32 位 MIPS 处理器设计

实验报告

无 64 邓程昊 2016011070
无 64 徐泽来 2016011088
无 64 应睿 2016011097

一、 实验目的

1. 熟悉现代处理器的基本工作原理。
2. 掌握单周期和流水线处理器的设计方法。

二、 设计方案

1. 32 位 ALU（完成人：邓程昊）

ALU 的设计基本上是按照实验指导书上的思路进行的，其大致结构如图 1 所示，可以分为四个主要单元：加减运算单元、比较运算单元、逻辑运算单元和移位运算单元。最后的结果会由输入到 ALU 的控制信号 ALUFun 所控制的四路选择器给出。

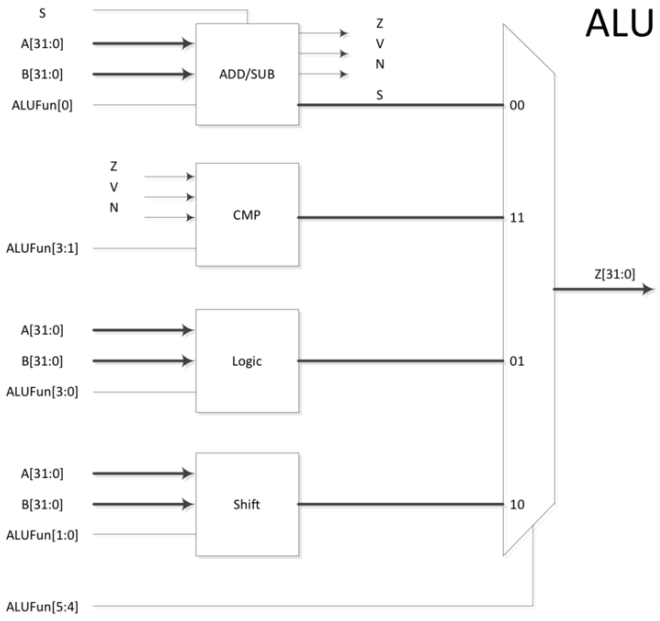


图 1 ALU 结构图

加减运算单元基本可以视为一个普通的全加器。分析 ALUFun 的信号表（图 2）可得，其最低位为 0 时进行正常的加法运算，为 1 时进行减法运算，实现方法即为将 B 逐位取反与 A 相加后再加 1。计算得到的结果即可作为 ALUFun 前两位为 00 时的输出。同时通过逐位或判断计算结果是否为零，作为 nonzero 信号；通过 A 和 B 的最高位与输入到 ALU 的控制信号 Sign 判断计算结果是否为负，作为 negative 信号；通过 ALUFun

的相关位与控制信号 Sign 判断指令关注的结果(即 A 或 A 与 B 的差)是否小于等于零，作为 compare_zero 信号。这三种信号会作为比较运算单元及其他两个单元的输入信号。

类型	功能	ALUFun	描述
算术	ADD	000000	S=A+B
	SUB	000001	S=A-B
位运算	AND	011000	S=A&B
	OR	011110	S=A|B
	XOR	010110	S=A^B
	NOR	010001	S=~(A|B)
	“A”	011010	S=A
移位运算	SLL	100000	S=B<<A[4:0]
	SRL	100001	S=B>>A[4:0]
	SRA	100011	S=B>>a[4:0] 算术移位
关系运算	EQ	110011	If(A==B) S=1 else S=0
	NEQ	110001	If(A!=B) S=1 else S=0
	LT	110101	If(A<B) S=1 else S=0
	LEZ	111101	If(A<=0) S=1 else S=0
	LTZ	111011	If(A<0) S=1 else S=0
	GTZ	111111	If(A>0) S=1 else S=0

图 2 ALU 功能表

逻辑运算单元的实现也比较简单。只要根据 ALUFun 信号的相关位判定，进行相应的按位与、按位或、按位异或、按位或非以及直接输出 A 即可。输出的结果即可作为 ALUFun 前两位为 01 时的输出。

移位运算单元通过五级多路选择器级联组合实现的。每一级会通过 A 操作数的相应为上的值决定 B 保持不变还是移动 1、2、4、8 或 16 位，这样就可以实现 32 位的移位。需要强调的是，除了逻辑左移和逻辑右移以外，还有算术右移。算术右移在 B 为正数时与逻辑右移完全相同，但 B 为负数时应该用 1 而不是 0 填补移位后的空缺位。输出的结果即可作为 ALUFun 前两位为 10 时的输出。

比较运算单元对应的指令较多，但是都可以用 nonzero、negative、compare_zero 三种信号以及 A 的最高位经过特定的逻辑运算得到，因此这部分也基本可以视为门电路与多路选择器的组合。输出的结果即可作为 ALUFun 前两位为 11 时的输出。

尽管在 Verilog 代码中将一些变量声明为了 reg 型，但实际上整个 ALU 并没有用到寄存器，因此可以视为组合逻辑电路。这样的设计不仅能提升电路在运算方面的性能，也可以简化后面流水线的工作。

2. 外设（完成人：应睿）

总的来说，外设模块主要负责 CPU 主模块与一系列 FPGA 板载设备的交流，也就是负责外设的读取与写入。

对于读取请求，外设模块是以组合逻辑的方式进行响应，当 rd 信号为 1 时即时输出当前地址指向的外设的数据；对于写入请求，外设需要一个时钟周期完成写入任务，因此在时钟上升沿来临时，当 wr 信号为 1，将当前数据写入对应地址。读写地址与外设的对应关系见图 3。

地址范围（字节地址）	功能	描述
0x00000000~0x000003FF	数据存储器	256×32bits（可以根据需要自行扩展大小）
0x40000000~0x4000000B	定时器	定时器外设地址 Timer
0x4000000C	外部 LEDs	0bit：LED 0 1bit：LED 1 …… 7bit：LED 7
0x40000010	外部 SWITCH	0bit：Switch 0 1bit：Switch1 …… 7bit：Switch7
0x40000014	七段数码管	0bit：CA 1bit：CB …… 7bit：DP 8bit：AN0 9bit：AN1 10bit：AN2 11bit：AN3
0x40000018~0x40000023	UART	UART 外设地址

图 3 外设地址

对于 LED 的写入可以用以设置 LED 的亮暗，用来表示程序运算的最终结果。对数码管的写入则满足一定规则：高 4 位表示哪一个数码管亮起，低 8 位则存储生成数字的数码管电平；数码管的扫描不由 Verilog（即硬件）控制，而是由 MIPS 汇编代码进行软件意义上的扫描，因此外设只需要负责数码管读写。外部 switch 作为外设的输入设备，可以用于调试，虽然已在外设模块的设计中，但本次实验未使用其输入。

外设模块还负责一个 CPU 中的重要功能：中断。本实验中中断的产生由一个定时器模拟，定时器到相应时间即产生一个中断信号，由 MIPS 汇编代码捕捉，再进行下一步操作；定时器周期即为中断周期。在控制信号 TCON 为 3 时，定时器在每一时钟周期进行累加，从汇编代码设定的 TH 不断累加到 32'hffffffff，再恢复到 TH。通过写寄存器改变 TH 的值即可改变中断频率。由于数码管的更新是在汇编代码的中断处理中进行，改变计时器参数就可以改变数码管扫描频率。当定时器触发中断，CPU 就会进入内核态，进行数码管更新等中断处理。

3. UART（完成人：应睿）

UART 作为接收程序输入和发送程序输出的模块，由外设模块进行统一控制。对 UART 地址的写入操作会触发数据传输，对 UART 地址的读取则可以获得之前接收的数据。

UART 的控制信号被分为五个位，分别是 tx_status（发送状态），rx_flag（接受完成），tx_flag（发送完成），rx_enable（开启接收），tx_enable（开启发送）。在本次实验中，rx_enable 默认开启，tx_flag 则基本不被用到。当 UART 模块处于发送状态，tx_status 会处于高电平；当 UART 接收到一个新的数据包，rx_flag 会置 1，但是一旦该数据被读取，rx_flag 恢复为 0；每当 UART 对应地址被写入，tx_enable 被给予一个脉冲，指示 UART 模块开始发送工作。

rx_status（未包含在控制信号中，仅用于外设模块的控制）和 tx_status 由 UART 模块内部控制并输出到外设模块中，用于判断 UART 的状态：当 rx_status 为 1，则置 rx_flag 为 1；当 tx_status 为 1，则置 tx_flag 为 1。将状态信号（status）与指示信号（flag）分开的好处是：状态信号可由 UART 模块内部改变，而指示信号可由外设模块改变，避免了时钟多驱动的问题，使 UART 和外设充分解耦。

4. 汇编器（完成人：邓程昊）

汇编器负责将 MIPS 代码翻译成对应的机器码，本质上是一个字符串处理的问题，考虑到实际需要，最终选择了 Python 语言来实现，这是因为 Python 作为脚本语言处理许多问题都很方便，而且还可以调用各种强大的库。

由于每条 MIPS 指令都是由几个关键词组成的，彼此之间是用逗号和空格分隔开来，因此可以调用 Python 的 re 模块进行正则化处理，从而得到分词后的结果，这些关键词只有三类：指令种类、寄存器编号以及标签。首先将整段 MIPS 代码逐行遍历一遍，如果以":"结尾则为标签，其余均为指令，按顺序得到标签和指令对应的 PC 值，再对所有的指令进行生成机器码的操作：先用正则表达式进行分词处理，再根据指令种类进行生成对应的字段，组合在一起得到机器码。

需要注意的是，现在的汇编器对 MIPS 代码有比较严苛的要求，例如不支持注释，不允许空行，标签处必须换行。尽管可以实现生成机器码的功能，但是其鲁棒性还有很大的提升空间。

得到的机器码以及对应 MIPS 代码的注释直接以存储器中的命令的形式存储在“ROM.v”的 Verilog 代码的文件中。

5. MIPS 汇编程序（完成人：徐泽来）

MIPS 汇编程序可以分为主程序，中断处理程序，和异常处理程序；其中主程序完成定时器配置，UART 轮询接收，求最大公约数，和 UART 发送与 LED 显示；中断处理程序完成定时器中断禁止与中断状态清零，数码管扫描显示，七段数码管译码，和使能中断；异常处理程序采用死循环处理。下面详细介绍各处理程序的实现方式。

(1) 主程序

(a) 定时器配置：关闭定时器，TCON 写入 0；设置定时器周期，TH 写入-30000；设置定时器 TL 为-1；启动定时器，TCON 写入 3；

(b) UART 轮询接收：该部分包括两个循环，分别实现两个操作数的 UART 轮询接收，下仅说明第一个操作数的轮询接收过程，第二个同。循环入口处读取 UART 状态，即读取 UART_CON；判断是否进入接收中断，即判断 UART_CON[3]是否为 1，若否，则跳回循环入口继续轮询；若是，则读取接收到的数据到$a0 寄存器，即读取 UART_RXD；

(c) 求最大公约数：在 MIPS 实现中采取了相对简单的辗转相减法(?)，即对两个操作数循环求差，并将较大的操作数更新为差值，直至两个操作数的值相同。在实现中需要注意判断两个操作数的大小关系，并在必要时对其进行交换。

(d) UART 发送与 LED 显示：向 UART_TXD 写入待发送数据，即可触发新的 UART 发送；向 LED 外设对应地址写入待发送数据。

(2) 中断处理程序

(a) 定时器中断禁止中断状态清零：即将 TCON 的 1-2bit 清零，通过 TCON&0xffff_ffff9 实现；

(b) 数码管扫描显示：读取当前数码管状态，其中高 4 位控制点亮哪个数码管，低 8 位控制数码管显示；采用类似 switch-case 语句的方式判断当前点亮的是哪个数码管，并进入响应的处理函数；在处理函数中，先根据数码管读取待显示数据，再调用 BCD 译码函数修改低 8 位数据，最后将当前点亮的数码管左移一位；最后将更新后的数码管状态写入对应的外设地址；

