程电路的元器件不足，主要是因为本次 CPU 设计中，使用的 4Kx32 的 RAM 占用的空间过大，同时因为异步时序的原因，也不能将其规划到 FPGA 开发板中专门用于 RAM 的模块上面，导致了 LUT 器件的不足，但是换到更大的板子上，就可以正确通过测试，这说明我的设计在语法和设计合法性上，基本不存在问题。



五、调试和心得体会

这是我头一次写这么复杂 verilog 项目，对于高级功能 CPU 整体花了两天半（大约 20 小时）的时间，设计过程主要参考的是计算机组成原理教材第六章和第七章的内容，整体来说是经历了以下几个过程。1.深入学习教材第六第七章，理解流水线和控制单元相关内容，初步了解控制冒险和数据冒险。2.确定各个部件的性能指标(比如要求几读几写)。3.从 DRAM 入手，理解流水线中的时序电路和逻辑电路的关系，对于 verilog 中，阻塞赋值和非阻塞赋值有更深入的理解和感受。4.依次完成了 IMem RegFIle FI EX ID CPU 的模块。5.进行测试，并在先前的基础上，添加解决数据冒险的旁路数据前推功能，并进行更多程序的仿真测试，至此完成了本次实验的最终结果。

实验中也确实遇到了相当多的问题，但是基本都很快得到了解决，我认为像这样再有充分的知识储备的前提下，以及早期规划下进行设计是一个相当流畅的过程，我在 20 小时内就完成了这么一个设计，并且针对多个程序都确认了正确性，这基本上确保了我的设计没有很大的问题，同时对于宏的使用也避免了一些错误。

我觉得完成这么一个实验其实是一个很酷的事情，尤其是超标量听起来就很酷，甚至超标量这一部分在计组教材上都没有涉及。这次实验可以说我已经把计组书中对应的章节都翻烂了，对于 CPU 设计的基本方法和设计理念都有了一定的了解，虽然最终设计出的也只是一个玩具级别的代码，但是看到仿真波形正确，CPU 实现了预计的性能指标的时候，我还是非常的激动，实际上到了后面，尤其是在有一堆旁路数据转发的时候，这个 CPU 中很多的信号量我已经很难看出来正确结果了，旁路数据转发方案的正确性主要还是靠 verilog 行为级描述的正确性来保证，我觉得可以说我现在已经基本掌握了 verilog 硬件描述语言的使用和编写方法，做出了我自己认为值得满足的结果。

六、CPU 设计代码

ALU 模块与 ADD 加法器，单周期 CPU 使用的 CU，在先前实验中已经给出，本次实验中仅给出存在变动的代码。

宏定义

```
`define ALU_OP_LOGIC_LEFTSHIFT  4'b0111 // Result = X << Y  逻辑左移
`define ALU_OP_ARITH_RIGHTSHIFT 4'b1000 // Result = X >>> Y 算术右移
`define ALU_OP_LOGIC_RIGHTSHIFT 4'b1001 // Result = X >> Y  逻辑右移
`define ALU_OP_UNSIGNED_MULTIPLY 4'b1010 // Result = (X * Y)[31:0]; Result2 = (X * Y)[63:32] 无符号乘法
```

```
`define ALU_OP_UNSIGNED_DIVIDE   4'b1111 // Result = X / Y; Result2 = X % Y 无符号除法
`define ALU_OP_ADD              4'b0100 // Result = X + Y (Set OF/UOF)
`define ALU_OP_SUBTRACT         4'b0101 // Result = X - Y (Set OF/UOF)
`define ALU_OP_BITWISE_AND      4'b0000 // Result = X & Y   按位与
`define ALU_OP_BITWISE_OR       4'b0001 // Result = X | Y   按位或
`define ALU_OP_BITWISE_XOR      4'b0010 // Result = X ⊕ Y   按位异或
`define ALU_OP_BITWISE_NOR      4'b0011 // Result = ~(X | Y) 按位或非
`define ALU_OP_SIGNED_COMPARISON 4'b0110 // Result = (X < Y) ? 1 : 0 符号比较
`define ALU_OP_UNSIGNED_COMPARISON 4'b1100 // Result = (X < Y) ? 1 : 0 无符号比较
`define ALU_OP_LUI_16TO32          4'b1110
//所有指令{add, sub, addi,  lw, sw, beq, j, nop , ori , lui}
//以及 addu subu

//R-type
`define _OP_add  6'b100000
`define _OP_sub  6'b100010
`define _OP_addu 6'b100001
`define _OP_subu 6'b100011

`define _OP_addi 6'b001000
`define _OP_lw   6'b100011
`define _OP_sw   6'b101011
`define _OP_beq  6'b000100
`define _OP_j    6'b000010
`define _OP_ori  6'b001101
`define _OP_lui  6'b001111
```

单周期 CPU

```
`include "macro.v"

`timescale 1ns/1ps


module single_cycle_CPU (
    input CLK,RST
);

    reg [31:0] inst;
    reg [31:0] inst_PC,inst_PC__;
    wire [31:0] inst__;
    wire ALU_ZF;
    wire MemToReg,MemWrite,PCSrc,ALUSrc,RegDst,RegWrite,Jump;
    wire [2:0] ALUControl;
    reg now_RST;

    always @(posedge CLK) begin
        if(RST) begin
            now_RST = 1;
            inst_PC <= 0;
            inst <= 0;
        end
        else begin
            now_RST = 0;
            inst <= inst__;
            inst_PC <= inst_PC__;
        end
    end

    IMem        ROM1(.A1(inst_PC__[31:2]),.RD1(inst__));

    Controller  cu1(.Op(inst[31:26]),
                    .Funct(inst[5:0]),
                    .Zero(ALU_ZF),
                    .MemToReg(MemToReg),
                    .MemWrite(MemWrite),
                    .PCSrc(PCSrc),
                    .ALUSrc(ALUSrc),
                    .RegDst(RegDst),
                    .RegWrite(RegWrite),
```

```
                .Jump(Jump),
                .ALUControl(ALUControl)
                );

    always @(*) begin
        if(now_RST) inst_PC__ <= 0;
        else
        if(Jump) begin
            inst_PC__ <= {inst_PC[31:28],inst[27:0],4'b0};
        end
        else if(PCSrc) begin
            if(inst[15])
                inst_PC__ <= inst_PC  + {12'b1,inst[15:0],4'b0};
            else
                inst_PC__ <= inst_PC  + {12'b0,inst[15:0],4'b0};
        end
        else
            inst_PC__ <= inst_PC+4;
    end

    reg  [31:0] ALU_A,ALU_B;
    reg W1E;
    reg [4:0] RAddr1,RAddr2,WAddr1;
    wire [31:0] RData1,RData2;
    reg [31:0] WData1;
    wire [31:0] ALU_F;

    always @(*) begin
        if(RegDst) begin
            RAddr1 <= inst[25:21];
            RAddr2 <= inst[20:16];
            ALU_A <= RData1;
            ALU_B <= RData2;
        end
        else if(inst[31:26]==`_OP_lui) begin
            RAddr1 <= inst[25:21];
            ALU_A <= RData1;
            ALU_B <= 12;
        end
        else if(ALUSrc) begin
            RAddr1 <= inst[25:21];
            ALU_A <= RData1;
            if(inst[15])
                ALU_B <= {16'b1,inst[15:0]};
            else
                ALU_B <= {16'b0,inst[15:0]};
        end
        else begin
            //beq
            RAddr1 <= inst[25:21];
            RAddr2 <= inst[20:16];
            ALU_A <= RData1;
            ALU_B <= RData2;
        end
    end




    ALU alu1(.OP({1'b0,ALUControl}),
            .A(ALU_A),
            .B(ALU_B),
            .F(ALU_F),
            .ZF(ALU_ZF)
            );

    RegFile regfile1(.CLK(CLK),
                    .RST(RST),
                    .W1E(W1E),
                    .RAddr1(RAddr1),
                    .RAddr2(RAddr2),
                    .RData1(RData1),
```

```
                    .RData2(RData2),
                    .WAddr1(WAddr1),
                    .WData1(WData1)
                    );

    wire [31:0] RAM_Data_wire;
    wire R_W;
    assign RAM_Data_wire = (MemWrite) ? RData2 : 16'bz;
    assign R_W = (MemWrite) ? 1'b0 : 1'b1;

    RAM_4Kx32_inout ram11(
        .Data(RAM_Data_wire),  // 数据输出
        .Addr({2'b0,ALU_F[31:2]}),   // 地址输入
        .Rst(RST),                     // 复位信号
        .R_W(R_W)                     // 读写信号
    );


    always @(*) begin
        if(RegWrite && RegDst) begin
            W1E <= 1;
            WAddr1 <= inst[15:11];
        end
        else if(RegWrite) begin
            W1E <= 1;
            WAddr1 <= inst[20:16];
        end
        else begin
            W1E <= 0;
            WAddr1 <= 0;
        end
    end

    always @(*) begin
        if(MemToReg) begin
            WData1 <= RAM_Data_wire;
        end
        else begin
            WData1 <= ALU_F;
        end
    end


endmodule
```

