# 数字逻辑与处理器实验夏季学期

聂浩 王璞瑞 魏齐辉

2013011280 2013011274 2013011278

无 31 无 31 无 31

2015.8.31~9.6

实验名称：32 位 MIPS 处理器设计

实验目的：熟悉现代处理器的基本工作原理；
掌握单周期和流水线处理器的设计方法。

实验原理：参考教材第五章和第六章相关内容。

实验内容：

对电路进行优化通常有三种途径，分别是：

A. 对电路进行优化，达到最大性能（最小延时）

B. 对电路进行优化，达到最小成本（最小面积）

C. 对电路进行优化，达到最佳的性价比（最小的面积延时积）

# 一、 实验分工情况：

| 聂浩 | 王璞瑞 | 魏齐辉 |
|---|---|---|
| ALU、控制模块、单周期、串口、mips 汇编代码、流水线顶层与指令寄存器 | 汇编器、PC、流水线冒险单元 | 外设调试、流水线转发单元 |

# 二、 ALU 的设计与优化—聂浩

## 实验要求：

设计一个 32 位 ALU，实现基本的算术、逻辑、关系、位与移位运算，尽量优化以达到最小的面积延时积。ALU 功能表如下所示：

| 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 |
| | GEZ | 111001 | If(A>=0) S=1 else S=0 |
| | GTZ | 111111 | If(A>0) S=1 else S=0 |

ALU 端口如下表所示：

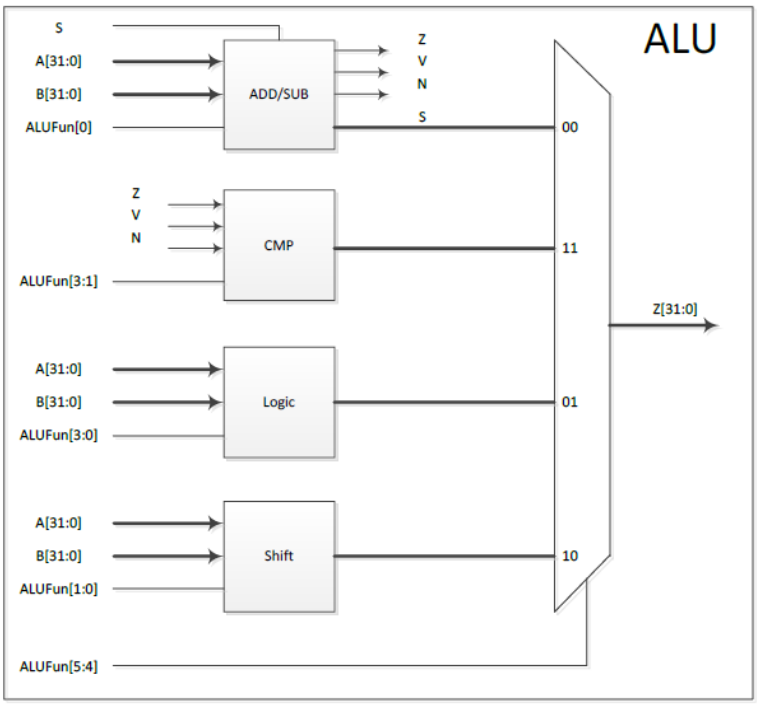
| 名称 | 类型 | 描述 |
| --- | --- | --- |
| A[31:0] | 输入 | 操作数 1 |
| B[31:0] | 输入 | 操作数 2 |
| ALUFun[5:0] | 输入 | ALU 功能 |
| Sign | 输入 | 运算符号，1：有符号；0：无符号 |
| Z[31:0] | 输出 | 结果输出 |

A. 加法运算实现可以采用逐次进位、超前进位等结构，减法可以通过加法实现（参见见面理论课讲义或者前面实验）；同时输出 Z（结果为零）、V（结果溢出）、N（结果为负）等标志位，注意有符号数和无符号数标志产生的不同。

B. 比较运算根据减法运算的结果（Z/V/N）产生，自行分析比较操作与算术运算之间的关系。

C. 移位运算可以考虑将移位操作拆分为 16 位移位、8 位移位、4 位移位、2 位移位、1 位移位等几个子运算的组合，然后级联形成最后的运算结果。

D. 逻辑运算可以根据要求直接产生。




## 设计思路及代码：

在三个文件夹里的 alu 是一致的；按照上图所示的模块，分成五个模块：

1) 顶层模块，代码如 alu.v,

这里简单的进行了连线，同时完成了上图的选择器

```
//By Neil 2015/8/21
module ALU(A,B,Sign,ALUFun,result);
input [31:0]A,B;
input [5:0]ALUFun;
input Sign;
output reg [31:0]result;
wire zero,out_of_range,no_zero,cmp;
wire [31:0]B_tmp,addS,shift,lg;

assign B_tmp=(ALUFun[3]==1'b1)?32'b0:B;

ALUadd
add(.A(A),.B(B_tmp),.Sign(Sign),.ALUFun(ALUFun),.zero(zero),.out_of_range(out_of_range),.no_z
ero(no_zero),.addS(addS));
ALUcmp compare(.Z(zero),.V(out_of_range),.N(no_zero),.ALUFun(ALUFun[3:1]),.cmp(cmp));
ALUlg lgc(.A(A),.B(B),.ALUFun(ALUFun[3:0]),.lg(lg));
ALUshift shi(.A(A[4:0]),.B(B),.ALUFun(ALUFun[1:0]),.shift(shift));


always@(*) begin
    case(ALUFun[5:4])
        2'b00: result    <=  addS;
        2'b11: result    <=  {31'b0,cmp};
        2'b01: result    <=  lg;
        2'b10: result    <=  shift;
        default: result  <=  0;
    endcase
end
endmodule
```