(c) 七段数码管译码：采用类似 switch-case 语句的方式判断待显示的数据，并

直接跳转至相应位置译码赋值；

(d) 使能中断：即将 TCON[1]置 1，通过 TCON|0x0000_0002 实现。

(3) 异常处理程序

采取死循环处理。

6. 单周期处理器（完成人：徐泽来）

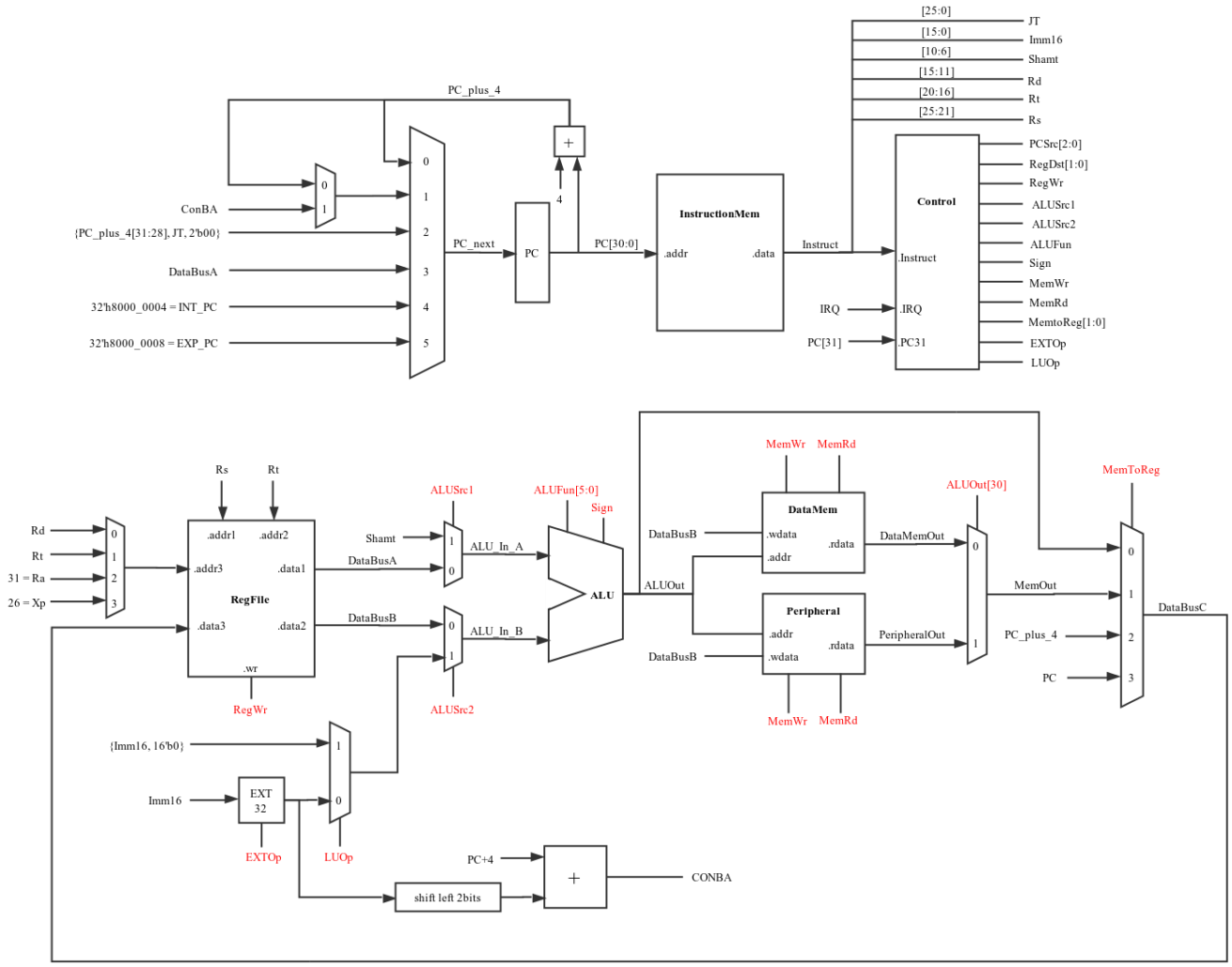


图 4 单周期处理器设计图（原图见附件）

单周期 CPU 设计的主要任务是完成 PC 的正确更新，并构建连接完整的数据通路；其工作流程可分为取指(IF)、译码(ID)、执行 (EX)、读写存储器(MEM)、和写回(WB) 五个阶段；如图 4，其硬件模块包括指令存储器、控制单元、寄存器堆、ALU、数据存储器、外设 6 个子模块。下面以工作流程的顺序详细分析单周期 CPU 的设计与原理。

取指(IF)：在每个时钟上升沿，将 PC_next 的值写入 PC 寄存器；完成 PC+4 的运算；将 PC 低 31 位的值作为地址输入指令存储器，读出对应地址的指令 Instruct；

译码(ID)：从指令 Instruc 对应位读出当前指令的 JT, Imm16, Shamt,Rd, Rt, Rs 值；将指令 Instruct 和中断控制信号 IRQ 作为输入给进控制单元，生成 PCSrc, RegDst 等控制信号；将 Rs 和 Rt 作为寄存器地址输入寄存器堆，读出对应地址的数据 DataBusA 和 DataBusB；完成 16 位立即数 Imm16 的有/无符号扩展；完成 lui 的 32 位立即数低 16 位填充；完成条件分支地址 ConBA 的计算；

执行(EX)：根据 ALUSrc1 和 ALUSrc2 控制信号选择输入 ALU 的计算数；根据 ALUFun

和 Sign 控制信号完成 ALU 内的计算，并输出计算结果 ALUOut

读写存储器(MEM)：将 ALUOut 作为地址，DataBusB 作为写入数据分别输入数据存储器和外设；数据存储器和外设根据 MemRd 和 MemWr 控制信号完成存储器和外设的读写，并分别输出读出数据 DataMemOut 和 PeripheralOut；将 ALUOut 的第 30 位作为控制信号选择数据寄存器输出或外设输出为存储器输出数据；

写回(WB)：根据 MemToReg 控制信号选择正确的写回数据 DataBusC；根据 RegDst 控制信号选择正确的写回寄存器地址 Rw；将 DataBusC 和 Rw 作为输入，根据 RegWr 控制信号判断是否进行寄存器堆的写入；根据 PCSrc 控制信号完成下一周期 PC_next 信号值的更新。

7. 5 级流水线处理器（完成人：邓程昊）

流水线设计所依据的是 D·A·Patterson 和 J·L·Hennesy 在《Computer Organization and Design: The Hardware/Software Interface, Fifth Edition》中介绍的五级流水线结构，其大体结构如图 5 所示。整个流水线共分为 IF、ID、EX、MEM 和 WB 五段，这五段间由四组段间寄存器分隔开来，分别是 IF/ID、ID/EX、EX/MEM、MEM/WB，整个流水线除了各个模块的某些接口与段间寄存器以外没有任何寄存器，均为透明传输，这样大大简化了调试工作、也提升了流水线的延时性能。

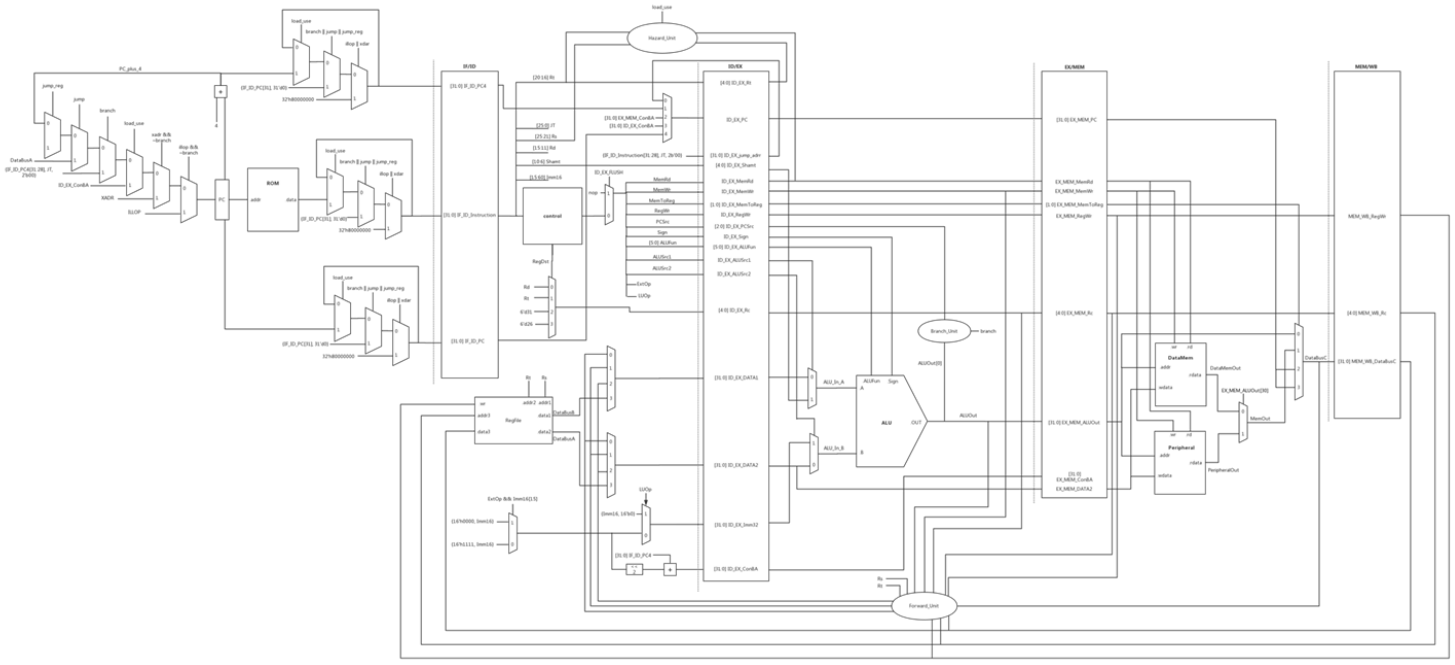


图 5 流水线处理器设计图（原图见附件）

IF 段主要完成 PC 的更新，这主要是通过若干级多路选择器级联得到正确的 PC，一方面将 PC 加 4 得到 PC_plus_4，另一方面从指令存储器 ROM 中取得 PC 对应的 Instruction，PC、PC_plus_4 和 Instruction 都将作为 IF/ID 段寄存器的输入。

ID 段主要完成指令的翻译，一方面控制器 Control 会根据 IF_ID_Instruction 生成对应的控制信号，另一方面从 Instruction 本身也可以翻译得到有关寄存器、移位以及立即数的信号，可以直接由这些信号生成的信号都会直接在 ID 段生成，与其他控制信号或之后各段需要的其他信号一起作为 ID/EX 段寄存器的输入。

EX 段主要完成 ALU 的运算和 branch 信号的生成，ALU 的运算与单周期时基本一致，输入的信号为 ID/Ex 段寄存器中的对应值。同时 branch 单元会根据指令的类型与 ALU 输出的结果判断是否有分支发生，进而输出 branch 信号。除此之外，一些会在后

面用到的信号也会被继续传递下去，与 ALU 的输出一起作为 EX/MEM 段寄存器的输入。

MEM 段主要完成与存储器有关的操作，这里涉及到的存储器有数据存储器 DataMem 和外设 Perpherl 两种，取字指令时两个存储器都会取字，再根据存取地址的高位进行选择。除此之外，还会根据 EX_MEM_MemToReg 决定写入寄存器的数据 DataBusC，最终将 DataBusC 和 Rc、RegWr 作为 MEM/WB 段寄存器的输入。