高级功能 CPU

```
`timescale 1ns/1ps


module MIPS32_CPU (
    input CLK,RST,
    //连接到外部的 ROM
    output [5:0] ROM_A1,
    output [5:0] ROM_A2,
    input [31:0] ROM_RD1,
    input [31:0] ROM_RD2
);
    wire FI_ex_setPC;
    wire [31:0] FI_ex_PC;
    wire [31:0] FI_out_PC1,FI_out_PC2,FI_inst_1,FI_inst_2;

    //流水线执行单元外的寄存器

    // EX pipeline

    //接受输入
    reg [31:0] EX_sig1_i,EX_sig2_i;
    reg [31:0] sw_data1_i,sw_data2_i;   //Reg to Mem
    //输出
```

```
    reg [31:0] EX_sig1_o,EX_sig2_o;
    reg [31:0] sw_data1_o,sw_data2_o;   //Reg to Mem
    // EX end

    // MA pipeline
    //接受输入
    reg [31:0] MA_sig1_i,MA_sig2_i;
    reg [31:0] MA_ALU1_TMP,MA_ALU2_TMP; //may ALU to Reg (skip MA)
    //输出
    reg [31:0] MA_sig1_o,MA_sig2_o;
    reg [31:0] MA_Data1_o,MA_Data2_o;   //chose ALU or MemR //旁路源
    // MA end

    // WB pipeline
    //输入
    reg [31:0] WB_sig1_i,WB_sig2_i; //处于 MA 阶段
    //输出
    reg [31:0] ID_clr_w_mask;

    // WB end

    //end


    FI_stage FI_1(
        .RST(RST),
        .CLK(CLK),

        .ROM_A1(ROM_A1),
        .ROM_A2(ROM_A2),
        .ROM_RD1(ROM_RD1),
        .ROM_RD2(ROM_RD2),

        .ex_setPC(FI_ex_setPC),
        .ex_PC(FI_ex_PC),
        .out_PC1(FI_out_PC1),
        .out_PC2(FI_out_PC2),
        .inst_1(FI_inst_1),
        .inst_2(FI_inst_2)
    );


    wire [4:0] reg_read_addr[0:3];
    wire [31:0] reg_read_data[0:3];
    wire EX_ex_set_PC;
    wire [31:0] EX_ex_PC;

    wire [31:0] IDo_src0,IDo_src1,IDo_src2,IDo_src3;
    wire [31:0] IDo_imm1,IDo_imm2;
    wire [31:0] IDo_PC1,IDo_PC2;
    //以下内容需要额外寄存器保存
    wire [31:0] IDo_sig1,IDo_sig2;
    wire [31:0] IDo_sw_data_1,IDo_sw_data_2;
    //内容需要额外寄存器保存 end

    ID_stage ID_1(
        .CLK(CLK),
        .RST(RST),
        .pipeline_hang(1'b0),
        .clr_w_mask(ID_clr_w_mask),

        .PC1(FI_out_PC1),
        .PC2(FI_out_PC2),
        .inst_1(FI_inst_1),
        .inst_2(FI_inst_2),

        .RAddr1(reg_read_addr[0]),
        .RAddr2(reg_read_addr[1]),
        .RAddr3(reg_read_addr[2]),
        .RAddr4(reg_read_addr[3]),
        .ID_set_PC(FI_ex_setPC),
```

```
        .ID_PC(FI_ex_PC),
        .RData1(reg_read_data[0]),
        .RData2(reg_read_data[1]),
        .RData3(reg_read_data[2]),
        .RData4(reg_read_data[3]),
        .EX_set_PC(EX_ex_set_PC),
        .EX_PC(EX_ex_PC),
        //.EX_clear_pipeline2(0),

        .src0(IDo_src0),
        .src1(IDo_src1),
        .src2(IDo_src2),
        .src3(IDo_src3),
        .imm1(IDo_imm1),
        .imm2(IDo_imm2),
        .PC1_o(IDo_PC1),
        .PC2_o(IDo_PC2),

        .inst1_signal(IDo_sig1),
        .inst2_signal(IDo_sig2),

        .sw_data_1(IDo_sw_data_1),
        .sw_data_2(IDo_sw_data_2)
    );

    reg [31:0] EX_bypass0,EX_bypass1,EX_bypass2,EX_bypass3;
    wire EX_clear_pipeline2;
    wire [31:0] EXo_ALU1,EXo_ALU2;

    EX_stage EX_1(
        .CLK(CLK),
        .RST(RST),

        .src0(IDo_src0),
        .src1(IDo_src1),
        .src2(IDo_src2),
        .src3(IDo_src3),
        .PC1(IDo_PC1),
        .PC2(IDo_PC2),
        .imm1(IDo_imm1),
        .imm2(IDo_imm2),
        .PCSRC1(IDo_sig1[31:30]),
        .PCSRC2(IDo_sig2[31:30]),
        .exsign1(IDo_sig1[29]),
        .exsign2(IDo_sig2[29]),
        .ALU1_SRCA(IDo_sig1[28:27]),
        .ALU1_SRCB(IDo_sig1[26:25]),
        .ALU2_SRCA(IDo_sig2[28:27]),
        .ALU2_SRCB(IDo_sig2[26:25]),
        .ALU_OP1(IDo_sig1[24:21]),
        .ALU_OP2(IDo_sig2[24:21]),
        .src_by_0(EX_bypass0),
        .src_by_1(EX_bypass1),
        .src_by_2(EX_bypass2),
        .src_by_3(EX_bypass3),

        .set_PC(EX_ex_set_PC),
        .ex_PC(EX_ex_PC),
        .clear_pipeline2(EX_clear_pipeline2),

        .ALU_out1(EXo_ALU1),
        .ALU_out2(EXo_ALU2)
    );

    reg [31:0] DRAM_bypass1,DRAM_bypass2;
    wire [31:0] MAo_data1,MAo_data2;
    DRAM DRAM_1(
        .CLK(CLK),
        .RST(RST),
        .Data1in(sw_data1_o),
        .Data2in(sw_data2_o),
        .Addr1(EXo_ALU1),
```

```
        .Addr2(EXo_ALU2),
        .R_W1(EX_sig1_o[18]),
        .R_W2(EX_sig2_o[18]),
        .CS1(EX_sig1_o[20]),
        .CS2(EX_sig2_o[20]),
        .DSRC1(EX_sig1_o[19]),
        .DSRC2(EX_sig2_o[19]),

        .Data1out(MAo_data1),
        .Data2out(MAo_data2),

        .bypass1(DRAM_bypass1),
        .bypass2(DRAM_bypass2)
    );


    RegFile MIPS32_REG(
        .CLK(CLK),
        .RST(RST),
        .W1E(MA_sig1_o[17]),
        .W2E(MA_sig2_o[17]),
        .RAddr1(reg_read_addr[0]),
        .RAddr2(reg_read_addr[1]),
        .RAddr3(reg_read_addr[2]),
        .RAddr4(reg_read_addr[3]),
        .WAddr1(MA_sig1_o[16:12]),
        .WAddr2(MA_sig2_o[16:12]),
        .WData1(MA_Data1_o),
        .WData2(MA_Data2_o),
        .RData1(reg_read_data[0]),
        .RData2(reg_read_data[1]),
        .RData3(reg_read_data[2]),
        .RData4(reg_read_data[3])
    );


    always @(posedge CLK) begin
        if(RST) begin
            //清空流水线输入寄存器
            EX_sig1_i <= 0; EX_sig2_i <= 0;
            sw_data1_i <= 0; sw_data2_i <=0;

            MA_sig1_i <= 0 ; MA_sig2_i <=0;
            MA_ALU1_TMP <= 0; MA_ALU2_TMP <= 0;

            WB_sig1_i <= 0; WB_sig2_i <= 0;

        end
        else begin
            //流水线交付 维护 swdata 旁路
            EX_sig1_i <= IDo_sig1;      EX_sig2_i <= IDo_sig2;

            if(IDo_sig1[10:8] == 3'b001)
                if(IDo_sig1[1] == 0)
                    sw_data1_i <= MA_Data1_o;
                else
                    sw_data1_i <= MA_Data2_o;
            else
                sw_data1_i <= IDo_src1;

            if(IDo_sig2[10:8] == 3'b001)
                if(IDo_sig2[1] == 0)
                    sw_data2_i <= MA_Data1_o;
                else
                    sw_data2_i <= MA_Data2_o;
            else
                sw_data2_i <= IDo_src3;

            MA_sig1_i <= EX_sig1_o ;    MA_sig2_i<= EX_sig2_o;
            MA_ALU1_TMP <= EXo_ALU1;    MA_ALU2_TMP <= EXo_ALU2;

            WB_sig1_i <= MA_sig1_o;     WB_sig2_i <= MA_sig2_o;
```

```
    end
end

reg now_RST;    //唯一的作用就是说明当前周期是不是 RST
always @(posedge CLK) begin
    if(RST) now_RST <= 1;
    else now_RST <= 0;
end

always @(*) begin
    //维护旁路数据连接情况与流水线清空
    if(now_RST) begin
        //EX
        sw_data1_o <= 0;    sw_data2_o <= 0;
        EX_sig1_o <= 0;     EX_sig2_o <= 0;
        //EX end
        //MA
        MA_sig1_o <= 0; MA_sig2_o <= 0;
        MA_Data1_o <= 0;
        MA_Data2_o <= 0;
        //MA end
        ID_clr_w_mask <= 0;
    end
    //EX
    //旁路
    case (IDo_sig1[11:8])
        4'b0010: EX_bypass0 <= MA_Data1_o;
        4'b0011: EX_bypass1 <= MA_Data1_o;
        4'b0100: EX_bypass0 <= EXo_ALU1;
        4'b0101: EX_bypass1 <= EXo_ALU1;
        4'b1010: EX_bypass0 <= MA_Data2_o;
        4'b1011: EX_bypass1 <= MA_Data2_o;
        4'b1100: EX_bypass0 <= EXo_ALU2;
        4'b1101: EX_bypass1 <= EXo_ALU2;
    endcase

    case (IDo_sig2[11:8])
        4'b0010: EX_bypass2 <= MA_Data1_o;
        4'b0011: EX_bypass3 <= MA_Data1_o;
        4'b0100: EX_bypass2 <= EXo_ALU1;
        4'b0101: EX_bypass3 <= EXo_ALU1;
        4'b1010: EX_bypass2 <= MA_Data2_o;
        4'b1011: EX_bypass3 <= MA_Data2_o;
        4'b1100: EX_bypass2 <= EXo_ALU2;
        4'b1101: EX_bypass3 <= EXo_ALU2;
    endcase
    //end
    sw_data1_o <= sw_data1_i;
    sw_data2_o <= sw_data2_i;
    EX_sig1_o <= EX_sig1_i;
    if(EX_clear_pipeline2) begin
        EX_sig2_o <= 0;
    end
    else begin
        EX_sig2_o <= EX_sig2_i;
    end
    //EX end

    //MA
    //旁路
    case (EX_sig1_o[11:8])
        4'b0110 : DRAM_bypass1 <= MA_Data1_o;
        4'b1110 : DRAM_bypass1 <= MA_Data2_o;
    endcase
    case (EX_sig2_o[11:8])
        4'b0110 : DRAM_bypass2 <= MA_Data1_o;
        4'b1110 : DRAM_bypass2 <= MA_Data2_o;
    endcase
    //end
    MA_sig1_o <= MA_sig1_i;
    MA_sig2_o <= MA_sig2_i;
```

```
        if(MA_sig1_o[20] && MA_sig1_o[18]) begin //MemToReg
            MA_Data1_o <= MAo_data1;
        end
        else begin
            MA_Data1_o <= MA_ALU1_TMP;
        end

        if(MA_sig2_o[20] && MA_sig2_o[18]) begin //MemToReg
            MA_Data2_o <= MAo_data2;
        end
        else begin
            MA_Data2_o <= MA_ALU2_TMP;
        end
        //MA end

        ID_clr_w_mask <= 0;
        ID_clr_w_mask[MA_sig1_o[16:12]] <= 1'b1;
        ID_clr_w_mask[MA_sig2_o[16:12]] <= 1'b1;

    end
endmodule
```