2) 算术运算模块（alu_add_sub.v）

module ALUadd(A,B,Sign,ALUFun,zero,out_of_range,no_zero,addS)

这里使用了八个四位超前进位加法器级联。通过判断 ALUFun 来对 B_tmp 进行赋值从而实现加减法。其中 sign 用于决定是有符号运算还是无符号运算，结果输出到 addS。

代码如下：

```
//by neil 2015/8/21
//ALU--加法器单元
module ALUadd(A,B,Sign,ALUFun,zero,out_of_range,no_zero,addS);
```

```
    input [31:0]A,B;
    input Sign;
    input [5:0]ALUFun;
    output zero,out_of_range,no_zero;
    output [31:0]addS;
    wire [31:0]B_tmp;
    wire [7:0]ctmp;
    wire ctmp_31;
    assign B_tmp=(ALUFun==5'b0)?B:((B^{32'hffffffff})+1);

    adder4
a1(.A(A[3 :0 ]),.B(B_tmp[3 :0 ]),.cout(ctmp[0]),.cin(1'b0)      ,.addS(addS[3 :0 ]),.cout_3());
    adder4
a2(.A(A[7 :4 ]),.B(B_tmp[7 :4 ]),.cout(ctmp[1]),.cin(ctmp[0]),.addS(addS[7 :4 ]),.cout_3());
    adder4
a3(.A(A[11:8 ]),.B(B_tmp[11:8 ]),.cout(ctmp[2]),.cin(ctmp[1]),.addS(addS[11:8 ]),.cout_3());
    adder4
a4(.A(A[15:12]),.B(B_tmp[15:12]),.cout(ctmp[3]),.cin(ctmp[2]),.addS(addS[15:12]),.cout_3());
    adder4
a5(.A(A[19:16]),.B(B_tmp[19:16]),.cout(ctmp[4]),.cin(ctmp[3]),.addS(addS[19:16]),.cout_3());
    adder4
a6(.A(A[23:20]),.B(B_tmp[23:20]),.cout(ctmp[5]),.cin(ctmp[4]),.addS(addS[23:20]),.cout_3());
    adder4
a7(.A(A[27:24]),.B(B_tmp[27:24]),.cout(ctmp[6]),.cin(ctmp[5]),.addS(addS[27:24]),.cout_3());
    adder4
a8(.A(A[31:28]),.B(B_tmp[31:28]),.cout(ctmp[7]),.cin(ctmp[6]),.addS(addS[31:28]),.cout_3(ctmp_
31));
    assign zero=(addS==0);
    assign
out_of_range=(Sign&(ctmp[7]==ctmp_31))|((~Sign)&((ctmp[7]&ALUFun)|(~ctmp[7]&~ALUFun)));
    assign no_zero=(Sign&addS[31]);
    endmodule

    //四位加法器
    module adder4(A,B,cout,cout_3,cin,addS);
    input [3:0]A,B;
    input cin;
    reg [2:0]ctmp;
    output cout,cout_3;
    output [3:0]addS;
    wire [3:0]p,g;

    assign      p[0]=A[0]^B[0];
    assign      p[1]=A[1]^B[1];
```

```
    assign      p[2]=A[2]^B[2];
    assign      p[3]=A[3]^B[3];
    assign      g[0]=A[0]&B[0];
    assign      g[1]=A[1]&B[1];
    assign      g[2]=A[2]&B[2];
    assign      g[3]=A[3]&B[3];
    assign
cout=g[3]|(g[2]&p[3])|(g[1]&p[2]&p[3])|(p[3]&p[2]&p[1]&g[0])|(p[0]&p[1]&p[2]&p[3]&cin);
    assign      cout_3=g[2]|(g[1]&p[2])|(p[2]&p[1]&g[0])|(p[0]&p[1]&p[2]&cin);
    assign      addS[0]=p[0]^cin;
    assign      addS[1]=p[1]^(g[0]|(p[0]&cin));
    assign      addS[2]=p[2]^(g[1]|(p[1]&g[0])|(p[1]&p[0]&cin));
    assign      addS[3]=p[3]^(g[2]|(g[1]&p[2])|(p[2]&p[1]&g[0])|(p[0]&p[1]&p[2]&cin));
    endmodule
```

3) 比较运算模块（alu_cmp.v）module ALUcmp(Z,V,N,ALUFun,cmp)

利用 add_sub 模块产生的信号得出结果

```
//By Neil 2015/8/21
module ALUcmp(Z,V,N,ALUFun,cmp);
input Z,V,N;
input [2:0]ALUFun;
output reg cmp;
always @(*) begin
case (ALUFun)
    3'b001:cmp   <=   Z;
    3'b000:cmp   <=   ~Z;
    3'b010:cmp   <=   N;
    3'b110:cmp   <=   Z|N;
    3'b100:cmp   <=   Z|~N;
    3'b111:cmp   <=   ~(Z|N);
    default:cmp <=   1'b0;
endcase
end
endmodule
```