WB 段主要完成寄存器堆的写回，根据 MEM/WB 段寄存器的值即可完成，相对比较简单。

除此之外，还有两个比较重要的单元：Forward 单元和 Hazard 单元。Forward 单元主要通过判断各段段间寄存器中写入寄存器与读取寄存器中的值来决定如何进行转发的操作，所有可能转发都会被发送到 ID/EX 段寄存器对应的 DATA，再输入到寄存器之前有多路选择器根据 Forward 单元给出的信号进行判断，决定 ID/EX_DATA 的输入。Harzard 单元则是通过判断 ID/EX 段是否发生存储器的读取，以及各段寄存器之间的关系来决定是否发生了 load_use 冒险，如果发生，则可以通过将控制器 control 的输出信号置零，从而起到阻塞的效果。

需要说明的是，由于采取的是行为级设计，在 Verilog 代码中并没有直接描述 Branch 单元、Forward 单元和 Hazard 单元，而是通过条件运算符和条件语句完成的。这样可以让 Vivado 在综合和仿真中有更大的优化空间。

三、关键代码与文件清单

（完成人：应睿）

1. ALU 模块

```
assign OUT_00 = A + (ALUFun[0]? (~B) + 32'h00000001: B);

assign nonzero = | OUT_00;

always @(*)
    case ({A[31], B[31]})
        2'b10: negative <= Sign;
        2'b01: negative <= ~Sign;
        default: negative <= OUT_00[31];
    endcase

always @(*)
    case ({ALUFun[2], Sign})
        2'b11: compare_zero <= A[31] | ~(|A);
        2'b10: compare_zero <= ~(|A);
        2'b01: compare_zero <= A[31];
        default: compare_zero <= 0;
    endcase

always @(*)
    case (ALUFun[3:2])
        2'b11: OUT_11 <= compare_zero ^ ALUFun[1];
```

```
            2'b10: OUT_11 <= compare_zero;
            2'b01: OUT_11 <= negative;
            default: OUT_11 <= nonzero ^ ALUFun[1];
        endcase

    always @(*)
        case (ALUFun[3:0])
            4'b1000: OUT_01 <= A & B;
            4'b1110: OUT_01 <= A | B;
            4'b0110: OUT_01 <= A ^ B;
            4'b0001: OUT_01 <= ~(A | B);
            default: OUT_01 <= A;
        endcase

    assign B_left_1 = A[0]? {B[30:0], 1'h0}: B;
    assign B_left_2 = A[1]? {B_left_1[29:0], 2'h0}: B_left_1;
    assign B_left_4 = A[2]? {B_left_2[27:0], 4'h0}: B_left_2;
    assign B_left_8 = A[3]? {B_left_4[23:0], 8'h0}: B_left_4;
    assign B_left = A[4]? {B_left_8[15:0], 16'h0}: B_left_8;

    assign B_right_1 = A[0]? {1'h0, B[31:1]}: B;
    assign B_right_2 = A[1]? {2'h0, B_right_1[31:2]}: B_right_1;
    assign B_right_4 = A[2]? {4'h0, B_right_2[31:4]}: B_right_2;
    assign B_right_8 = A[3]? {8'h0, B_right_4[31:8]}: B_right_4;
    assign B_right = A[4]? {16'h0, B_right_8[31:16]}: B_right_8;

    assign B_right_a_1 = A[0]? {1'b1, B[31:1]}: B;
    assign B_right_a_2 = A[1]? {2'b11, B_right_a_1[31:2]}: B_right_a_1;
    assign B_right_a_4 = A[2]? {4'b1111, B_right_a_2[31:4]}: B_right_a_2;
    assign B_right_a_8 = A[3]? {8'b1111_1111, B_right_a_4[31:8]}: B_right_a_4;
    assign B_right_a = A[4]? {16'b1111_1111_1111_1111, B_right_a_8[31:16]}:
B_right_a_8;

    always @(*)
        if(ALUFun[0])
        begin
            if(ALUFun[1] & B[31])
                OUT_10 <= B_right_a;
            else
                OUT_10 <= B_right;
        end
        else
            OUT_10 <= B_left;
```

```
    always @(*)
        case (ALUFun[5:4])
            2'b00: OUT <= OUT_00;
            2'b11: OUT <= {31'h00000000, OUT_11};
            2'b01: OUT <= OUT_01;
            2'b10: OUT <= OUT_10;
            default: OUT <= 32'h00000000;
        endcase
```

2. Peripheral 模块

```
// Interruption signal
assign irqout = TCON[2];

// Timer
always @(negedge reset or posedge clk) begin
    if (~reset) begin
        TH <= 32'b0;
        TL <= 32'b0;
        TCON <= 3'b0;
        digi <= 12'b0;
        led <= 8'b0;
        tx_flag <= 0;
        rx_flag <= 0;
        tx_enable <= 0;
        rx_enable <= 1;
        count <= 0;
    end
    else begin
        // Timer enabled
        if (TCON[0]) begin
            if (TL == 32'hffffffff) begin
                TL <= TH;
                // Interruption enabled
                if (TCON[1]) TCON[2] <= 1'b1;
            end
            else TL <= TL + 1;
        end

        // Write requires one cycle to be done; use Timer's clk
        if (wr) begin
            case (addr)
                32'h40000000: TH <= wdata;
                32'h40000004: TL <= wdata;
                32'h40000008: TCON <= wdata[2:0];
```

```
                32'h4000000C: led <= wdata[7:0];
                32'h40000014: digi <= wdata[11:0];
                32'h40000018: begin tx_data <= wdata[7:0]; tx_enable <= 1; end
                default: ;
            endcase
        end

        // Set flags to true when finished sending or receiving
        if (~rx_status)
            count <= 0;
        if (rx_status && ~count) begin
            rx_flag <= 1;
            count <= 1;
        end
        if(tx_status)
            tx_flag <= 1;

        //  Set flags to false when data is read
        if(rd && (addr == 32'h4000001C))
            rx_flag <= 0;
        if(rd && (addr == 32'h4000001C))
            tx_flag <= 0;

        // Only send once
        if (tx_enable == 1) tx_enable <= 0;
    end
end

// Read can be straight
always @(*) begin
    if (rd) begin
        case (addr)
            32'h40000000: rdata <= TH;
            32'h40000004: rdata <= TL;
            32'h40000008: rdata <= {29'b0, TCON};
            32'h4000000C: rdata <= {24'b0, led};
            32'h40000010: rdata <= {24'b0, switch};
            32'h40000014: rdata <= {20'b0, digi};
            32'h40000018: rdata <= {24'b0, tx_data};
            32'h4000001C: rdata <= {24'b0, rx_data};
            32'h40000020: rdata <= {27'b0, tx_status, rx_flag, tx_flag,
rx_enable, tx_enable};
            default: rdata <= 32'b0;
        endcase
```

```
    end
    else
        rdata <= 32'b0;
end
```

3. Control 模块

```
    assign OpCode = Instruct[31:26];
    assign Funct = Instruct[5:0];
    assign IRQ_valid = IRQ & (~PC31);
    assign Undefine = PC31 ? 0 :
            ~((OpCode >= 6'h00 && OpCode <= 6'h0c) || OpCode == 6'h0f || OpCode
== 6'h23 || OpCode == 6'h2b) ? 1 :
            ((OpCode != 0) || ((OpCode == 0) && (Funct[5:3] >= 3'b100 || Funct
== 6'h00 || Funct == 6'h02 || Funct == 6'h03 || Funct == 6'h08 || Funct ==
6'h09 || Funct == 6'h2a))) ? 0 : 1;
    // PCSrc
    always @(*)
        if(IRQ_valid)
            PCSrc <= 3'd4;
        else if(Undefine)
            PCSrc <= 3'd5;
        else if(OpCode == 6'h01 || OpCode >= 6'h04 && OpCode <= 6'h07)
            PCSrc <= 3'd1;
        else if(OpCode == 6'h02 || OpCode == 6'h03)
            PCSrc <= 3'd2;
        else if(OpCode == 6'h00 && (Funct == 6'h08 || Funct == 6'h09))
            PCSrc <= 3'd3;
        else
            PCSrc <= 3'd0;
    // RegDst
    always @(*)
        if(IRQ_valid || Undefine)
            RegDst <= 2'd3;
        else if(OpCode == 6'h03 || OpCode == 6'h00 && Funct == 6'h09)
            RegDst <= 2'd2;
        else if(OpCode == 6'h00)
            RegDst <= 2'd0;
        else
            RegDst <= 2'd1;
    // RegWr
    always @(*)
        if(IRQ_valid || Undefine)
            RegWr <= 1;
```

```
        else if(Instruct == 32'h0000_0000 || OpCode == 6'h2b || (OpCode ==
6'h00 && Funct == 6'h08) || (OpCode >= 6'h01 && OpCode <= 6'h07 && OpCode !=
6'h03))
            RegWr <= 0;
        else
            RegWr <= 1;
    // MemToReg
    always @(*)
        if(IRQ_valid)
            MemToReg <= 2'd3;
        else if(Undefine || OpCode == 6'h00 && Funct == 6'h09 || OpCode ==
6'h03)
            MemToReg <= 2'd2;
        else if(OpCode == 6'h23)
            MemToReg <= 2'd1;
        else
            MemToReg <= 2'd0;
    // ALUFun
    always @(*)
        if(OpCode == 6'h00)
            case(Funct)
                6'h00: ALUFun <= 6'b100000;
                6'h02: ALUFun <= 6'b100001;
                6'h03: ALUFun <= 6'b100011;
                6'h20: ALUFun <= 6'b000000;
                6'h21: ALUFun <= 6'b000000;
                6'h22: ALUFun <= 6'b000001;
                6'h23: ALUFun <= 6'b000001;
                6'h24: ALUFun <= 6'b011000;
                6'h25: ALUFun <= 6'b011110;
                6'h26: ALUFun <= 6'b010110;
                6'h27: ALUFun <= 6'b010001;
                6'h2a: ALUFun <= 6'b110101;
                default: ALUFun <= 6'b000000;
            endcase
        else
            case(OpCode)
                6'h01: ALUFun <= 6'b111011;
                6'h04: ALUFun <= 6'b110011;
                6'h05: ALUFun <= 6'b110001;
                6'h06: ALUFun <= 6'b111101;
                6'h07: ALUFun <= 6'b111111;
                6'h08: ALUFun <= 6'b000000;
                6'h09: ALUFun <= 6'b000000;
```

```
                6'h0a: ALUFun <= 6'b110101;
                6'h0b: ALUFun <= 6'b110101;
                6'h0c: ALUFun <= 6'b011000;
                6'h0f: ALUFun <= 6'b000000;
                6'h23: ALUFun <= 6'b000000;
                6'h2b: ALUFun <= 6'b000000;
                default: ALUFun <= 6'b000000;
            endcase

    assign ALUSrc1 = (OpCode == 6'h00 && (Funct == 6'h00 || Funct == 6'h02 ||
Funct == 6'h03))? 1 : 0;
    assign ALUSrc2 = (OpCode >= 6'h08) ? 1 : 0;
    assign Sign = (OpCode == 6'h0b) ? 0 : 1;
    assign MemWr = (OpCode == 6'h2b) ? 1 : 0;
    assign MemRd = (OpCode == 6'h23) ? 1 : 0;
    assign ExtOp = (OpCode == 6'h0c) ? 0 : 1;
    assign LUOp = (OpCode == 6'h0f) ? 1 : 0;
```