DRAM 模块，其中 ram4k32inout 先前实验已给出（异步实现）

```
module DRAM
    (
        input CLK,                  // 时钟信号
        input RST,                 // 复位信号

        //来自上级流水线内容
        input [31:0] Data1in, Data2in,  // 数据输入输出
        input [31:0] Addr1, Addr2,  // 地址输入
        input R_W1, R_W2,           // 双流水线读写信号，0 写 1 读
        input CS1,CS2,                  // 使能信号
        input [31:0] bypass1,bypass2,
        input DSRC1,DSRC2,               //SRC = 0 使用 Datain SRC =1 使用 bypass
        //end

        //DRAM 不接受可变中间控制信号，必须在上一时钟内确定全部信号

        //交付给下级流水线内容
        output reg [31:0] Data1out,         //支持旁路 DM->ALU DM->DM
        output [31:0] Data2out
        //end
    );

    // 内部信号 控制 ram
    wire [31:0] ram_data_inout;     // 从 RAM 输出的数据
    reg [31:0] ram_in_data;         //如果要写入
    reg [31:0] ram_addr;
    reg ram_RW;        // 写入标志
    reg ram_cs;
    //
    //pipeline reg for 2
    reg [31:0] tmp_in_data2;
    reg tmp_in_CS2;
    reg tmp_R_W2;
    reg [31:0] tmp_addr2;
    //pipeline reg for 2

    // 实例化 RAM 模块
    RAM_4Kx32_inout ram(
        .Data(ram_data_inout),      // RAM 输出数据
        .Addr(ram_addr[11:0]),       // 地址输入（使用低 12 位）
        .Rst(RST),                 // 复位信号
        .R_W(ram_RW),   // 读信号
        .CS(ram_cs),               // 使能信号
        .CLK(CLK)                  // 时钟信号(RAM 内部是异步的，实际不使用时钟信号)
    );

    always @(posedge CLK or negedge CLK or posedge RST) begin
        ram_RW = 1;
```

```
        ram_cs = 0;
        if(RST && CLK) begin
            ram_cs <=0;
            Data1out <= 0;
            tmp_in_data2 <=0;
            tmp_in_CS2 <= 0;
            tmp_R_W2 <= 0;
            tmp_addr2 <=0;
        end
        else if(RST && ~CLK) begin
            ram_cs <=0;
            Data1out <= 0;
            tmp_in_data2 <=0;
            tmp_in_CS2 <= 0;
            tmp_R_W2 <= 0;
            tmp_addr2 <=0;
        end
        else begin
            if(CLK) begin
                //时钟上升沿

                //非阻塞式保存下级时钟流水线 ASAP
                if(DSRC2)   tmp_in_data2 <= bypass2;
                else        tmp_in_data2 <= Data2in;
                //tmp_in_data2 <= (DSRC2)  bypass2 : Data2in;
                tmp_in_CS2 <= CS2;
                tmp_R_W2 <= R_W2;
                tmp_addr2 <= Addr2;
                //end

                //保存并输入来自上级流水线内容
                if(~R_W1)
                if(DSRC1)   ram_in_data <= bypass1; //如果写入
                else        ram_in_data <= Data1in; //如果写入
                else ram_in_data <= 0;

                ram_addr = Addr1;   //注意数据竞争，使用阻塞式，防止错误写入数据
                ram_cs = CS1;
                ram_RW = R_W1;
                //end
            end
            else begin
                //时钟下降沿
                Data1out <= ram_data_inout;

                ram_addr = tmp_addr2;

                ram_RW = tmp_R_W2;
                ram_cs = tmp_in_CS2;
                if(~R_W2) ram_in_data <= tmp_in_data2; //如果写入
                else ram_in_data = 0;
            end
        end
    end

    assign ram_data_inout = (~ram_RW) ? ram_in_data : 32'bz;
    assign Data2out = ram_data_inout;
endmodule
```