4) 逻辑运算模块(alu_logic.v) module ALUlg(A,B,ALUFun,lg)

直接使用 verilog 的逻辑运算符完成

```
//By Neil 2015/8/21
module ALUlg(A,B,ALUFun,lg);
input [31:0]A,B;
input [3:0]ALUFun;
output reg [31:0]lg;
```

```
always @(*) begin
    case (ALUFun)
        4'b1000:lg    <=   A&B;
        4'b1110:lg    <=   A|B;
        4'b0110:lg    <=   A^B;
        4'b0001:lg    <=   ~(A|B);
        4'b1010:lg    <=   A;
        default:lg    <=   A;
    endcase
end
endmodule
```

### 5) 移位运算模块（alu_shift.v）module ALUshift(A,B,ALUFun,shift)

移位运算可以理解为 1 位，2 位，4 位，8 位，16 位移位的组合，这样通过多级级联和选择器的方式可以得到三种（sra，sll，srl）指令的实现。

代码如下

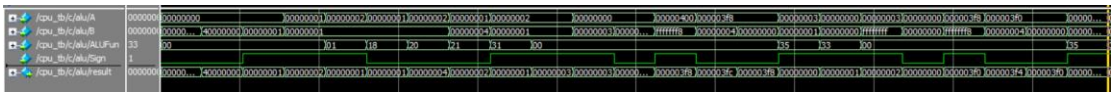
```
//By Neil 2015/8/21
module ALUshift(A,B,ALUFun,shift);
input [4:0]A;
input [31:0]B;
input [1:0]ALUFun;
output [31:0]shift;
wire [31:0] l1,l2,l4,l8,l16,r1,r2,r4,r8,r16;
assign l1    =    (A[0])?{B [30:0],1'b0 }:B ;
assign l2    =    (A[1])?{l1[29:0],2'b0 }:l1;
assign l4    =    (A[2])?{l2[27:0],4'b0 }:l2;
assign l8    =    (A[3])?{l4[23:0],8'b0 }:l4;
assign l16   =    (A[4])?{l8[15:0],16'b0}:l8;

assign r1    =    (A[0])?{((ALUFun[1]==1'b1)&&(B[31]==1))?1'b1 :1'b0 ,B [31:1] }:B ;
assign r2    =    (A[1])?{((ALUFun[1]==1'b1)&&(B[31]==1))?2'b1 :2'b0 ,r1[31:2] }:r1;
assign r4    =    (A[2])?{((ALUFun[1]==1'b1)&&(B[31]==1))?4'b1 :4'b0 ,r2[31:4] }:r2;
assign r8    =    (A[3])?{((ALUFun[1]==1'b1)&&(B[31]==1))?8'b1 :8'b0 ,r4[31:8] }:r4;
assign r16   =    (A[4])?{((ALUFun[1]==1'b1)&&(B[31]==1))?16'b1:16'b0,r8[31:16]}:r8;

assign shift = (ALUFun[0])?r16:l16;

endmodule
```

## 仿真结果



对照 ALUFun 参数的取值与题中所给表格可知输出（result）是正确的。

# 三、 单周期处理器

## 实验要求

A. 本实验完成一个 32 位的单周期 MIPS 架构处理器，实现核心 MIPS 指令集系统的一个子集，如下：

*i* 空指令：nop (0x00000000,即 sll $0,$0,0)

*ii* 存储访问指令：lw、sw、lui；

*iii* 算术逻辑指令：add、addu、sub、subu、addi、addiu、and、or、xor、nor、andi、sll、srl、sra、slt、slti、sltiu；

*iv* 分支和跳转指令 branch(beq、bne、blez、bgtz、bgez)和 jump(j、jal、jr、 jalr)；

*v* 其他指令可以根据情况自行添加。

该处理器支持异常（为简单起见，可以只支持未定义指令异常）和中断（定时器中断）的处理。

Register File 模块提供 32 路 32bits 的寄存器资源，注意：其中$0 的取值永远为 0，见附件。

指令存储器地址空间和数据存储器地址空间是分离的，指令存储器实现采用常量数组（ROM）的方式，根据自己设计的程序设定合理的大小，样例见附件。

数据存储的地址空间被划分为 2 部分：0x00000000～0x3FFFFFFF（字节地址）为数据 RAM，可以提供数据存储功能；0x40000000～0x7FFFFFFF（字节地址）为外设地址空间，对其地址的读写对应到相应的外设资源（LEDs、SWITCH…），见附件。

具体地址划分如下：

| 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<br>1bit：LED 1<br>.....<br>7bit：LED 7 |

| | | |
|---|---|---|
| 0x40000010 | 外部 SWITCH | 0bit：Switch 0<br>1bit：Switch1<br>.....<br>7bit：Switch7 |
| 0x40000014 | 七段数码管 | 0bit：CA<br>1bit：CB<br>.....<br>7bit：DP<br>8bit：AN0<br>9bit：AN1<br>10bit：AN2<br>11bit：AN3 |
| 0x40000018~0x40000023 | UART | UART 外设地址 |