4. SingleCycleCPU

```
wire [31:0] PC_plus_4;
assign PC_plus_4 = PC + 32'h4;

wire [31:0] Instruct;
ROM instructionMem(.addr(PC[30:0]), .data(Instruct));

wire [25:0] JT;
wire [15:0] Imm16;
wire [4:0] Shamt;
wire [4:0] Rd;
wire [4:0] Rt;
wire [4:0] Rs;
assign JT = Instruct[25:0];
assign Imm16 = Instruct[15:0];
assign Shamt = Instruct[10:6];
assign Rd = Instruct[15:11];
assign Rt = Instruct[20:16];
assign Rs = Instruct[25:21];

wire IRQ;
wire [2:0] PCSrc;
wire [1:0] RegDst;
wire RegWr;
wire ALUSrc1;
wire ALUSrc2;
```

```
wire [5:0] ALUFun;
wire Sign;
wire MemWr;
wire MemRd;
wire [1:0] MemToReg;
wire ExtOp;
wire LUOp;
Control control(.Instruct(Instruct), .IRQ(IRQ), .PC31(PC[31]),
                .PCSrc(PCSrc), .RegDst(RegDst), .RegWr(RegWr), .ALUSrc1(ALUSrc1
),
                .ALUSrc2(ALUSrc2), .ALUFun(ALUFun), .Sign(Sign), .MemWr(MemWr),
                .MemRd(MemRd), .MemToReg(MemToReg), .ExtOp(ExtOp), .LUOp(LUOp))
;

wire [4:0] Rw;
assign Rw = (RegDst == 2'b00) ? Rd :
            (RegDst == 2'b01) ? Rt :
            (RegDst == 2'b10) ? 5'b11111 :
            (RegDst == 2'b11) ? 5'b11010 : 5'b00000;

wire [31:0] DataBusA;
wire [31:0] DataBusB;
wire [31:0] DataBusC;
RegFile
regFile(.reset(reset), .clk(clk), .addr1(Rs), .data1(DataBusA),.addr2(Rt),
                .data2(DataBusB), .wr(RegWr), .addr3(Rw), .data3(DataBusC));

wire [31:0] Imm32;
wire [31:0] ALU_In_A;
wire [31:0] ALU_In_B;
assign Imm32 = (ExtOp && Imm16[15] ) ? {16'hffff, Imm16} : {16'h0000, Imm16};
assign ALU_In_A = ALUSrc1 ? Shamt : DataBusA;
assign ALU_In_B = ALUSrc2 ? (LUOp ? {Imm16, 16'b0} : Imm32) : DataBusB;

wire [31:0] ALUOut;
ALU
alu(.A(ALU_In_A), .B(ALU_In_B), .ALUFun(ALUFun), .Sign(Sign), .OUT(ALUOut));

wire [31:0] DataMemOut;
DataMem dataMem(.reset(reset), .clk(clk), .rd(MemRd), .wr(MemWr),
                .addr(ALUOut), .wdata(DataBusB), .rdata(DataMemOut));

wire [31:0] PeripheralOut;
```

```
Peripheral
peripheral(.sysclk(sysclk), .reset(reset), .clk(clk), .rd(MemRd), .wr(MemWr), .
addr(ALUOut),
                    .wdata(DataBusB), .rdata(PeripheralOut), .led(led), .swit
ch(switch),
                    .digi(digi), .irqout(IRQ), .PC_Uart_rxd(UART_RX), .PC_Uar
t_txd(UART_TX));

wire [31:0] MemOut;
assign MemOut = ALUOut[30] ? PeripheralOut : DataMemOut;

assign DataBusC = (MemToReg == 2'b00) ? ALUOut :
                  (MemToReg == 2'b01) ? MemOut :
                  (MemToReg == 2'b10) ? PC_plus_4 :
                  (MemToReg == 2'b11) ? PC : PC_plus_4;

wire [31:0] ConBA;
assign ConBA = PC_plus_4 + {Imm32[29:0], 2'b00};
assign PC_next = (PCSrc == 3'b000) ? PC_plus_4 :
                 (PCSrc == 3'b001) ? (ALUOut[0] ? ConBA : PC_plus_4) :
                 (PCSrc == 3'b010) ? {PC_plus_4[31:28], JT, 2'b00} :
                 (PCSrc == 3'b011) ? DataBusA :
                 (PCSrc == 3'b100) ? INT_PC :
                 (PCSrc == 3'b101) ? EXP_PC : EXP_PC;
```

5. PipelineCPU

```
// IF
assign PC_plus_4 = PC + 3'd4;

wire [31:0] PC_next;
assign PC_next = (illop && ~branch) ? ILLOP :
                 (xadr && ~branch) ? XADR :
                 load_use ? PC :
                 branch ? ID_EX_ConBA :
                 jump ? {IF_ID_PC4[31:28], JT, 2'b00} :
                 jump_reg ? DataBusA : PC_plus_4;

always @(posedge clk or negedge reset) begin
    if(~reset)
        PC <= 32'd0;
    else
        PC <= PC_next;
end
```

```
ROM instruction_memory(.addr(PC[30:0]), .data(Instruction));

// IF/ID
wire [31:0] IF_ID_Instruction_next;
wire [31:0] IF_ID_PC_next;
wire [31:0] IF_ID_PC4_next;
assign IF_ID_Instruction_next = (illop ||xadr || branch || jump ||jump_reg) ?
32'd0 :
                                ~load_use ? Instruction : IF_ID_Instruction;
assign IF_ID_PC_next = (illop || xadr) ? 32'h8000_0000 :
                       (branch || jump || jump_reg) ? {IF_ID_PC[31], 31'd0} :
                       (~load_use) ? PC : IF_ID_PC;
assign IF_ID_PC4_next = (illop || xadr) ? 32'h8000_0000 :
                       (branch || jump || jump_reg) ? {IF_ID_PC4[31], 31'd0} :
                       (~load_use) ? PC_plus_4 : IF_ID_PC4;

always @(posedge clk or negedge reset) begin
    if(~reset) begin
        IF_ID_Instruction <= 32'd0;
        IF_ID_PC <= 32'd0;
        IF_ID_PC4 <= 32'd0;
    end
    else begin
        IF_ID_Instruction <= IF_ID_Instruction_next;
        IF_ID_PC <= IF_ID_PC_next;
        IF_ID_PC4 <= IF_ID_PC4_next;
    end
end

// ID
Control control(.Instruct(IF_ID_Instruction), .IRQ(IRQ), .PC31(IF_ID_PC4[31]),
                .PCSrc(PCSrc), .RegDst(RegDst), .RegWr(RegWr), .ALUSrc1(ALUSrc1
), .ALUSrc2(ALUSrc2),
                .ALUFun(ALUFun), .Sign(Sign), .MemWr(MemWr), .MemRd(MemRd), .Me
mToReg(MemToReg),
                .ExtOp(ExtOp), .LUOp(LUOp));

assign JT = IF_ID_Instruction[25:0];
assign Imm16 = IF_ID_Instruction[15:0];
assign Imm32 = (ExtOp && Imm16[15]) ? {16'hffff, Imm16} : {16'h0000, Imm16};
assign Shamt = IF_ID_Instruction[10:6];
assign Rd = IF_ID_Instruction[15:11];
assign Rt = IF_ID_Instruction[20:16];
assign Rs = IF_ID_Instruction[25:21];
```

```
assign Rc = (RegDst == 2'b00)? Rd:
            (RegDst == 2'b01)? Rt:
            (RegDst == 2'b10)? 5'd31:
            (RegDst == 2'b11)? 5'd26:5'd0;
assign ConBA = {Imm32[29:0], 2'd0} + IF_ID_PC4;
assign load_use = (ID_EX_MemRd && (ID_EX_Rt == Rs || ID_EX_Rt == Rt));
assign jump = (PCSrc == 3'd2);
assign jump_reg = (PCSrc == 3'd3);
assign illop = (PCSrc == 3'd4);
assign xadr = (PCSrc == 3'd5);

RegFile
regfile(.reset(reset), .clk(clk), .addr1(Rs), .data1(DataBusA), .addr2(Rt), .da
ta2(DataBusB),
            .wr(MEM_WB_RegWr), .addr3(MEM_WB_Rc), .data3(MEM_WB_DataBusC));
// ID/EX
always @(posedge clk or negedge reset) begin
    if(~reset) begin
        ID_EX_DATA1 <= 32'd0;
        ID_EX_DATA2 <= 32'd0;
        ID_EX_PC <= 32'd0;
        ID_EX_Imm32 <= 32'd0;
        ID_EX_ConBA <= 32'd0;
        ID_EX_PCSrc <= 2'd0;
        ID_EX_Shamt <= 5'd0;
        ID_EX_Rs <= 5'd0;
        ID_EX_Rt <= 5'd0;
        ID_EX_Rc <= 5'd0;
        ID_EX_RegWr <= 0;
        ID_EX_ALUSrc1 <= 0;
        ID_EX_ALUSrc2 <= 0;
        ID_EX_Sign <= 0;
        ID_EX_MemWr <= 0;
        ID_EX_MemRd <= 0;
        ID_EX_MemToReg <= 2'd0;
        ID_EX_ALUFun <= 6'd0;
    end
    else begin
        ID_EX_DATA1 <= (EX_MEM_RegWr && (EX_MEM_Rc != 5'd0) && (EX_MEM_Rc ==
Rs)
                        && (ID_EX_Rc != Rs || ~ID_EX_RegWr))? DataBusC:
                        (ID_EX_RegWr && (ID_EX_Rc != 5'd0) && (ID_EX_Rc ==
Rs)) ? ALUOut:
```

```
                        (MEM_WB_RegWr && (MEM_WB_Rc != 5'd0) && (MEM_WB_Rc ==
Rs)) ? MEM_WB_DataBusC:DataBusA;
        ID_EX_DATA2 <= (EX_MEM_RegWr && (EX_MEM_Rc != 5'd0) && (EX_MEM_Rc ==
Rt)
                        && (ID_EX_Rc != Rt || ~ID_EX_RegWr))? DataBusC:
                        (ID_EX_RegWr && (ID_EX_Rc != 5'd0) && (ID_EX_Rc ==
Rt)) ? ALUOut:
                        (MEM_WB_RegWr && (MEM_WB_Rc != 5'd0) && (MEM_WB_Rc ==
Rt)) ? MEM_WB_DataBusC:DataBusB;
        ID_EX_PC <= (ID_EX_jump == 1) ?
ID_EX_jump_addr:((IF_ID_Instruction[31:26] ==
6'b000011)?IF_ID_PC4:((branch)?ID_EX_ConBA:(EX_MEM_branch)?EX_MEM_ConBA:IF_ID_P
C));
        ID_EX_Imm32 <= LUOp ? {Imm16, 16'd0}:Imm32;
        ID_EX_ConBA <= ConBA;
        ID_EX_PCSrc <= PCSrc;
        ID_EX_Shamt <= Shamt;
        ID_EX_Rs <= Rs;
        ID_EX_Rt <= Rt;
        if((load_use || branch) && ~xadr && ~illop) begin
            ID_EX_Rc <= 5'd0;
            ID_EX_RegWr <= 0;
            ID_EX_ALUSrc1 <= 0;
            ID_EX_ALUSrc2 <= 0;
            ID_EX_Sign <= 0;
            ID_EX_MemWr <= 0;
            ID_EX_MemRd <= 0;
            ID_EX_MemToReg <= 2'd0;
            ID_EX_ALUFun <= 5'd0;
            ID_EX_jump <= 0;
            ID_EX_jump_addr <= 32'd0;
        end
        else begin
            ID_EX_Rc <= Rc;
            ID_EX_RegWr <= RegWr;
            ID_EX_ALUSrc1 <= ALUSrc1;
            ID_EX_ALUSrc2 <= ALUSrc2;
            ID_EX_Sign <= Sign;
            ID_EX_MemWr <= MemWr;
            ID_EX_MemRd <= MemRd;
            ID_EX_MemToReg <= MemToReg;
            ID_EX_ALUFun <= ALUFun;
            ID_EX_jump <= jump;
            ID_EX_jump_addr <= {IF_ID_Instruction[31:28], JT, 2'b00};
```

```
        end
    end
end
// EX
assign ALU_In_A = ID_EX_ALUSrc1 ? {27'd0, ID_EX_Shamt} : ID_EX_DATA1;
assign ALU_In_B = ID_EX_ALUSrc2 ? ID_EX_Imm32 : ID_EX_DATA2;
assign branch = (ID_EX_PCSrc == 3'b001) && ALUOut[0];

ALU
alu(.A(ALU_In_A), .B(ALU_In_B), .ALUFun(ID_EX_ALUFun), .Sign(ID_EX_Sign), .OUT(
ALUOut));
// EX/MEM
always @(posedge clk or negedge reset) begin
    if(~reset) begin
        EX_MEM_RegWr <= 0;
        EX_MEM_MemWr <= 0;
        EX_MEM_MemRd <= 0;
        EX_MEM_branch <= 0;
        EX_MEM_PC <= 32'd0;
        EX_MEM_DATA2 <= 32'd0;
        EX_MEM_ALUOut <= 32'd0;
        EX_MEM_Rc <= 5'd0;
        EX_MEM_MemToReg <= 2'd0;
        EX_MEM_ConBA <= 31'd0;
    end
    else begin
        EX_MEM_RegWr <= ID_EX_RegWr;
        EX_MEM_MemWr <= ID_EX_MemWr;
        EX_MEM_MemRd <= ID_EX_MemRd;
        EX_MEM_branch <= branch;
        EX_MEM_PC <= ID_EX_PC;
        EX_MEM_DATA2 <= ID_EX_DATA2;
        EX_MEM_ALUOut <= ALUOut;
        EX_MEM_Rc <= ID_EX_Rc;
        EX_MEM_MemToReg <= ID_EX_MemToReg;
        EX_MEM_ConBA <= ID_EX_ConBA;
    end
end
// MEM
DataMem
datamem(.reset(reset), .clk(clk), .rd(EX_MEM_MemRd), .wr(EX_MEM_MemWr), .addr(E
X_MEM_ALUOut), .wdata(EX_MEM_DATA2), .rdata(DataMemOut));
```

```
Peripheral
peripheral_pipe(.sysclk(sysclk), .reset(reset), .clk(clk), .rd(EX_MEM_MemRd), .
wr(EX_MEM_MemWr), .addr(EX_MEM_ALUOut),
                    .wdata(EX_MEM_DATA2), .rdata(PeripheralOut), .led(led), .s
witch(switch), .digi(digi),
                    .irqout(IRQ), .PC_Uart_rxd(UART_RX), .PC_Uart_txd(UART_TX)
);

assign MemOut = EX_MEM_ALUOut[30] ? PeripheralOut : DataMemOut;
assign DataBusC = (EX_MEM_MemToReg == 2'd0)? EX_MEM_ALUOut :
                (EX_MEM_MemToReg == 2'd1)? MemOut :
                (EX_MEM_MemToReg == 2'd2)? EX_MEM_PC : EX_MEM_PC;

// MEM/WB
always @(posedge clk or negedge reset) begin
    if(~reset) begin
        MEM_WB_RegWr <= 0;
        MEM_WB_DataBusC <= 32'd0;
//        MEM_WB_PC4 <= 32'd0;
        MEM_WB_Rc <= 5'd0;
    end
    else begin
        MEM_WB_RegWr <= EX_MEM_RegWr;
//        MEM_WB_PC4 <= EX_MEM_PC4;
        MEM_WB_DataBusC <= DataBusC;
        MEM_WB_Rc <= EX_MEM_Rc;
    end
end
```