EX 模块，其中 ALU 先前实验已给出

```
`include "macro.v"
`timescale 1ns/1ps


//执行阶段

module EX_stage (
    input CLK,
    input RST,
    //上级流水线交付内容，需及时保存
    input [31:0] src0,src1,src2,src3,
```

```
    input [31:0] PC1,PC2,
    input [31:0] imm1,imm2,
    input [1:0] PCSRC1,PCSRC2,                //00 不更改 PC，01 使用 beq（PC+ sigext(imm1) << 2），
10 使用 j（pc 高 4 位 拼接 26 位 target 拼接 00）
    input exsign1,exsign2,
    input [1:0] ALU1_SRCA,ALU1_SRCB,
    input [1:0] ALU2_SRCA,ALU2_SRCB,
                                        //SRCA1  01:  使用 src0 寄存器  00: 使用
32'b0                                  11:使用 src_by_0
                                        //SRCB1  01:  使用 src1 寄存器  00: 使用 32'b0  10: 使用
sigext16(imm1)符号扩展  11:使用 src_by_1
                                        //SRCA2  01:  使用 src2 寄存器  00: 使用
32'b0                                  11:使用 src_by_2
                                        //SRCB2  01:  使用 src3 寄存器  00: 使用 32'b0  10: 使用
sigext16(imm2)符号扩展  11:使用 src_by_3
    input [3:0] ALU_OP1,ALU_OP2,        //ALU OP 定义见 macro

    //旁路
    input [31:0] src_by_0,src_by_1,src_by_2,src_by_3,     //DM->ALU OR ALU->ALU 由外部决定
    //end

    //ex 接受的逻辑电路控制信号
    //end

    //当前要实时向外输出的逻辑电路控制信号
    output reg set_PC,
    output reg [31:0] ex_PC,
    output reg clear_pipeline2,//如果 beq/j 在第 1 条流水线上，则要清空第 2 条流水线内容 实际上如果第一
条是 j，第二条就不应该进内容
    //end

    //向下级流水线交付内容 //后续支持也属于旁路 ALU->ALU
    output reg [31:0] ALU_out1,ALU_out2
    //end
);

    //内部保存流水线状态
    reg [31:0] src0_in,src1_in,src2_in,src3_in;
    reg [31:0] PC1_in,PC2_in;
    reg [31:0] imm1_in,imm2_in;
    reg [1:0] PCSRC1_in,PCSRC2_in;
    reg exsign1_in,exsign2_in;
    reg [1:0] ALU1_SRCA_in,ALU1_SRCB_in;
    reg [1:0] ALU2_SRCA_in,ALU2_SRCB_in;
    reg [3:0] ALU_OP1_in,ALU_OP2_in;
    reg [31:0] src_by_0_in,src_by_1_in,src_by_2_in,src_by_3_in;


    wire [31:0] imm1_ext32 , imm2_ext32;
    reg [31:0] PC1_beq , PC1_j , PC2_beq , PC2_j;

    always @(*) begin
        PC1_beq <= PC1_in + (imm1_ext32 << 2) ;
        PC1_j   <= {PC1_in[31:28],imm1_ext32,2'b00};
        PC2_beq <= PC2_in + (imm2_ext32 << 2) ;
        PC2_j   <= {PC2_in[31:28],imm2_ext32,2'b00};
    end


    Extender    SigExt16_1(imm1_in[15:0],exsign1_in,imm1_ext32),
                SigExt16_2(imm2_in[15:0],exsign2_in,imm2_ext32);

    //需要处理 ALU 输入
    reg [31:0] ALU1_A,ALU1_B,ALU2_A,ALU2_B;
    wire [31:0] ALU1_F,ALU2_F;
    wire ZF1,CF1,OF1,SF1,PF1;
    wire ZF2,CF2,OF2,SF2,PF2;

    always @(*) begin
        case (ALU1_SRCA_in)
```

```
            2'b00: ALU1_A <= 0;
            2'b01: ALU1_A <= src0_in;
            2'b11: ALU1_A <= src_by_0_in;
            default: ALU1_A <= 32'bz;
        endcase
        case (ALU1_SRCB_in)
            2'b00: ALU1_B <= 0;
            2'b01: ALU1_B <= src1_in;
            2'b10: ALU1_B <= imm1_ext32;
            2'b11: ALU1_B <= src_by_1_in;
            default: ALU1_B <= 32'bz;
        endcase
        case (ALU2_SRCA_in)
            2'b00: ALU2_A <= 0;
            2'b01: ALU2_A <= src2_in;
            2'b11: ALU2_A <= src_by_2_in;
            default:  ALU2_A <= 32'bz;
        endcase
        case (ALU2_SRCB_in)
            2'b00: ALU2_B <= 0;
            2'b01: ALU2_B <= src3_in;
            2'b10: ALU2_B <= imm2_ext32;
            2'b11: ALU2_B <= src_by_3_in;
            default: ALU2_B <= 32'bz;
        endcase

        case (ALU_OP1_in)
            `ALU_OP_LUI_16TO32 : ALU_out1 <= {imm1_in[15:0],16'b0};
            default: ALU_out1 <= ALU1_F;
        endcase

        case (ALU_OP2_in)
            `ALU_OP_LUI_16TO32 : ALU_out2 <= {imm2_in[15:0],16'b0};
            default:  ALU_out2 <= ALU2_F;
        endcase
    end

    ALU         ALU1(
                    .OP(ALU_OP1_in),
                    .A(ALU1_A),
                    .B(ALU1_B),
                    .F(ALU1_F),
                    .ZF(ZF1),
                    .CF(CF1),
                    .OF(OF1),
                    .SF(SF1),
                    .PF(PF1)
                ),
                ALU2 (
                    .OP(ALU_OP2_in),
                    .A(ALU2_A),
                    .B(ALU2_B),
                    .F(ALU2_F),
                    .ZF(ZF2),
                    .CF(CF2),
                    .OF(OF2),
                    .SF(SF2),
                    .PF(PF2)
                );


    always @(posedge CLK) begin
    if (RST) begin
        // 当复位信号为高时，清零所有内部寄存器
        src0_in <= 32'b0;
        src1_in <= 32'b0;
        src2_in <= 32'b0;
        src3_in <= 32'b0;
        PC1_in <= 32'b0;
        PC2_in <= 32'b0;
        imm1_in <= 32'b0;
        imm2_in <= 32'b0;
        PCSRC1_in <= 1'b0;
```

```
        PCSRC2_in <= 1'b0;
        exsign1_in <= 1'b0;
        exsign2_in <= 1'b0;
        ALU1_SRCA_in <= 2'b00;
        ALU1_SRCB_in <= 2'b00;
        ALU2_SRCA_in <= 2'b00;
        ALU2_SRCB_in <= 2'b00;
        ALU_OP1_in <= 4'b0000;
        ALU_OP2_in <= 4'b0000;
        src_by_0_in <= 32'b0;
        src_by_1_in <= 32'b0;
        src_by_2_in <= 32'b0;
        src_by_3_in <= 32'b0;
    end else begin
        // 当复位信号为低时，将上级流水线的信号复制到内部寄存器
        src0_in <= src0;
        src1_in <= src1;
        src2_in <= src2;
        src3_in <= src3;
        PC1_in <= PC1;
        PC2_in <= PC2;
        imm1_in <= imm1;
        imm2_in <= imm2;
        PCSRC1_in <= PCSRC1;
        PCSRC2_in <= PCSRC2;
        exsign1_in <= exsign1;
        exsign2_in <= exsign2;
        ALU1_SRCA_in <= ALU1_SRCA;
        ALU1_SRCB_in <= ALU1_SRCB;
        ALU2_SRCA_in <= ALU2_SRCA;
        ALU2_SRCB_in <= ALU2_SRCB;
        ALU_OP1_in <= ALU_OP1;
        ALU_OP2_in <= ALU_OP2;
        src_by_0_in <= src_by_0;
        src_by_1_in <= src_by_1;
        src_by_2_in <= src_by_2;
        src_by_3_in <= src_by_3;
    end
end

    always @(*) begin
        //处理控制逻辑
        clear_pipeline2 = 1'b0;
        if( PCSRC1_in != 2'b00) begin
            //第一条是跳转
            case (PCSRC1_in)
                2'b01: begin
                    //beq
                    if(ZF1) begin
                        //zf 需要跳转
                        set_PC = 1'b1;
                        ex_PC = PC1_beq;
                        clear_pipeline2 = 1'b1;
                    end
                    else begin
                        // 不跳转
                        set_PC = 1'b0;
                        ex_PC = PC1_beq;
                        clear_pipeline2 = 1'b0;
                    end
                end
                2'b10: begin
                    //j 必然跳转
                    set_PC = 1'b1;
                    ex_PC = PC1_j;
                    clear_pipeline2 = 1'b1;
                end
                default: begin
                    set_PC = 1'bz;
                    ex_PC =  32'bz;
                    clear_pipeline2 = 1'bz;
                end
            endcase
```

```
        end
        if(clear_pipeline2 == 0) begin
            if( PCSRC2_in != 2'b00) begin
                //第二条是跳转
                case (PCSRC2_in)
                    2'b01: begin
                        //beq
                        if(ZF2) begin
                            //zf 需要跳转
                            set_PC = 1'b1;
                            ex_PC = PC2_beq;
                        end
                        else begin
                            // 不跳转
                            set_PC = 1'b0;
                            ex_PC = PC2_beq;
                        end
                    end
                    2'b10: begin
                        //j 必然跳转
                        set_PC = 1'b1;
                        ex_PC = PC2_j;
                    end
                    default: begin
                        set_PC = 1'bz;
                        ex_PC =  32'bz;
                    end
                endcase
            end
            else begin
                //第一第二条都不是跳转
                set_PC = 0 ;
                ex_PC = 0 ;
                clear_pipeline2 = 0;
            end
        end


    end




endmodule


module Extender #(
    parameter X_WIDTH = 16 // 输入位宽
) (
    input  wire [X_WIDTH-1:0] in,          // 输入
    input  wire        is_signed,          // 符号扩展
    output wire [31:0] out                 // 输出
);
    // 符号扩展或无符号扩展逻辑
    assign out = is_signed ? { {32-X_WIDTH{in[X_WIDTH-1]}}, in }  : { 16'b0, in };     // 符号扩
展 与 无符号扩展
endmodule
```