注：对于 Altera DE2 平台，需额外使用示例代码 digitube_scan.v 将硬件上的非扫描模式的七段数码管转换为扫描模式。

提供一个定时器外设，可以根据设定周期产生外部中断，通过该定时器触发 7 段数码管的扫描显示。

| 地址范围 | 功能 | 备注 |
|---|---|---|
| 0x40000000 | 定时器 TH | 每当 TL 计数到全 1 时，自动加载 TH 值到 TL |
| 0x40000004 | 定时器 TL | 定时器计数器，TL 值随时钟递增 |
| 0x40000008 | 定时器控制 TCON | 0bit：定时器使能控制，1-enable，0disable<br>1bit：定时器中断控制，1-enable，0disable<br>2bit：定时器中断状态 |

定时器软件操作流程：

i. 关闭定时器，TCON 写入 0;

ii. 设置定时器周期，TH 取值决定定时器的计数周期;

iii. 设置定时器 TL 为 0xFFFFFFFF;

iv. 启动定时器，TCON 写入

定时器中断软件服务程序流程(此时处理器处于内核态,监督位为'1')：

i. 定时器中断禁止，同时中断状态清零，TCON 的 1-2bit 清零， TCON &=0xfffffff9;

ii. 保护现场;

iii. 中断处理代码;

iv. iv. 恢复现场;

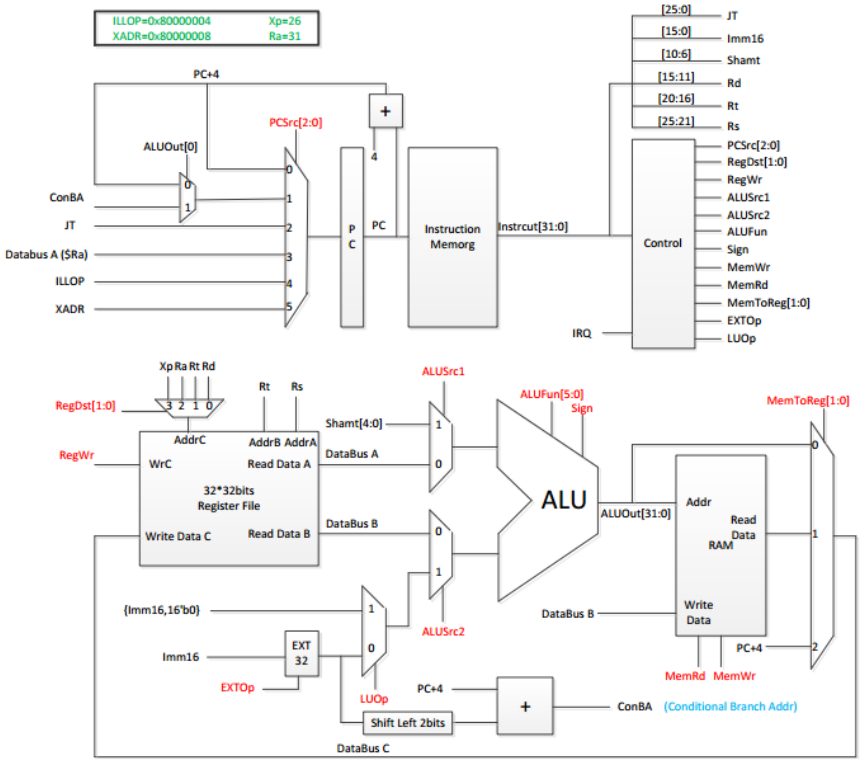
使能中断，TCON 的 1bit 置 1，TCON |= 0x00000002;

退出中断服务程序，跳转到中断发生时保存的断点地址处继续执行（$26）.

根据春季学期设计的 UART 实现一个 MIPS 的 UART 外设，波特率固定在 9600，起始位 1 位，停止位 1 位，无校验位。访问方式可以采用中断方式也可以采用轮询方式，建议采用轮询方式会比较简单。寄存器根据设计自行决定，参考实现的寄存器映射如下：

| 地址范围 | 功能 | 备注 |
|---|---|---|
| 0x40000018 | 串口发送数据<br>UART_TXD | 串口发送数据寄存器，只有低 8bit 有效；对该地址的写操作将触发新的 UART 发送 |
| 0x4000001C | 串口接收数据<br>UART_RXD | 串口接收数据寄存器，只有低 8bit 有效 |
| 0x40000020 | 串口状态、控制<br>UART_CON | 0bit：发送中断使能，1-enable，0disable<br>1bit：接收中断使能，1-enable，0disable<br>2bit：发送（中断）状态，每当 UART_TXD 中的数据发送完毕后该比特置‘1’，当执行对该地址的读操作后，将自动清零<br>3bit：接收（中断）状态，每当 UART_RXD 中已经接收到一个完整的字节时该比特置‘1’，当执行对该地址的读操作后，将自动清零 |
| | | 4bit：发送模块状态，0-发送模块处于空闲状态，1-发送模块处于发送状态 |