6. 文件清单

SingleCycleCPU 与 PipelineCPU 分别为单周期和流水线处理器的顶层模块，它们共用其他文件中的模块。后缀为 tb 的文件为仿真文件。gcd.asm 为实现求取最大公约数的 MIPS 汇编程序。assembler.py 为汇编器 Python 源文件。Ego1.xdc 为最后硬件实现所用的约束文件。

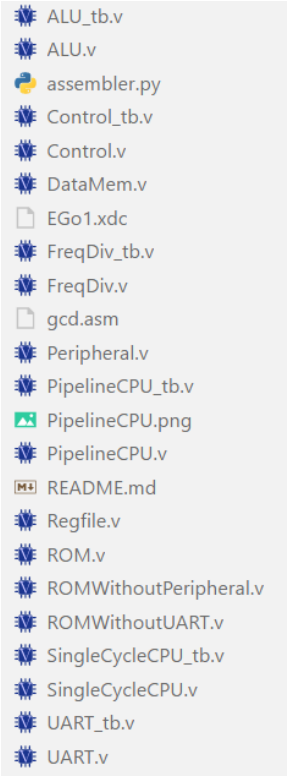


图 6 文件清单

四、仿真结果及分析

（完成人：应睿）

1. ALU 模块

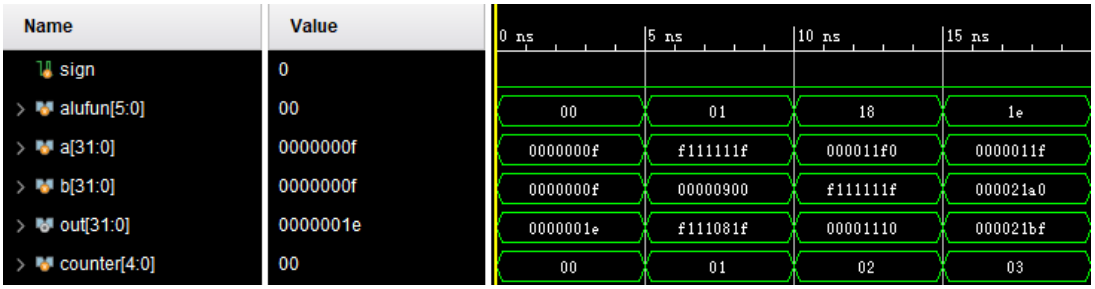


图 7 ALU 模块仿真

a) 如图 7 为例，以 alufun=01 来看，此时执行 a-b 操作，out 为 0xf111081f，而 0xf111111f-0x00000900 确实为 0xf111081f。

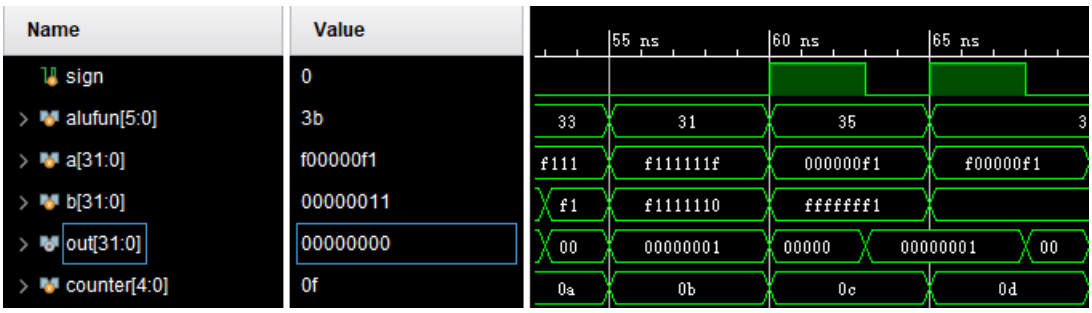


图 8 ALU 模块仿真

b) 如图 8 为例，当 alufun=35，且 sign=1，即进行有符号 LT 功能时，0x000000f1 大于负数 0xffffffff，故结果 out 为 0；当 sign=0，无符号比较，则 0xffffffff 大于 0x000000f1，out 为 1。

2. Control 模块

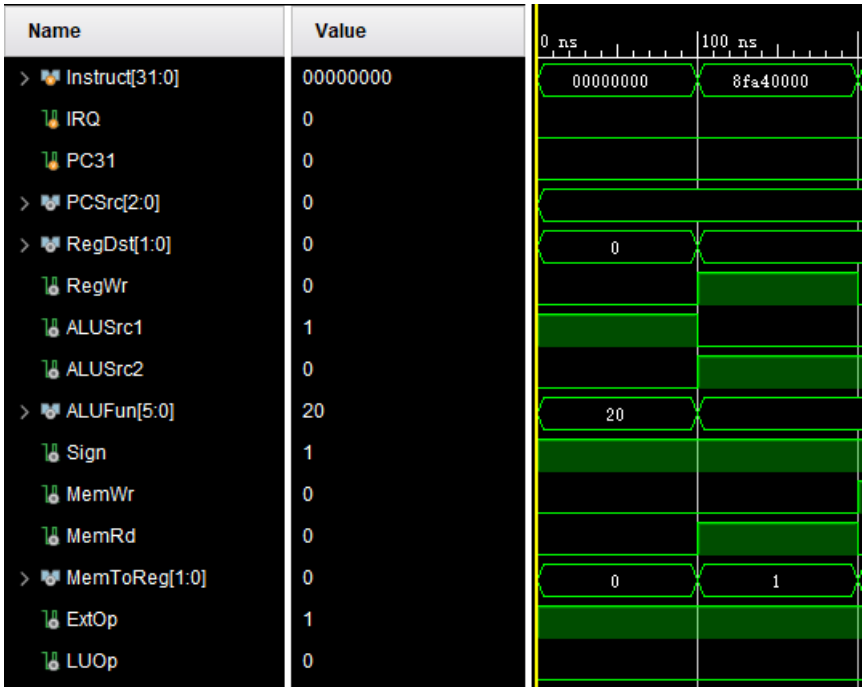


图 9 Control 模块仿真

以图 9 为例，当指令 Instruct 为 0x8fa40000，代表语句 lw $a0, 0($sp)。此时 PCSrc 为 0（未显示全），此时只从 PC 寄存器读取下一个加 4 地址的指令；RegDst 为 1（未显示全），表明写入的寄存器地址来自指令自身；ALUSrc1 为 0，ALUSrc2 为 1，表明操作数一个来自指令中（0），一个来自寄存器（$sp）；lw 指令中偏移量默认为有符号数，故 Sign、ExtOp 均为 1；MemToReg 为 1 也符合 lw 指令从 Mem 中拿取数据并存到寄存器 $a0 中的行为。

3. UART 模块

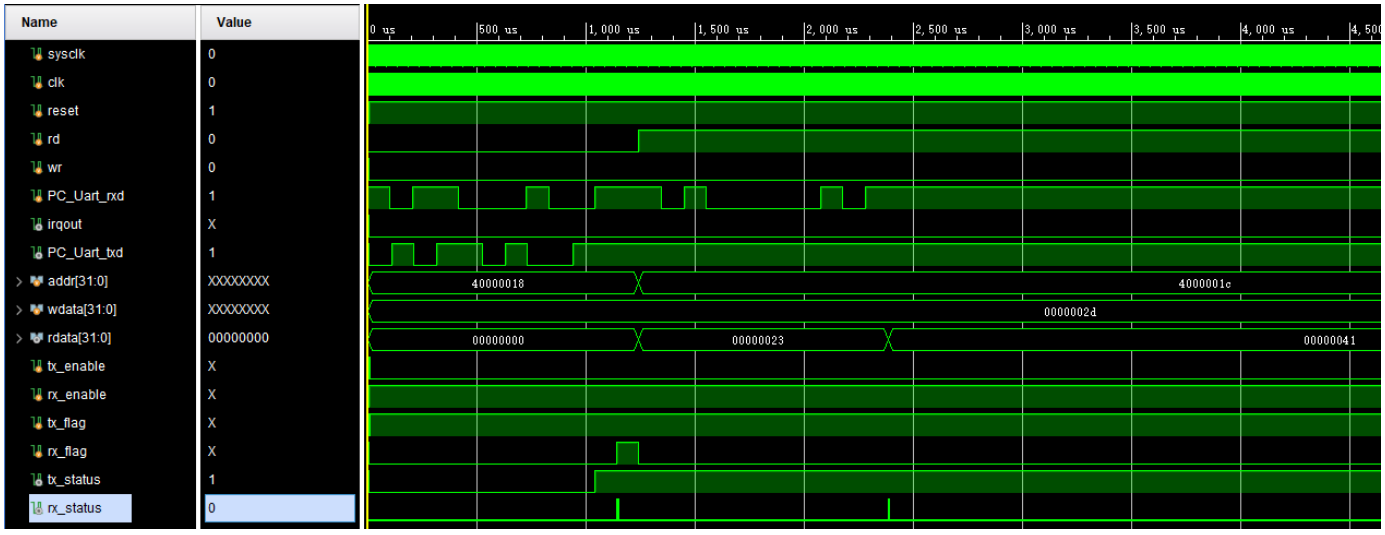


图 10 UART 模块仿真

以图 10 为例，当向特定地址写入数据时，触发 tx_enable 脉冲，PC_Uart_txd 开始发送数据，且 tx_status 保持低电平显示发送状态；当手动给 PC_Uart_rxd 传输数据，每一个数据包接收完毕，rx_status 产生脉冲表明接收完毕，rx_flag 产生高电平，且 rdata 中写入了接收到的数据；读取数据后，rx_flag 归零。