FI 模块

```
`timescale 1ns/1ps
`define DATA_WIDTH 32


//设计目标
//维护内部 PC，ex_setPC 为外部控制信号，要求为逻辑电路
//不存在上一流水线内容
```

```v
module FI_stage (
    input RST,
    input CLK,


    //连接到外部 IMem
    output [5:0] ROM_A1,
    output [5:0] ROM_A2,
    input [31:0] ROM_RD1,
    input [31:0] ROM_RD2,
    //连接到外部 IMem end
    //控制信号，实时变化来自其他流水线输出
    input ex_setPC,
    input wire [31:0] ex_PC,
    //end

    //交付给下级流水线
    output [31:0] out_PC1,   // 第一条指令输出时的 PC(即指向第二条指令开始位置)
    output [31:0] out_PC2,   // 第一条指令输出时的 PC(即指向第二条指令开始位置)
    output [31:0] inst_1,  // 第一条指令输出
    output [31:0] inst_2  // 第二条指令输出
    //end

);
    // 计算指令地址
    wire [5:0] inst1_addr;
    wire [5:0] inst2_addr;
    reg [31:0] PC;


    assign inst1_addr = (ex_setPC) ? ex_PC[7:2] : PC[7:2];
    assign inst2_addr = (ex_setPC) ? ex_PC[7:2] + 1 : PC[7:2] + 1;
    assign out_PC1 = (ex_setPC) ? ex_PC + 4 : PC + 4;
    assign out_PC2 = (ex_setPC) ? ex_PC + 8 : PC + 8;
    // 其实可以改为 always @(*)类型

    // 将指令存储器搬到 cpu 外部
    assign ROM_A1 = inst1_addr;
    assign ROM_A2 = inst2_addr;
    assign inst_1 = ROM_RD1;
    assign inst_2 = ROM_RD2;
    // 将指令存储器搬到 cpu 外部 end

    // IMem imem (
    //     .A1(inst1_addr),
    //     .A2(inst2_addr),
    //     .RD1(inst_1),
    //     .RD2(inst_2)
    // );

    reg last_RST;
    always @(posedge CLK) begin
        if (RST) begin
            PC=32'b0;
            last_RST <= 1;
        end else begin
            if(last_RST) begin
                last_RST <= 0;
                PC = PC + 8;
            end
            else if(ex_setPC) begin
                //意味上一刻使用的是 expc
                PC = ex_PC + 8;
            end
            else begin
                //意味着上一刻使用的内置 pc
                PC = PC + 8 ;
            end
        end
    end
endmodule
```