单周期处理器可以参照下面的结构实现，也可以根据讲义或者扩充后的指令集自行设计实现。

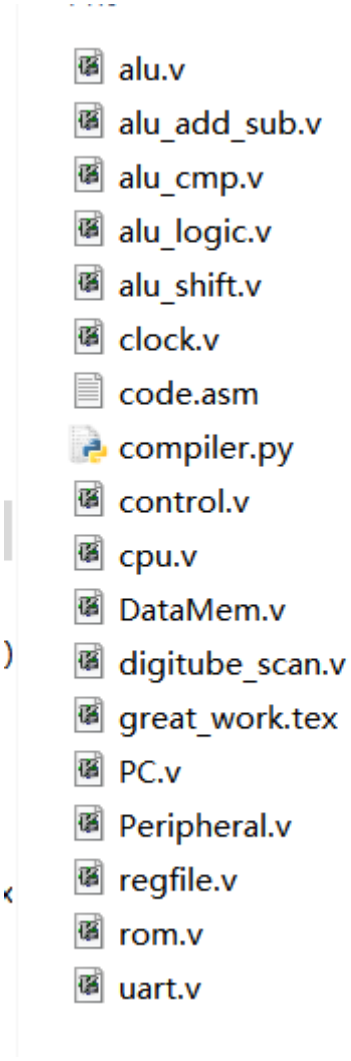


# 设计思路及代码

单周期处理器需要再单独设计 PC 模块，分频模块，控制信号模块和之前的 ALU 模块，再加上 Rom，RegFile，外设模块等已知模块，组装即可。

需要注意的是，中断跳转地址和异常跳转指令之间只有 1 个指令的空间，虽然中断的实现是通过中断跳转（即 80000004 是跳转）到指定位置去，但是一条指令还是不太够，所以把 XADR 改成了 80000020。这部分的代码放在 single_soft 文件夹中。

汇编代码见 code.asm 文件。文件列表如下




## 1) 顶层模块—聂浩

外设、串口和 ram 以类似于总线的方式在顶层模块中并联在一起，在三个模块内部进行地址判断，然后用选择器选择三者出来的信号。其他的连接和实验要求中的图是一样的。

代码详见 cpu.v

```
//by Neil 2015/9/2
//CPU
```

```
module
cpu(sysclk,reset,switch,digi_out1,digi_out2,digi_out3,digi_out4,led,U
ART_RXD,UART_TXD);
input sysclk,reset;
input [7:0]switch;
output [6:0]digi_out1,digi_out2,digi_out3,digi_out4;
output [7:0]led;
wire clk;
wire debug;
wire [7:0]digi;
wire [25:0]JT;
wire [31:0]ConBA,Pc;
wire [31:0]instruct;
wire irq;
wire [15:0]Imm16;
wire [4:0]Shamt;
wire [4:0]Rd,Rt,Rs;
wire [5:0]Opcode;
wire [5:0]Funct;

wire [2:0] PCSrc;
wire [1:0] RegDst;
wire RegWrite;
wire ALUSrc1,ALUSrc2;
wire [5:0] ALUFun;
wire Sign;
wire MemWrite,MemRead;
wire [1:0] MemtoReg;
wire ExtOp;
wire LuOp;


wire [4:0]Addrc;
wire [31:0]write_Data_C;
wire [31:0]AluA,AluB;
wire [31:0]DataBusA,DataBusB,DataBusC;
wire [31:0]ALUOut;
wire [31:0]pout,dmout;
wire [31:0]lu,Ext_out,readdataout;

input UART_RXD;
output UART_TXD;
wire [31:0]uout;
wire sig,sig16;
```

```
//assign led[0]=UART_TXD;
assign  JT=instruct [25:0];
cloc ck(sysclk,clk,sig,sig16);
PC pc(clk,reset,ALUOut[0],ConBA,DataBusA,JT,PCSrc,Pc);

ROM rom(Pc,instruct);
assign  Opcode  =instruct[31:26];
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];
assign  Funct   =instruct[5:0];
Control control(Opcode,  Funct, irq , reset,
             PCSrc,
             RegDst,  RegWrite,
             ALUSrc1, ALUSrc2,    ALUFun,
             Sign,
             MemRead, MemWrite,   MemtoReg,
             ExtOp, LuOp);

assign Addrc=(RegDst==2'b0)?Rd:
             (RegDst==2'b01)?Rt:
             (RegDst==2'b10)?5'd31:
             5'd26;

RegFile
rf(reset,clk,Rs,DataBusA,Rt,DataBusB,RegWrite,Addrc,DataBusC);

assign
Ext_out=(Imm16[15]==1&&ExtOp==1)?{16'b1111_1111_1111_1111,Imm16}:{16'
b0000_0000_0000_0000,Imm16};
assign ConBA=(Ext_out<<2)+Pc+4;



assign lu=(LuOp==1'b1)?{Imm16,16'b0}:Ext_out;

assign AluA=(ALUSrc1==1'b1)?Shamt:DataBusA;
assign AluB=(ALUSrc2==1'b1)?lu:DataBusB;

ALU alu(AluA,AluB,Sign,ALUFun,ALUOut);
```

```
Peripheral
per(reset,clk,MemRead,MemWrite,ALUOut,DataBusB,pout,led,switch,digi,i
rq);
digitube_scan ds(digi,digi_out1,digi_out2,digi_out3,digi_out4);

DataMem dm(reset,clk,MemRead,MemWrite,ALUOut,DataBusB,dmout);

UART
ua(clk,reset,ALUOut,uout,MemRead,DataBusB,MemWrite,UART_RXD,UART_TXD,
sig,sig16);

assign
readdataout=(ALUOut<32'h40000000)?dmout:(ALUOut>=32'h40000018)?uout:p
out;

assign
DataBusC=(irq==1&&(Opcode==6'h02||Opcode==6'h03))?{4'h0,JT,2'b00}:(Me
mtoReg==2'b0)?ALUOut[31:0]:
              (MemtoReg==2'b1)?readdataout:
              Pc+4;
endmodule
```