可以发现一个问题：rx_status 产生了两次脉冲，而 rx_flag 只上升了一次，但是按道理接收两个数据包应该会使得 rx_flag 上升两次。这里就是当初仿真和硬件调试冲突的地方。我们最初在 always 块中对 rx_flag 直接进行归零操作，是可以产生正确的仿真结果的，但是在硬件上却无法正常运行；后更改成以 count 为中间变量，对 rx_flag 进行另一种方式的赋值，硬件调试通过，但是仿真出现了如图所示的问题。具体原因及问题解决参见“硬件调试情况”一节。

4. 单周期处理器

对使用 ROM_without_UART 的 SingleCycleCPU 进行仿真，此时两个输入数 65 和 35 已在 ROM 中写死。

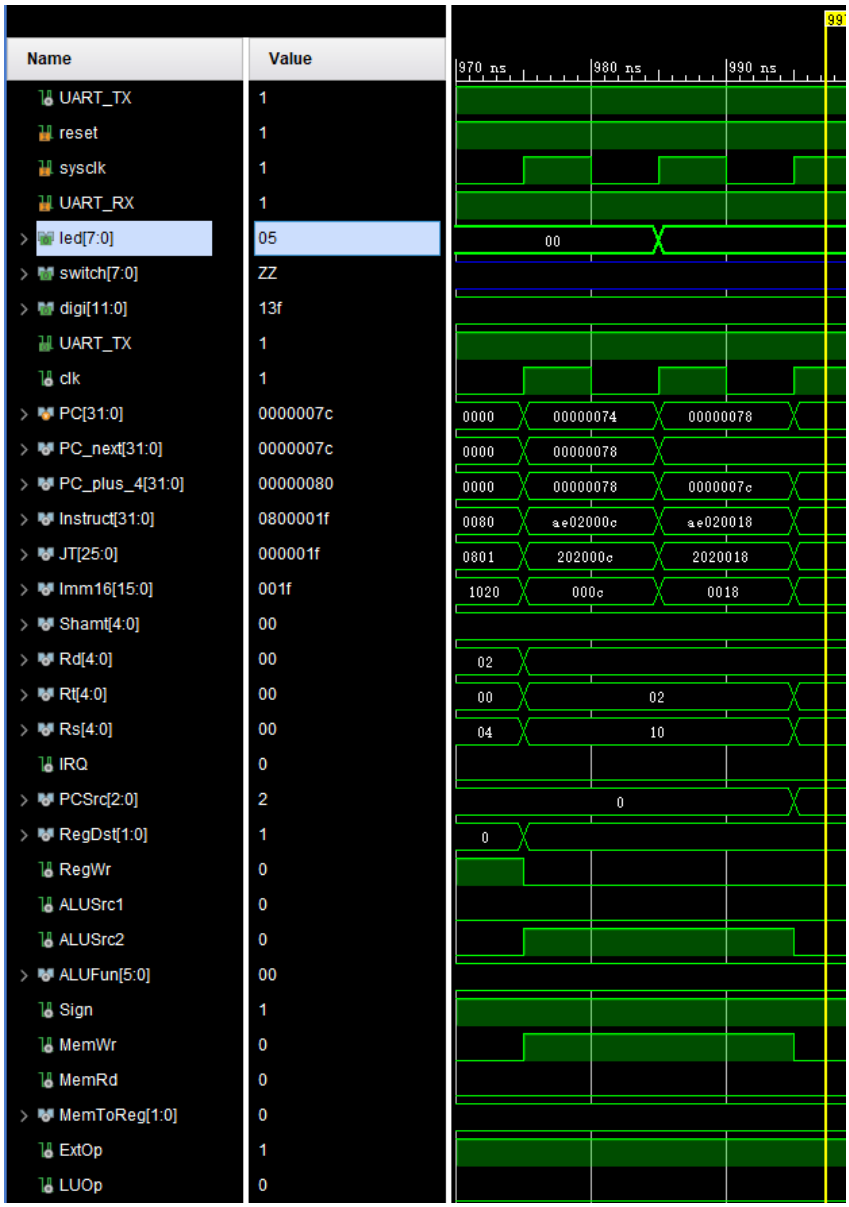


图 11 单周期处理器仿真

以图 11 为例，运算结果为 5，且结果已写入到 LED 中进行显示。最后一条指令 0xae200018 表示 sw $v0, 24($s0)，即将结果写入 LED 指令。Rt 为 0x02，Rs 为 0x10，ALUSrc2 为 1，MemWr 为 1，都符合 sw 指令的要求。

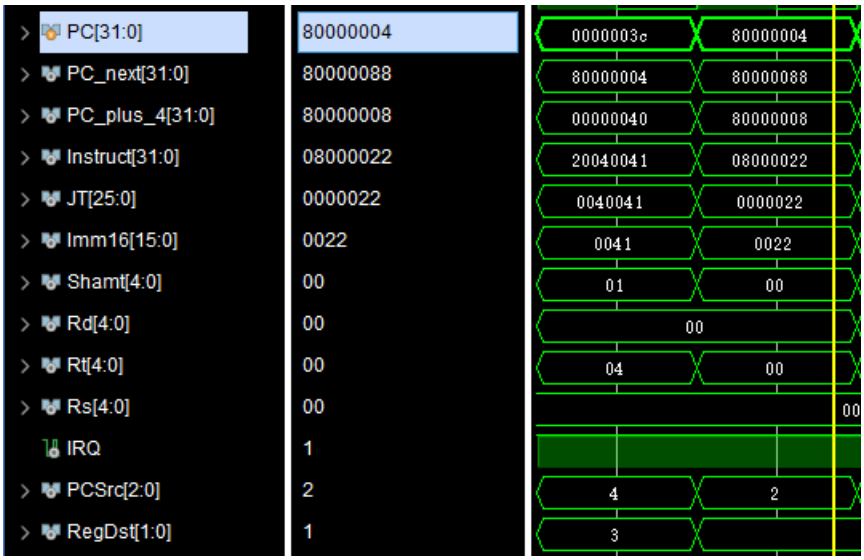


图 12 单周期处理器仿真

以图 12 为例，可以看到在指令 0x08000022（j interrupt）后，PC 最高位变为 1，CPU 进入中断时的内核态。

5. 流水线处理器

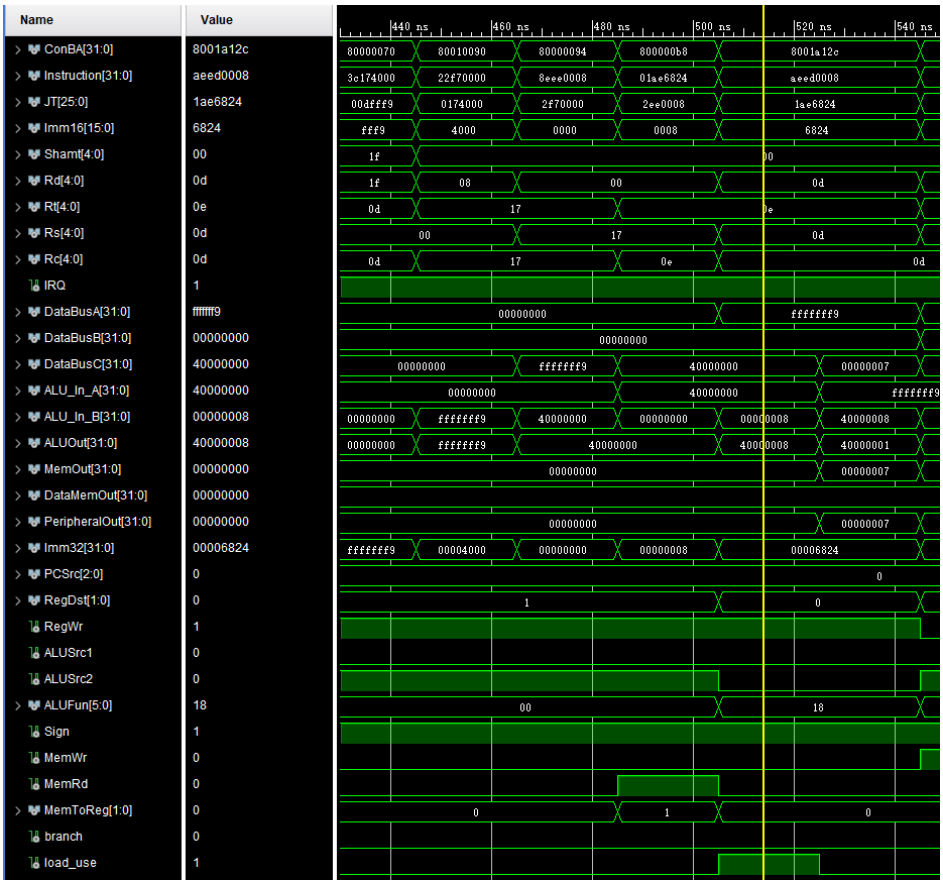


图 13 流水线处理器仿真

以图 13 为例。指令 0xaeed0008（sw $t5, 8($s7)）占据了两个周期，表明下一个指令 0x8eed0014（lw $t5, 20($s7)）被推迟一个周期。很明显能够看出 sw 后紧接着 lw 导致了 load-use 问题，仿真中 load_use 信号正确地提供了一个脉冲，使得 lw 指令被延迟了一个周期。