ID 模块

```
`include "macro.v"
`timescale 1ns/1ps
`define DATA_WIDTH 32


module ID_stage (
    input CLK,
    input RST,
    input pipeline_hang,              //请求流水线暂停，其他流水线上一刻输出(暂未实现)
    //上级流水线输出
    input [31:0] clr_w_mask, //（掩码类）当前时钟下寄存器写入完成

    input [31:0] PC1,   // 第一条指令输出时的 PC(即指向第二条指令开始位置)
    input [31:0] PC2,   // 第二条指令输出时的 PC
    input [31:0] inst_1,  // 第一条指令输出
    input [31:0] inst_2,  // 第二条指令输出
    //end

    //实时控制信号
    //输出 请求 4 个寄存器读地址 控制 FI 实时数据
    output [4:0] RAddr1, RAddr2, RAddr3, RAddr4,

    output reg ID_set_PC,       //请求重新设置 PC
    output reg [31:0] ID_PC,    //设置的 PC 值

    //输入 4 个寄存器读数据 EX 阶段的 beq 和 j 数据分支判断数据
    input [31:0] RData1,RData2,RData3,RData4,

    input EX_set_PC,
    input [31:0] EX_PC,
    //input EX_clear_pipeline2, // 应该没用
    //end

    //交付给下一级流水线数据
    output [31:0] src0,src1,src2,src3,              //ALU 4 个寄存器
    output [31:0] imm1,imm2,
    output reg [31:0] PC1_o,PC2_o,
    //以上需要直接交付给 ALU
    output reg [31:0] inst1_signal,inst2_signal,        //最重要的控制信号
    output [31:0] sw_data_1,sw_data_2               //想要 sw 的数据
    //end

);


    //内部寄存器
    reg [31:0] reg_w_mask;          //（掩码类）寄存器占用状态（写入）为 1 时，表示对应寄存器正在被写
入
    reg [3:0] reg_w_status[0:31] ;   //寄存器占用原因（用于支持旁路数据） 是状态机

    reg [4:0] inst1_target_reg , inst2_target_reg; //标志两个指令如果发射分别设置的 mask（set），是
超标量计算结果
    reg [3:0] reg_w_status_1, reg_w_status_2 ;     //标志两个指令如果发射分别设置的 status，是超标
量的计算结果(修正，可无状态逻辑电路计算)

    reg inst1_valid,inst2_valid ;    //标志两个指令是否可发射，是超标量计算结果
    reg inst1_launch,inst2_launch ;  //标志两个指令实际最终有没有发射。
    reg [3:0] by_pass_1,by_pass_2 ;   //标志两个指令旁路数据使用状态，影响 signal 的 ALU_SRC DSRC 以
及状态



    //局部临时寄存器
    reg now_RST;

    //保存上级流水线寄存器
```

```
    reg [31:0] clr_w_mask_in, PC1_in,PC2_in,inst_1_in,inst_2_in;
    //内部寄存器 end

    integer i,j,k;

    always @(posedge CLK) begin
        if(RST) begin
            now_RST <=1'b1;
            //清空所有内部寄存器
            reg_w_mask <= 32'b0;
            inst1_target_reg <= 0;
            inst2_target_reg <= 0;
            reg_w_status_1 <= 4'b0;
            reg_w_status_2 <= 4'b0;
            for (i = 0; i<=31 ; i = i+1 ) begin
                reg_w_status[i] <= 4'b0;
            end
            clr_w_mask_in <= 0;
            PC1_in <= 0;
            PC2_in <= 0;
            inst_1_in <= 0;
            inst_2_in <= 0;

            PC1_o <= 0;
            PC2_o <= 0;
            by_pass_1 <= 0;
            by_pass_2 <= 0;
        end
        else begin
            now_RST <= 1'b0;
            //时序电路，阻塞式函数
            //保存上级流水线数据
            PC1_o <= PC1;
            PC2_o <= PC2;
            clr_w_mask_in = clr_w_mask;
            PC1_in = PC1;
            PC2_in = PC2;
            inst_1_in = inst_1;
            inst_2_in = inst_2;
            //保存上级流水线数据 end

            //更新 reg_w_mask 和 status 使用状态机

            //for(reg_w_status) 状态机进一步
            for(i = 0 ; i<32 ; i=i+1) begin
                if(reg_w_status[i][2:0] == 3'b010) reg_w_status[i][2:0] = 3'b001;
                else if(reg_w_status[i][2:0] == 3'b011) reg_w_status[i][2:0] = 3'b001;
                else if(reg_w_status[i][2:0] == 3'b001) begin
                    reg_w_status[i][2:0] = 3'b000;
                    reg_w_mask[i] = 0;
                end
            end

            if(inst1_launch) begin
                reg_w_mask[inst1_target_reg] = 1;
                reg_w_status[inst1_target_reg] = reg_w_status_1;
            end
            if(inst2_launch) begin
                reg_w_mask[inst2_target_reg] = 1; //如果上一刻成功发射
                reg_w_status[inst2_target_reg] = reg_w_status_2;
            end
            reg_w_mask [0] = 0; //0 号寄存器无论如何都不会被占用
            reg_w_mask = reg_w_mask & (~clr_w_mask_in); //这一刻完成寄存器写入的寄存器


            //更新 reg_w_mask 和 status end

            //超标量-更新两个 inst_vali 以及 by_pass_

            inst1_valid = 1;
            inst2_valid = 1;
```

```
            //所有指令{add, sub, addi,  lw, sw, beq, j, nop , ori , lui}
            //需要写入到寄存器的指令 R-type(add sub) 写入到$rd  I-type(addi lw ori lui) 写入到 rt
            case (inst_1_in[31:26])
                6'b000000: begin
                    //R-type
                    inst1_target_reg = inst_1_in[15:11];
                end
                `_OP_addi , `_OP_lw , `_OP_ori , `_OP_lui : begin
                    //I-type
                    inst1_target_reg = inst_1_in[20:16];
                end
                default: inst1_target_reg = 0;
            endcase

            case (inst_2_in[31:26])
                6'b000000: begin
                    //R-type
                    inst2_target_reg = inst_2_in[15:11];
                end
                `_OP_addi , `_OP_lw , `_OP_ori , `_OP_lui : begin
                    //I-type
                    inst2_target_reg = inst_2_in[20:16];
                end
                default: inst2_target_reg = 0;
            endcase

            //如果要写入的寄存器正被写，则无效
            if(reg_w_mask[inst1_target_reg]==1) begin
                inst1_valid = 0;
            end
            if(reg_w_mask[inst2_target_reg]==1) begin
                inst2_valid = 0;
            end
            //如果两条指令写入寄存器相同，则第二条无效
            if(inst1_target_reg!= 0 && inst1_target_reg == inst2_target_reg) inst2_valid = 0;
            //计算要写入的寄存器 end

            //判断要读的寄存器有没有占用
            //所有指令{add, sub, addi,  lw, sw, beq, j, nop , ori , lui}
            //以及 addu subu
            case (inst_1_in[31:26])
                6'b000000 , `_OP_beq : begin
                    //R-type
                    case ({reg_w_mask[inst_1_in[25:21]],reg_w_mask[inst_1_in[20:16]]})
                        2'b11 : inst1_valid = 0 ; //如果两个寄存器都被占用，就拒绝了
                        2'b10 : begin
                            case(reg_w_status[inst_1_in[25:21]][2:0])
                                3'b010: inst1_valid = 0;
                                3'b011: by_pass_1 <= {reg_w_status[inst_1_in[25:21]][3],3'b100};
                                3'b001: by_pass_1 <= {reg_w_status[inst_1_in[25:21]][3],3'b010};
                                default : inst1_valid = 0;
                            endcase
                        end
                        2'b01 : begin
                            case(reg_w_status[inst_1_in[20:16]][2:0])
                                3'b010: inst1_valid = 0;
                                3'b011: by_pass_1 <= {reg_w_status[inst_1_in[20:16]][3],3'b101};
                                3'b001: by_pass_1 <= {reg_w_status[inst_1_in[20:16]][3],3'b011};
                                default : inst1_valid = 0;
                            endcase
                        end
                        2'b00 : by_pass_1 <= 0000;
                    endcase
                end
                `_OP_addi , `_OP_ori ,`_OP_lw  : begin
                    //I-type
                    if(reg_w_mask[inst_1_in[25:21]]==1) begin
                        case(reg_w_status[inst_1_in[25:21]][2:0])
                            3'b010: inst1_valid = 0;
                            3'b011: by_pass_1 <= {reg_w_status[inst_1_in[25:21]][3],3'b100};
                            3'b001: by_pass_1 <= {reg_w_status[inst_1_in[25:21]][3],3'b010};
                            default : inst1_valid = 0;
                        endcase
```

```
                    end
                    else by_pass_1 <= 0000;
                end
                `_OP_sw  : begin
                    if(reg_w_mask[inst_1_in[25:21]]==1) inst1_valid = 0;
                    else if(reg_w_mask[inst_1_in[20:16]]==1) begin
                        case(reg_w_status[inst_1_in[20:16]][2:0])
                            3'b010: by_pass_1 <= {reg_w_status[inst_1_in[20:16]][3],3'b110};
                            3'b011: by_pass_1 <= {reg_w_status[inst_1_in[20:16]][3],3'b110};
                            3'b001: by_pass_1 <= {reg_w_status[inst_1_in[20:16]][3],3'b001};
                            default : inst1_valid = 0;
                        endcase
                    end
                    else by_pass_1 <= 0000;
                end
                //default: inst1_valid = 1;
            endcase

            if(inst1_target_reg != 0 && (inst_2_in[25:21] == inst1_target_reg || inst_2_in[20:16]
== inst1_target_reg)) begin
                inst2_valid = 0;
            end
            else
            case (inst_2[31:26])
                6'b000000 , `_OP_beq : begin
                    //R-type
                    case ({reg_w_mask[inst_2_in[25:21]],reg_w_mask[inst_2_in[20:16]]})
                        2'b11 : inst2_valid = 0 ; //如果两个寄存器都被占用，就拒绝了
                        2'b10 : begin
                            case(reg_w_status[inst_2_in[25:21]][2:0])
                                3'b010: inst2_valid = 0;
                                3'b011: by_pass_2 <= {reg_w_status[inst_2_in[25:21]][3],3'b100};
                                3'b001: by_pass_2 <= {reg_w_status[inst_2_in[25:21]][3],3'b010};
                                default : inst2_valid = 0;
                            endcase
                        end
                        2'b01 : begin
                            case(reg_w_status[inst_2_in[20:16]][2:0])
                                3'b010: inst2_valid = 0;
                                3'b011: by_pass_2 <= {reg_w_status[inst_2_in[20:16]][3],3'b101};
                                3'b001: by_pass_2 <= {reg_w_status[inst_2_in[20:16]][3],3'b011};
                                default : inst2_valid = 0;
                            endcase
                        end
                        2'b00 : by_pass_2 <= 0000;
                    endcase
                end
                `_OP_addi , `_OP_ori,`_OP_lw  : begin
                    //I-type
                    if(reg_w_mask[inst_2_in[25:21]]==1) begin
                        case(reg_w_status[inst_2_in[25:21]][2:0])
                            3'b010: inst2_valid = 0;
                            3'b011: by_pass_2 <= {reg_w_status[inst_2_in[25:21]][3],3'b100};
                            3'b001: by_pass_2 <= {reg_w_status[inst_2_in[25:21]][3],3'b010};
                            default : inst2_valid = 0;
                        endcase
                    end
                    else by_pass_2 <= 0000;
                end
                `_OP_sw  : begin
                    if(reg_w_mask[inst_2_in[25:21]]==1) inst2_valid = 0;
                    else if(reg_w_mask[inst_2_in[20:16]]==1) begin
                        case(reg_w_status[inst_2_in[20:16]][2:0])
                            3'b010: by_pass_2 <= {reg_w_status[inst_2_in[20:16]][3],3'b110};
                            3'b011: by_pass_2 <= {reg_w_status[inst_2_in[20:16]][3],3'b110};
                            3'b001: by_pass_2 <= {reg_w_status[inst_2_in[20:16]][3],3'b001};
                            default : inst2_valid = 0;
                        endcase
                    end
                    else by_pass_2 <= 0000;
                end
                //default: inst2_valid = 1;
            endcase
```

```
        //判断要读的寄存器有没有占用 end

        //如果第一条无效，第二条必定无效
        if(~inst1_valid) inst2_valid = 0;

        //超标量-更新两个 inst_valid end

    end

end



//使用 assign 或者 always 的逻辑电路，主要考虑 ex 的 beq 和 j 的问题
assign RAddr1 = inst_1_in[25:21];
assign RAddr2 = inst_1_in[20:16];
assign RAddr3 = inst_2_in[25:21];
assign RAddr4 = inst_2_in[20:16];

assign src0 = RData1;
assign src1 = RData2;
assign src2 = RData3;
assign src3 = RData4;

assign imm1 = {6'b0,inst_1_in[25:0]};
assign imm2 = {6'b0,inst_2_in[25:0]};

assign sw_data_1 = RData2;
assign sw_data_2 = RData4;

wire [31:0] dec_signal1,dec_signal2;
MainDecoder maindec1(inst_1_in,by_pass_1,dec_signal1),
            maindec2(inst_2_in,by_pass_2,dec_signal2);


always @(*) begin
    if(now_RST) begin
        ID_set_PC <= 0;
        ID_PC <= 0;
    end
    else begin
       //inst_valid 决定输出到下级流水线的两个 signal 是否有效,同时决定下一个取回的 PC
        if(inst1_valid) begin
            inst1_signal <= dec_signal1;
            ID_set_PC <= 1;
            ID_PC <= PC1_in;
        end
        else begin
            inst1_signal <= 0;
        end

        if(inst2_valid) begin
            inst2_signal <= dec_signal2;
            ID_set_PC <= 0; //inst2 有效，说明 inst1 必有效
            ID_PC <= PC2_in;
        end
        else begin
            inst2_signal <= 0;
        end
        //ex 阶段 beq/j 返回决定下一个取回来的 pc，以及两个 signal 是否有效
        //设置 ID_set_PC 和 ID_PC 取决于两个 valid（上方已完成） 取决于 EX
        inst1_launch = inst1_valid;
        inst2_launch = inst2_valid;

        if(inst1_launch && inst2_launch) begin
            //发射双指令
            ID_set_PC <= 0;
            ID_PC <= 0;
        end
        else if(inst1_launch) begin
```

```
                //发射单指令
                ID_set_PC <= 1;
                ID_PC <= PC1_o;
            end
            else begin
                //不发射指令
                ID_set_PC <= 1;
                ID_PC <= PC1_o-4;
            end

            if(EX_set_PC) begin
                //超控
                inst1_launch = 0;
                inst2_launch = 0;
                inst1_signal <= 0; //插入空泡 nop
                inst2_signal <= 0; //插入空泡 nop
                ID_set_PC <= 1;
                ID_PC <= EX_PC;
            end

        end
    end

    //逻辑电路计算 reg_w_status_1, reg_w_status_2
    always @(*) begin
        reg_w_status_1[3] <= 0;
        reg_w_status_2[3] <= 1;
        //所有指令{add, sub, addi,  lw, sw, beq, j, nop , ori , lui}
        //以及 addu subu
        case(inst_1_in[31:26])
            6'b000000,`_OP_addi,`_OP_ori,`_OP_lui : reg_w_status_1[2:0] <= 3'b011;
            `_OP_lw : reg_w_status_1[2:0] <= 3'b010;
            default : reg_w_status_1[2:0] <= 3'b000;
        endcase

        case(inst_2_in[31:26])
            6'b000000,`_OP_addi,`_OP_ori,`_OP_lui : reg_w_status_2[2:0] <= 3'b011;
            `_OP_lw : reg_w_status_2[2:0] <= 3'b010;
            default : reg_w_status_2[2:0] <= 3'b000;
        endcase
    end
endmodule


module MainDecoder (
    input [31:0] inst,
    input [3:0] by_pass_status,
    output reg [31:0] signal);

    wire [5:0] Op;
    assign Op = inst[31:26];

    always @(*) begin
        signal[7:0] <= 8'b0; //保留位
        case (Op)
            6'b000000: begin
                //RTYPE
                if(inst == 32'b0) begin
                    signal <= 32'b0;
                end
                else begin
                   signal[31:29] <= 3'b001;
                    signal[20] <= 1'b0; //使用 DRAM
                    signal[18] <= 1'b1; //读=1/写=0 DRAM
                    signal[17] <= 1'b1; //写寄存器
                    signal[16:12] <= inst[15:11];

                    signal[28:25] <= 4'b0101; //SRCA SRCB R-type
                    signal[19] <= 1'b0; //DSRC
                end
            end
```

```
            6'b100011: begin
                //LW
                signal[31:29] <= 3'b001;
                signal[20] <= 1'b1; //使用 DRAM
                signal[18] <= 1'b1; //读=1/写=0 DRAM
                signal[17] <= 1'b1; //写寄存器
                signal[16:12] <= inst[20:16];   //I-type

                signal[28:25] <= 4'b0110; //SRCA SRCB I-type
                signal[19] <= 1'b0; //DSRC
            end
            6'b101011: begin
                //SW
                signal[31:29] <= 3'b001;
                signal[20] <= 1'b1; //使用 DRAM
                signal[18] <= 1'b0; //读=1/写=0 DRAM
                signal[17] <= 1'b0; //写寄存器
                signal[16:12] <= 5'b0;   //I-type

                signal[28:25] <= 4'b0110; //SRCA SRCB I-type
                signal[19] <= 1'b0; //DSRC
            end
            6'b000100: begin
                //BEQ
                signal[31:29] <= 3'b011; //PCSRC-BEQ
                signal[20] <= 1'b0; //使用 DRAM
                signal[18] <= 1'b1; //读=1/写=0 DRAM
                signal[17] <= 1'b0; //写寄存器
                signal[16:12] <= 5'b0;   //I-type

                signal[28:25] <= 4'b0101; //SRCA SRCB COMP-SUB
                signal[19] <= 1'b0; //DSRC
            end
            6'b001000: begin
                //ADDI
                signal[31:29] <= 3'b001;
                signal[20] <= 1'b0; //使用 DRAM
                signal[18] <= 1'b1; //读=1/写=0 DRAM
                signal[17] <= 1'b1; //写寄存器
                signal[16:12] <= inst[20:16];   //I-type $rt

                signal[28:25] = 4'b0110; //SRCA SRCB I-type
                signal[19] = 1'b0; //DSRC
            end
            6'b001101: begin
                //ORI
                signal[31:29] <= 3'b000; //符号拓展
                signal[20] <= 1'b0; //使用 DRAM
                signal[18] <= 1'b1; //读=1/写=0 DRAM
                signal[17] <= 1'b1; //写寄存器
                signal[16:12] <= inst[20:16];   //I-type $rt

                signal[28:25] <= 4'b0110; //SRCA SRCB I-type
                signal[19] <= 1'b0; //DSRC
            end
            6'b001111: begin
                //LUI
                signal[31:29] <= 3'b001;
                signal[20] <= 1'b0; //使用 DRAM
                signal[18] <= 1'b1; //读=1/写=0 DRAM
                signal[17] <= 1'b1; //写寄存器
                signal[16:12] <= inst[20:16];   //I-type $rt

                signal[28:25] <= 4'b0010; //SRCA SRCB SRCA 置 0
                signal[19] <= 1'b0; //DSRC
            end
            6'b000010: begin
                //J
                signal[31:29] <= 3'b101; //PCSRC-J
                signal[20] <= 1'b0; //使用 DRAM
                signal[18] <= 1'b1; //读=1/写=0 DRAM
```

```
                signal[17] <= 1'b0; //写寄存器
                signal[16:12] <= 5'b0;   //J-type

                signal[28:25] <= 4'b0000; //置 0
                signal[19] <= 1'b0; //DSRC
            end
            default: begin
                // illegal Op
                signal[31:29] <= 3'bzzz; //PCSRC-J
                signal[20] <= 1'bz; //使用 DRAM
                signal[18] <= 1'bz; //读=1/写=0 DRAM
                signal[17] <= 1'bz; //写寄存器
                signal[16:12] <= 5'bz;   //J-type

                signal[28:25] <= 4'b0000; //置 0
                signal[19] <= 1'b0; //DSRC
            end
        endcase

        case (by_pass_status[2:0])
            3'b010,3'b100 : signal[28:27] <= 2'b11;
            3'b011,3'b101 : signal[26:25] <= 2'b11;
            3'b110 : signal[19] <= 1'b1;
        endcase
            //按照旁路规则修改 ALU SRCA SRCB DSRC

        signal[11:8] <= by_pass_status;

        //操作计算 ALUOP
        if(Op == 6'b0) begin
            //R-type
            case (inst[5:0])
                6'b100000: signal[24:21] <= `ALU_OP_ADD; //add
                6'b100001: signal[24:21] <= `ALU_OP_ADD; //addu
                6'b100010: signal[24:21] <= `ALU_OP_SUBTRACT; //sub
                6'b100011: signal[24:21] <= `ALU_OP_SUBTRACT; //subu
                default: signal[24:21] <= 6'b0; //nop
            endcase
        end
        else begin
            case (Op)
                6'b001000: signal[24:21] <= `ALU_OP_ADD; //ADDI
                6'b001101: signal[24:21] <= `ALU_OP_BITWISE_OR; //ORI
                6'b000100: signal[24:21] <= `ALU_OP_SUBTRACT; //beq
                `_OP_lui : signal[24:21] <= `ALU_OP_LUI_16TO32; // LUI
                default: signal[24:21] <= `ALU_OP_ADD; //lw,sw,j
            endcase
        end

    end

endmodule
```