2) PC –王璞瑞

根据 PCSrc 和 ALUOut0 的值的不同可以将 PC 的值分为以下 7 种情况。

| PCSrc | 000 | 001 | 001 | 010 | 011 | 100 | 101 |
|---|---|---|---|---|---|---|---|
| ALUOut0 | x | 0 | 1 | x | x | x | x |
| PC | PC+4 | PC+4 | ConBA | T | DatabusA | ILLOP | XADR |

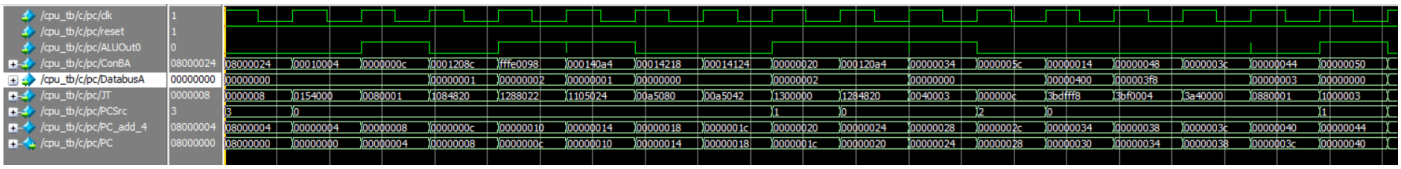
然后利用 case 语句进行分类赋值，具体函数见下：

```
//by Pray Wolf
//2015.9.2
module PC(clk,reset,ALUOut0,ConBA,DatabusA,T,PCSrc,PC);
  input clk,reset;
  input ALUOut0;
  input[31:0] ConBA,DatabusA;
  input[25:0] JT;
  input[2:0] PCSrc;
  wire [31:0]PC_add_4;
  output reg[31:0] PC;
  parameter ILLOP=32'h8000_0004;
  parameter XADR=32'h8000_0008;
initial
```

```
    PC=32'b0;
      assign PC_add_4=PC+32'd4;//为了保证 PC 最高位不变

      always @(posedge clk,negedge reset)
       begin
         if(reset==0)//reset????
           PC<=32'h8000000;
         else
          begin
            case(PCSrc)
              3'b000:PC<={PC[31],PC_add_4[30:0]};
              3'b001:PC<=(ALUOut0==1)?
{PC[31],ConBA[30:0]}:{PC[31],PC_add_4[30:0]};
              3'b010:PC<={PC[31:28],JT,2'b00};//,al 不改变最高位
              3'b011:PC<=DatabusA;
              3'b100:PC<=ILLOP;
              3'b101:PC<=XADR;
                  default:PC<=32'h8000000;
            endcase
          end
       end
    endmodule
```

仿真后得到的图如下:



当 PCSrc 为 011 时，下一周期 PC 变为 DatabusA 的值，当 PCSrc=000 时，PC 在原基础上+4，当 PCSrc 为 001 且 ALUOut0 也为 1 时，PC 变为 ConBA，当 PCSrc 为 010 时，PC 是由 JT 拼接而成。由仿真可见，PC 模块符合设计需求。