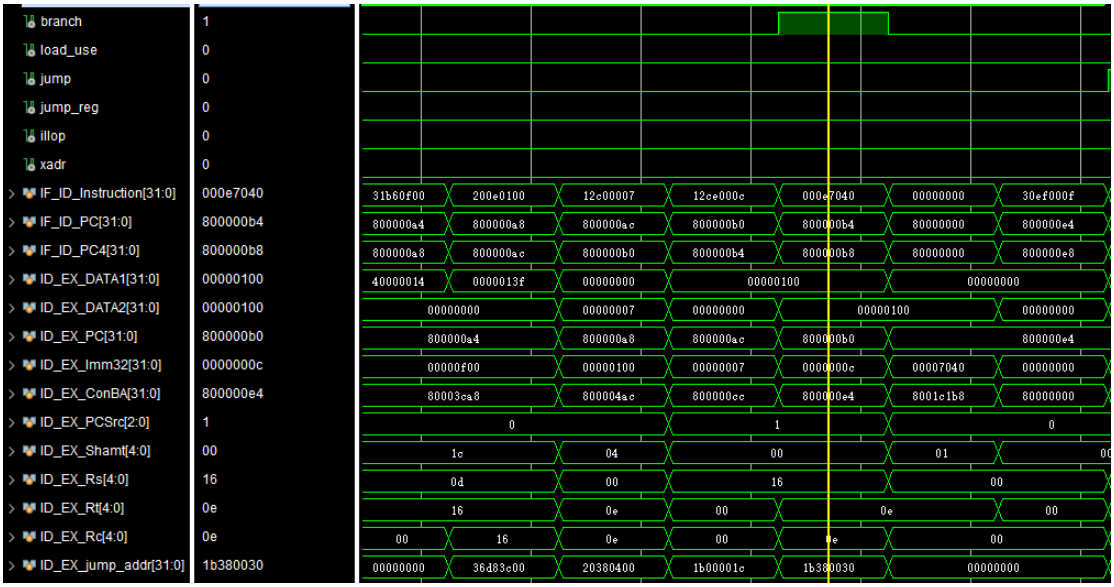


图 14 流水线处理器仿真

以图 14 为例，在判断出分支会发生时，IF、ID 指令被清空。

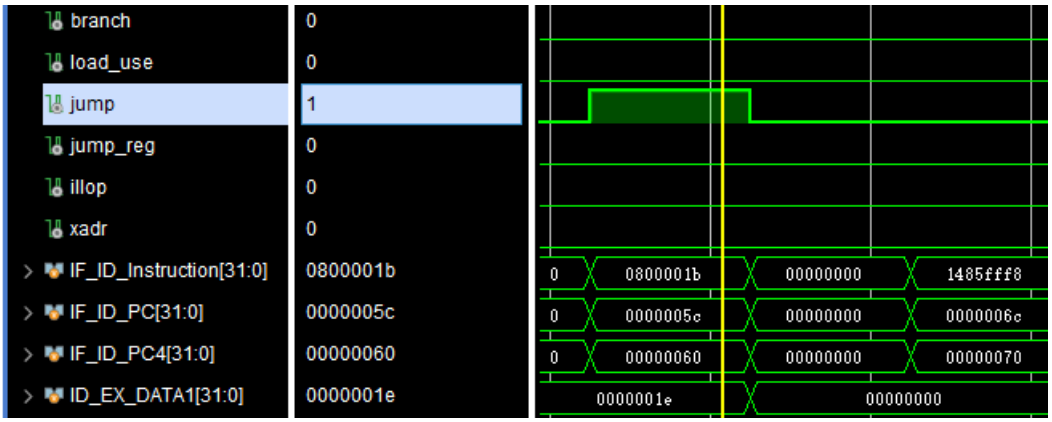


图 15 流水线处理器仿真

以图 15 为例，在 J 型指令的 ID 阶段判断 jump 信号，并将 IF 阶段指令取消。

五、 综合情况

（完成人：应睿）

1. 单周期处理器

a) 时序

当时序约束设置为 create_clock -period 16.800 -name CLK -waveform {0.000 8.400} [get_ports sysclk]

```
# sysclk
set_property -dict {PACKAGE_PIN P17 IOSTANDARD LVCMOS33} [get_ports sysclk]
create_clock -period 16.800 -name CLK -waveform {0.000 8.400} [get_ports sysclk]
#create_clock -period 20.000 -name CLK -waveform {0.000 10.000} [get_ports sysclk]
```

DRC | Methodology | Power | Timing

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	0.064 ns	Worst Hold Slack (WHS):	0.188 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	3567	Total Number of Endpoints:	3567

All user specified timing constraints are met.

图 16 单周期时序报告

如图 16，算得理论最高频率 59.8MHz。

b) 面积

Resource	Utilization	Available	Utilization %
LUT	1670	20800	8.03
LUTRAM	128	9600	1.33
FF	1214	41600	2.92
IO	32	210	15.24

图 17 单周期 LUT 与寄存器使用情况

2. 流水线处理器

a) 时序

当时序约束设置为 create_clock -period 8.280 -name CLK -waveform {0.000 4.140} [get_ports sysclk]，时序报告如图 18。

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.217 ns	Worst Hold Slack (WHS):	0.095 ns	Worst Pulse Width Slack (WPWS):	2.890 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4213	Total Number of Endpoints:	4213	Total Number of Endpoints:	1786

All user specified timing constraints are met.

图 18 流水线时序报告

可以算得理论最高频率 124MHz。

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception
Path 1	0.217	11	11	154	ID_EX_ALUSrc1_reg/C	IF_ID_Instruction_reg[18]/D	7.762	2.982	4.780	8.3	CLK	CLK	
Path 2	0.257	11	11	154	ID_EX_ALUSrc1_reg/C	PC_reg[20]/CE	7.753	2.987	4.766	8.3	CLK	CLK	
Path 3	0.257	11	11	154	ID_EX_ALUSrc1_reg/C	PC_reg[28]/CE	7.753	2.987	4.766	8.3	CLK	CLK	
Path 4	0.264	11	11	154	ID_EX_ALUSrc1_reg/C	PC_reg[10]/CE	7.737	2.987	4.750	8.3	CLK	CLK	

图 19 流水线关键路径

从图 19 可以看到，关键路径主要在 CPU 模块中，为 ID_EX_ALUSrc1 到其他寄存器的相关路径。

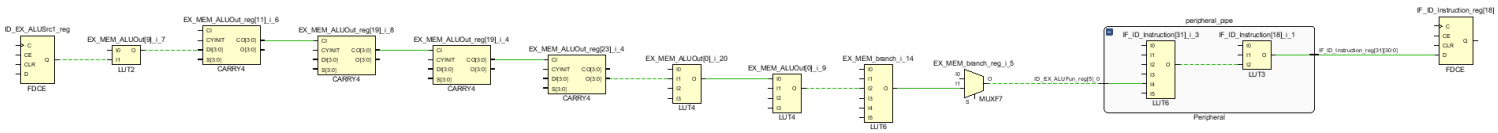


图 20 关键路径图示

从图 20 所示的图表来看，该关键路径从 ID_EX_ALUSrc1 开始，到 ALUOut 的产生，再穿过外设（被使用）后进入到 IF_ID_Instrction 中，基本上就是 CPU 中完整的数据通路。

b) 面积

Resource	Utilization	Available	Utilization %
LUT	1886	20800	9.07
LUTRAM	128	9600	1.33
FF	1707	41600	4.10
IO	32	210	15.24

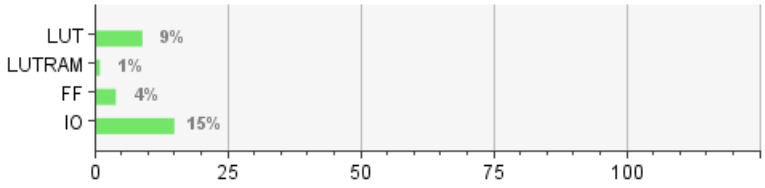


图 21 流水线资源利用率

Name	Slice LUTs (20800)	Slice Registers (41600)
PipelineCPU	1886	1707
datamem (DataMem)	145	0
instruction_memory (R...	95	0
peripheral_pipe (Perip...	439	190
regfile (RegFile)	624	992

图 22 流水线 LUT 与寄存器使用情况

六、 硬件调试情况

（完成人：邓程昊）

我在调试中遇到的问题主要出现在流水线的转发、处理与冒险单元。由于采取的是行为级设计，我并没有开辟出独立的模块来描述这三个单元，而是直接用条件运算符和条件语句来实现。这样做尽管使调试的时候更加方便，但是由于不能与电路很好地对应，变得很容易出错。

在编写好 Verilog 代码后，我开始在 Modelsim 上对流水线进行仿真。结果在每次结束中断时都会发生错误，程序不能返回到之前发生中断的位置。仔细排查过后才发现是选择写回寄存器堆的多路选择器的输入出现了问题。有些关于跳转的 PC 更新并没有正确地输入到这个寄存器中。修改好了这个错误，我又发现在扫描数码管时，其余三位的时间间隔都很正常，但是第三位的持续时间非常短，排查过后发现是因为没有正确地处理好中断信号与其他信号之间的关系。

行为级设计的确为设计者提供了很多方便，将更多底层的优化工作交给了计算机去完成，但同时也一定要对电路的结构有清晰的认识，这样才能够更加顺利地完成设计。

（完成人：徐泽来）

1. 在仿真过程时发现 CPU 无法正确执行 j，jal 等跳转指令。

   解决方案：经逐周期检查 PC 相关信号后发现，这是由于跳转地址计算错误。受实验指导书中参考设计的影响，在最初的单周期 CPU 设计中，我们直接将 Instruct[25:0]即 JT 作为跳转地址赋给了 PC_next，而事实上应将{PC_plus_4[31:28], JT, 2'b00}作为跳转地址赋给 PC_next。经此修改后单周期 CPU 得以正常实现 j, jal 等跳转指令。

2. 在仿真过程中发现中断结束后返回主程序无法正常执行。

   解决方案：重点考察中断前和中断后的若干周期，检查 ra 寄存器以及相关信号后发现，触发中断时 PC 对应的指令并没有正常执行完成，而在最初的单周期设计中，发生中断后是将 PC+4 作为返回地址放入 ra 寄存器中，从而导致在中断结束返回时跳过了中断触发时的一条指令，从而导致此后主程序无法正常运行。通过修改 Control 模块中的 MemToReg 信号，将中断发生时存入 ra 的返回地址改为 PC，从而保证中断结束后能够正常返回主程序中的对应位置，保证后续程序的正常运行。

3. 在烧入 FPGA 后发现数码管存在残影，即原本不应发光的一些段发出了微弱的光。

   解决方案：最初发现这一问题时，我们怀疑这是由于 MIPS 程序中没有正常实现数码管的扫描显示导致，故选择将数码管的扫描频率降低以便于观察；而在将数码管扫描频率降至 1Hz 后却发现原本预期出现的扫描问题并没有出现；此后我们逐渐提高扫描频率，发现在扫描频率较大（定时器的 TH 参数设置为小于-50000）时，数码管会出现残影的情况；经实验观察，在 TH 参数被设置为-30000 时，数码管显示数值既没有残影，由足以保证显示的稳定，故最终将其作为我们数码管的扫描频率。

（完成人：应睿）

在 UART 与 CPU 联调的初期，电脑收到的数据总是 4 位十六进制数，而按道理应该是 2 位十六进制数。此时仿真也未看出问题所在。后来通过分析 UART 发送的十六进制对应的二进制，发现存在重复的比特，怀疑是每个比特接收（或发送）了两次。最后发现是 CPU 时钟频率设为 50MHz，而 UART 将其当作 100MHz 进行波特率分频导致。

但是令 UART 对 50MHz 进行分频又带来了新的问题。CPU 接收到的数据不稳定，存在错位的情况，影响结果。该问题在仿真中也无法复现。经后来考虑，起初 UART 使用 100MHz 时钟时是利用 16 倍波特率作为采样时钟，如今换为 50MHz，由于无法整除，精度下降，采样出错率不再是原来的 5%，导致 UART 接收和发送均出现严重误码。将原生 100MHz 时钟作为 UART 时钟传入后，UART 再次恢复了原有的精度，可以正常工作。

在调试外设的过程中，仿真结果正确，在 CPU 接收到数据并读取后，相应的指示信号（rx_flag）成功清零，且代码通过了综合、实现和烧写，但是在实际硬件上无法正常工作。经过 LED 输出相关信号进行调试，发现 rx_flag 未成功归零，再次参看代码，怀疑是在两个代码块对其赋值导致的多驱动问题，或硬件实现中的信号矛盾问题。通过增加一个 count 中间量，来实现 rx_flag 的正常改变，解决了不能归零的问题，从而使得程序不会卡死在轮询阶段。