IMem 模块

```
`timescale 1ns/1ps


`define DATA_WIDTH 32

module IMem(  // 指令存储器
    input [5:0] A1,
    input [5:0] A2,
    output [`DATA_WIDTH-1:0] RD1,
    output [`DATA_WIDTH-1:0] RD2
);
    parameter IMEM_SIZE = 64;  // 指令存储器大小

    // 指令存储器
    reg [`DATA_WIDTH-1:0] ROM[IMEM_SIZE-1:0];

    initial begin
```

```
        //$readmemh("C:/code/vivado/jz00_simpleCPU/test/memfile.txt", ROM);  // 从文件读取数据
        //加载指令（机器码）
        // ROM[0] = 32'h34080005;  // ori $t0, $zero, 5
        // ROM[1] = 32'h3409000A;  // ori $t1, $zero, 10
        // ROM[2] = 32'h01095021;  // addu $t2, $t0, $t1
        // ROM[3] = 32'h01285823;  // subu $t3, $t1, $t0
        // ROM[4] = 32'h200E0002;  // addi $t6 $zero 0x2
        // ROM[5] = 32'h016E7820;  // add $t7 $t3 $t6
        // ROM[6] = 32'hAD0A0000;  // sw $t2, 0($t0)
        // ROM[7] = 32'h8D0C0000;  // lw $t4, 0($t0)
        // ROM[8] = 32'h114C0002;  // beq $t2, $t4, label
        // ROM[9] = 32'h3C0D1234;  // lui $t5, 0x1234
        // ROM[10] = 32'h00000000;  // nop
        // ROM[11] = 32'h00000000;  // nop (label 后续指令)
        // ROM[12] = 32'h3C0D5678;  // lui $t5, 0x5678
        // //ROM[12] = 32'h00000000;  // nop
        // ROM[13] = 32'h00000000;  // nop
        // ROM[14] = 32'h00000000;  // nop
        // ROM[15] = 32'h00000000;  // nop
        // ROM[16] = 32'h00000000;  // nop
        // ROM[17] = 32'h00000000;  // nop

        // ROM[0] = 32'h34080005;  // ori $t0, $zero, 5
        // ROM[1] = 32'h3409000A;  // ori $t1, $zero, 10
        // ROM[2] = 32'h01095021;  // addu $t2, $t0, $t1
        // ROM[3] = 32'h01285823;  // subu $t3, $t1, $t0
        // ROM[4] = 32'hAD0A0000;  // sw $t2, 0($t0)
        // ROM[5] = 32'h8D0C0000;  // lw $t4, 0($t0)
        // ROM[6] = 32'h114C0002;  // beq $t2, $t4, label
        // ROM[7] = 32'h3C0D1234;  // lui $t5, 0x1234
        // ROM[8] = 32'h00000000;  // nop
        // ROM[9] = 32'h00000000;  // nop (label 后续指令)
        // //ROM[10] = 32'h3C0D5678;  // lui $t5, 0x5678
        // ROM[10] = 32'h00000000;  // nop
        // ROM[11] = 32'h00000000;  // nop
        // ROM[12] = 32'h00000000;  // nop
        // ROM[13] = 32'h00000000;  // nop
        // ROM[14] = 32'h00000000;  // nop
        // ROM[15] = 32'h00000000;  // nop

        ROM[0] = 32'h20080005;
        ROM[1] = 32'h01095020;
        ROM[2] = 32'h21290001;
        ROM[3] = 32'h11090002;
        ROM[4] = 32'h08000000;
        ROM[5] = 32'h00000000;
        ROM[6] = 32'h00000000;
        ROM[7] = 32'h00000000;
        ROM[8] = 32'h00000000;
        ROM[9] = 32'h1109FFFF;
        ROM[10] = 32'h00000000;
        ROM[11] = 32'h00000000;
        ROM[12] = 32'h00000000;
        // addi $t0  $zero 5 //循环次数
        // addu $t2  $t0  $t1
        // addi $t1 $t1 1
        // beq $t0,$t1,2
        // nop
        // j 0
        // beq $t0 $t1 -1
    end
    assign RD1 = ROM[A1];  // 指令输出
    assign RD2 = ROM[A2];  // 指令输出
endmodule
```