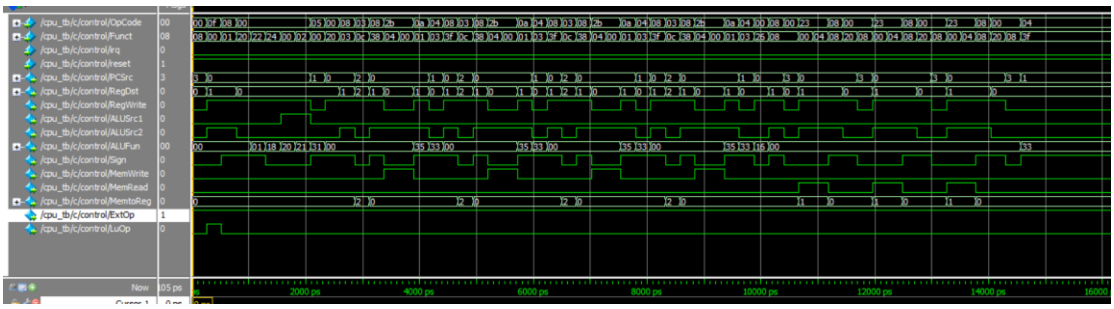
3) 控制模块—聂浩

| | PCSrc[1:0] | RegWrite | RegDst[1:0] | MemRead | MemWrite | MemtoReg[1:0] | ALUSrc1 | ALUSrc2 | ExtOp | LuOp | sign | AluFun |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| lw | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| sw | 0 | 0 | x | 0 | 1 | x | 0 | 1 | 1 | 0 | 0 | 0 |
| lui | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| add | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 1 | 0 |
| addu | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | 0 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| sub | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 1 | 1 |
| subu | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | 1 |
| addi | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| addiu | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| and | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | x | 011000 |
| or | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | x | 011110 |
| xor | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | x | 010110 |
| nor | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | x | 011010 |
| andi | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | x | 011000 |
| sll | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | x | x | x | 100000 |
| srl | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | x | x | x | 100001 |
| sra | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | x | x | x | 100011 |
| slt | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 1 | 110101 |
| sltu | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | 110101 |
| slti | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 110101 |
| sltiu | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 110101 |
| beq | 1 | 0 | x | 0 | 0 | x | 0 | 0 | 1 | x | 1 | 110011 |
| j | 2 | 0 | x | 0 | 0 | x | x | x | x | x | x | x |
| jal | 2 | 1 | 2 | 0 | 0 | 2 | x | x | x | x | x | x |
| jr | 3 | 0 | x | 0 | 0 | x | x | x | x | x | x | x |
| jalr | 3 | 1 | 0 | 0 | 0 | 2 | x | x | x | x | x | x |
| bne | 1 | 0 | x | 0 | 0 | x | 0 | 0 | 1 | x | 1 | 110001 |
| blez | 1 | 0 | x | 0 | 0 | x | 0 | x | 1 | x | 1 | 111101 |
| bgtz | 1 | 0 | x | 0 | 0 | x | 0 | x | 1 | x | 1 | 111111 |
| bgez | 1 | 0 | x | 0 | 0 | x | 0 | x | 1 | x | 1 | 111001 |

控制模块真值表

控制模块的逻辑比较复杂，本身是打算利用课程中讲的逻辑化简法进行完成，但是这给编程和调试带来了很大的麻烦，最终改用了 case 的判断方式，能够很好的完成任务，并使代码有了很好的可读性，仿真如图。



代码见 control.v：

```
    //By neil
//2015.9.2
module Control(OpCode,  Funct, irq , reset,
               PCSrc,
               RegDst,  RegWrite,
```

```
            ALUSrc1, ALUSrc2,   ALUFun,
            Sign,
            MemRead, MemWrite,  MemtoReg,
            ExtOp, LuOp);
input [5:0] OpCode;
input [5:0] Funct;
input irq,reset;//,clk;
output reg [2:0] PCSrc;
output reg [1:0] RegDst;
output reg RegWrite;
output reg ALUSrc1;
output reg ALUSrc2;
output reg [5:0] ALUFun;
output reg Sign;
output reg MemWrite;
output reg MemRead;
output reg [1:0] MemtoReg;
output reg ExtOp;
output reg LuOp;


initial begin
    PCSrc      <=   3'b0;
    RegDst     <=   2'b0;
    RegWrite   <=   1'b0;
    ALUSrc1    <=   1'b0;
    ALUSrc2    <=   1'b0;
    ALUFun     <=   6'b000000;
    Sign       <=   1'b0;
    MemWrite   <=   1'b0;
    MemRead    <=   1'b0;
    MemtoReg   <=   2'b0;
    ExtOp      <=   1'b1;
    LuOp       <=   1'b0;
end


always @(*) begin
     PCSrc      <=   3'b0;
    RegDst     <=   2'b0;
    RegWrite   <=   1'b0;
    ALUSrc1    <=   1'b0;
    ALUSrc2    <=   1'b0;
    ALUFun     <=   6'b000000;
```

```
Sign       <=   1'b0;
MemWrite    <=   1'b0;
MemRead     <=   1'b0;
MemtoReg    <=   2'b0;
ExtOp       <=   1'b1;
LuOp        <=   1'b0;
if(reset==1) begin
    if(irq==1) begin//
              PCSrc<=3'b100;
      RegDst<=2'b11;
      RegWrite<=1'b1;
      MemtoReg<=2'b10;
    end
    else begin
        case(OpCode)
            6'b0: begin//
            RegWrite    <= 1'b1;
                     case( Funct )
                   6'h20:begin//add
                       Sign<= 1'b1;
                   end

                   6'h21:begin//addu
                   end

                   6'h22:begin//sub
                       Sign<= 1'b1;
                       ALUFun<=6'b1;
                   end

                   6'h23:begin//subu
                       ALUFun<=6'b1;
                   end

                   6'h24:begin//and
                       ALUFun<=6'b011000;
                   end

                   6'h25:begin//or
                       ALUFun<=6'b011110;
                   end

                   6'h26:begin//xor
                       ALUFun<=6'b010110;
```

```
        end

        6'h27:begin//nor
            ALUFun<=6'b011010;
        end

        6'h0 :begin//sll
            ALUSrc1<=1'b1;
            ALUFun<=6'b100000;
        end

        6'h02:begin//srl
            ALUSrc1<=1'b1;
            ALUFun<=6'b100001;
        end

        6'h03:begin//sra
            ALUSrc1<=1'b1;
            ALUFun<=6'b100011;
        end

        6'h2a:begin//slt
            Sign<=1'b1;
            ALUFun<=6'b110101;
        end

        6'h2b:begin//sltu
            ALUFun<=6'b110101;
        end

        6'h08: begin//jr
            PCSrc<= 3'b011;
            RegWrite<=1'b0;
        end

        6'h09: begin//jalr
            PCSrc <= 3'b011;
            MemtoReg<=2;
        end
               default:begin
               end
    endcase
end
```

```
6'h23: begin//lw
   RegWrite<=1'b1;
   RegDst<=2'b01;
   MemRead<=1'b1;
   MemtoReg<=2'b01;
   ALUSrc2<=1'b1;
end

6'h2b: begin//sw
   MemWrite<=1'b1;
   ALUSrc2<=1'b1;
end

6'h0f: begin//lui
   RegWrite<=1'b1;
   RegDst<=2'b01;
   ALUSrc2<=1'b1;
   LuOp<=1'b1;
end

6'h08: begin//addi
   RegWrite<=1'b1;
   RegDst<=2'b01;
   ALUSrc2<=1'b1;
   Sign<=1'b1;
end

6'h09: begin//addiu
   RegWrite<=1'b1;
   RegDst<=2'b01;
   ALUSrc2<=1'b1;
end

6'h0c: begin//andi
   RegWrite<=1'b1;
   RegDst<=2'b01;
   ALUSrc2<=1'b1;
   ExtOp<=1'b0;
   ALUFun<=6'b011000;
end

6'h0a: begin//slti
   RegWrite<=1'b1;
   RegDst<=2'b1;
```

```
    ALUSrc2<=1'b1;
    Sign<=1'b1;
    ALUFun<=6'b110101;
end

6'h0b: begin//sltiu
    RegWrite<=1'b1;
    RegDst<=2'b1;
    ALUSrc2<=1'b1;
    ALUFun<=6'b110101;
end

6'h04: begin//beq
    PCSrc<=3'b001;
    Sign<=1'b1;
    ALUFun<=6'b110011;
end

6'h05: begin//bne
    PCSrc<=3'b001;
    Sign<=1'b1;
    ALUFun<=6'b110001;
end

6'h06: begin//blez
    PCSrc<=3'b001;
    Sign<=1'b1;
    ALUFun<=6'b111101;
end

6'h07: begin//bgtz
    PCSrc<=3'b001;
    Sign<=1'b1;
    ALUFun<=6'b111111;
end
6'h01: begin//bgez
    PCSrc<=3'b001;
    Sign<=1'b1;
    ALUFun<=6'b111001;
end

6'h02: begin//j
    PCSrc<=3'b010;
end
```

```
                6'h03: begin//jal
                    RegWrite<=1'b1;
                    PCSrc<=3'b010;
                    MemtoReg<=2'b10;
                    RegDst<=2'b10;
                end

                default: begin
                   PCSrc<=3'b101;
                    RegDst<=2'b11;
                    RegWrite<=1'b1;
                    MemtoReg<=2'b10;
                end
            endcase
        end
    end
 end
endmodule
```