七、 思想体会

（完成人：邓程昊）

这次实验是上大学以来任务量比较重的一次，我和团队中的其他两位同学经过了近一周半的努力，终于完成了任务。ALU、单周期和流水线的设计回顾了课上的内容，外设的引入则对课上并没有详细展开的中断和异常有了更深的讨论，除此之外，无论是仿真还是在开放板上真实的运行，这些都锻炼了调试的能力。如何去设计测试方案，如何从测试中找到问题所在，这些都是需要去思考的。测试和开发是同样重要的。

最后还要感谢我的两位队友，不仅是因为他们出色地完成了自己的任务，更重要的是我在和他们交流与合作的过程中也学到了很多东西，例如徐泽来同学告诉我如何更高效地使用 Modelsim 进行仿真，应睿同学告诉我如何利用 GitHub 进行团队开发，这些对我来说都是非常宝贵的经验。团队合作的确也是一种学习的方式。

（完成人：徐泽来）

本次实验是我所完成的 5 次数字逻辑与处理器基础实验中设计最复杂，调试难度最大的一次实验，同时也是让我收获最大的一次实验，体会较深的有以下几点：

1. 先完善设计，后代码实现。

   在过去的几次实验设计中，由于模块数量较少，设计相对简单，故在完成实验时往往是一遍设计电路一遍编写代码实现；而本次 CPU 的模块较多，结构相对复杂，因此如果在设计考虑不够完善的情况下就开始编写代码，那么在后期进行修改完善的时候往往就会出现牵一发而动全身的情况，导致工作量大大增加。在吸取此前的经验教训后，我在编写代码前先完整画出了 CPU 的电路结构，并仔细分析了各指令在 CPU 中的执行情况，因此在编写代码时就只需对着电路图用 Verilog 将其描述出来，且调试遇到的重大问题较少，修改起来相对容易。

2. 极大锻炼了仿真调试的能力。

   在此前的实验设计中，我最多只遇到不到 10 个信号的仿真分析；而在本次实验中，我第一次遇到了数十个，甚至上百个信号的仿真分析工作。在实际调试的过程中，我采取了逐级分部分调试的方法：首先是用一个不含中断和 UART 的求最大公约数程序对 CPU 进行调试，熟悉关键信号变化的同时解决了部分指令实现出现的问题；然后是用一个包含定时器中断但不含 UART 的求最大公约数程序对 CPU 进行调试，重点解决了中断发生前后相关信号的存储处理问题；最后是用完整的，包含中断和 UART 的求最大公约数程序对 CPU 进行调试，重点解决了 UART 接收和发送时信号的处理问题。通过这种逐级分部分的调试方法，我得以将重点关注的问题集中在某一特定部分，并防止了调试难度的陡然增加，让我收获很大。

3. 锻炼了在 FPGA 板上调试的能力。

   由于此前的实验设计都相对简单，故我从来没有使用过 FPGA 上的相关硬件资源来进行片上的调试；而在此次实验过程中，我第一次使用了 FPGA 上闲置的 LED 灯来显示 FPGA 的内部信号，逐步定位问题所在，提高调试效率的同时提升了板上调试的能力。

（完成人：应睿）

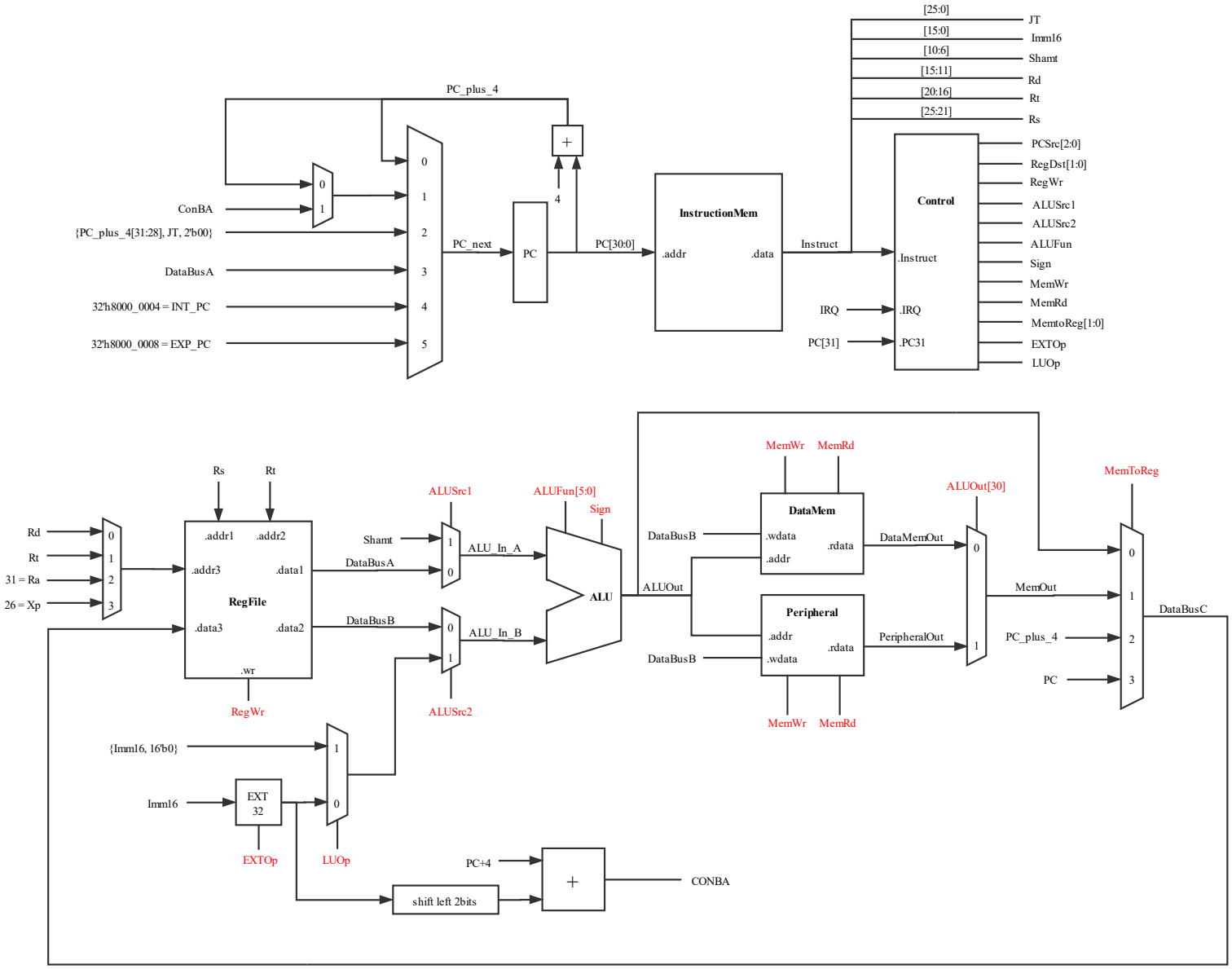
在本次实验中，我负责的 UART 比较关键，是 CPU 最终效果展现的保障。尽管 UART 的实现早已在之前实验中解决，但是将其与 CPU 所需的控制信号结合在一起，还是产生了许多问题。尤其是出现了许多仿真无法发现的问题，着实让我苦恼了好久。但在队友的帮助下，我们一起解决了这些玄而又玄的 bug。队友们对于问题的关键定位总能让我五体投地。

在这个过程中，我学会了看 Vivado 输出来寻找问题，也能熟练地通过 FPGA 的其他外设输出来确定硬件实际运行情况。在与 CPU 部分联调的过程中，通过阅读队友的代码，建

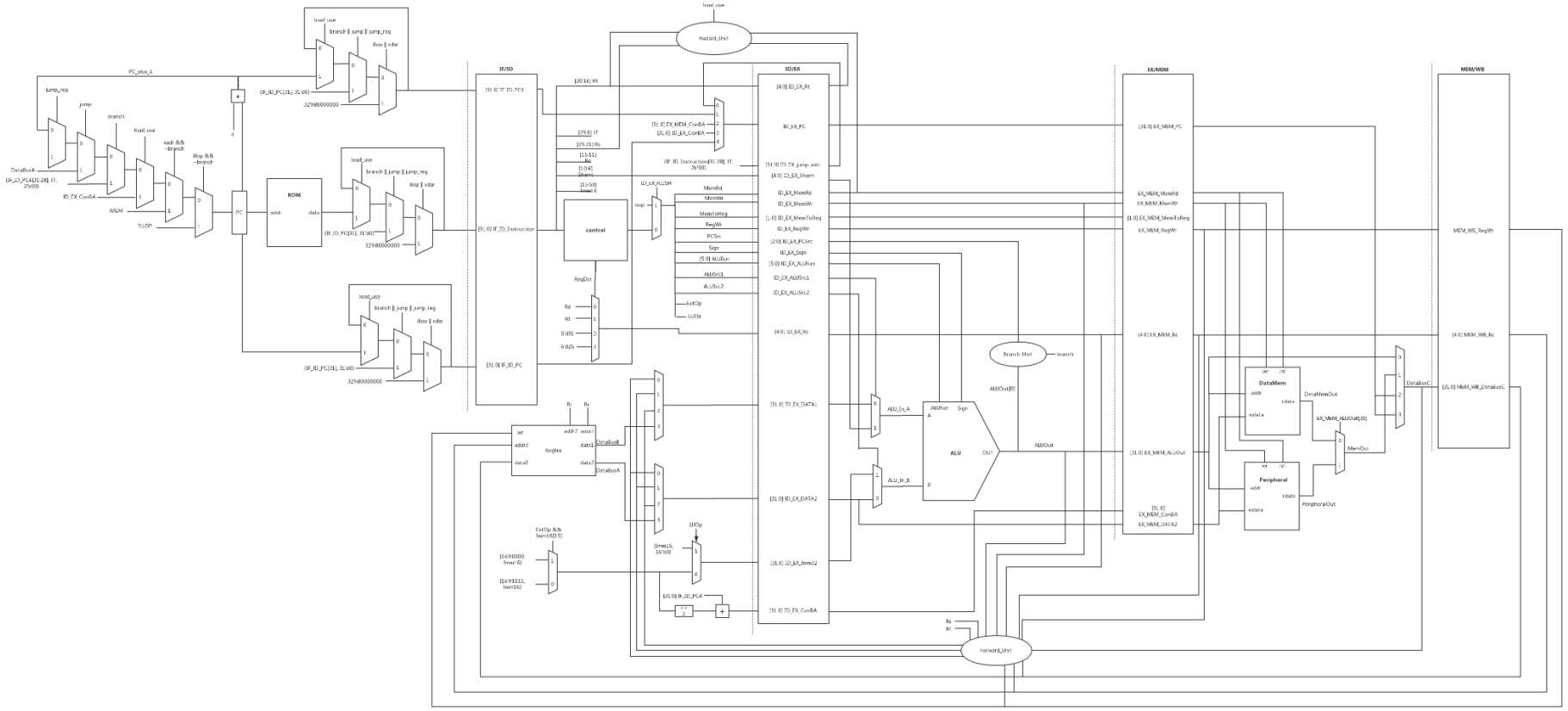
立了 CPU 行为级设计的框架，让我对当时在理论课上始终无法理解到底如何实现的许多设计有了更清晰的认识。

八、 附录

1. 单周期处理器设计图



2. 流水线处理器设计图


IF/ID
ID/EX
EX/MEM
MEM/WB
Hazard_Unit
Forward_Unit
Branch_Unit
ALU
ROM
control
RegFile
DataMem
Peripheral
load_use
jump_reg
jump
branch
jalr
PC_plus_4
DataBusA
ALUOut
MemOut
DataMemOut
PeripheralOut
ALUOut[8]
branch