RegFile 模块

```
`define DATA_WIDTH 32


//支持 4 读 2 写
module RegFile  // 寄存器文件
    #(parameter ADDR_SIZE = 5) //2^5=32
    (input CLK, RST ,W1E, W2E,
```

```
    input [ADDR_SIZE-1:0] RAddr1, RAddr2, RAddr3, RAddr4, WAddr1, WAddr2,
    input [`DATA_WIDTH-1:0] WData1, WData2,
    output [`DATA_WIDTH-1:0] RData1, RData2, RData3, RData4
);
    integer i;
    reg [`DATA_WIDTH-1:0] rf[2**ADDR_SIZE-1:0];  // 寄存器数组

    // 数据写入需要时钟同步
    always @(posedge CLK or posedge RST) begin
        if (RST) begin
            // 复位时，将所有寄存器初始化为0
            for (i = 0; i < 2**ADDR_SIZE; i = i + 1) begin
                rf[i] <= 0;
            end
        end
        else begin
            if (W1E) rf[WAddr1] <= WData1;
            if (W2E) rf[WAddr2] <= WData2;
        end
    end

    // 数据读取
    assign RData1 = (RAddr1 != 0) ? rf[RAddr1] : 0;
    assign RData2 = (RAddr2 != 0) ? rf[RAddr2] : 0;
    assign RData3 = (RAddr3 != 0) ? rf[RAddr3] : 0;
    assign RData4 = (RAddr4 != 0) ? rf[RAddr4] : 0;

    // initial begin
    //     $readmemh("C:/code/vivado/jz04/jz04.srcs/sources_1/memfile.txt", rf);  // 从文件读取
数据
    // end

endmodule
```