4) 串口—聂浩

串口利用聂浩的串口的发送、控制模块和魏齐辉的接收模块修改而来。这里值得一提的是，由于采用轮询的串口读取方式，如果和之前一样保留接收到的数据，会导致多次轮询读到一样的数据。

因此必须在读到数据后很快的把该数据清零，但是串口的时钟和 cpu 时钟不同，其清零所需时间对应 cpu 很多个周期。解决方案是给 cpu 编程时添加一些空指令来等待串口数据清零，之后再读取下一个数据。代码见 uart.v：

```
module UART(clk,reset,Add,rdata,rd,wdata,wr,UART_RXD,UART_TXD,sig,sig16);
input clk,reset,UART_RXD,sig,sig16,rd,wr;
input [31:0]wdata,Add;
wire [7:0]led;
output reg [31:0]rdata;
//output [7:0]led;
reg   [2:0] CON;
output UART_TXD;
wire [7:0] RX_DATA;
reg [7:0]data;
reg[7:0] tx;
wire RX_status,TX_STATUS,TX_EN;
```

```
    initial begin
        rdata=32'b0;
        data=8'b0;
        CON=3'b0;
    end
    assign led[0]=RX_status;
    assign led[1]=TX_STATUS;
    assign led[2]=TX_EN;
    always @(clk) begin
      CON[0]=TX_STATUS;
      CON[1]=RX_status;
    end
    always @(posedge RX_status or posedge clk or negedge reset) begin
        if(reset==0)
           data=8'b0;
        else data=RX_DATA;
    end
    //assign led=RX_DATA;
    always @(posedge clk) begin
        if(rd) begin
            case(Add)
                32'h4000001C:begin
                    rdata={24'b0,data};
                end
                32'h40000020:rdata={30'b0,CON};
                default:rdata=32'b0;
            endcase
        end
        else if(RX_status==0)rdata=32'b0;
    end

    always @(posedge clk) begin
        if(wr==1&&Add==32'h40000018&&TX_STATUS==1)
            tx=wdata[7:0];
        else if(wr==1&&Add==32'h40000020)
            CON[2]=wdata[2];
    end
    recevier
rx(.in(UART_RXD),.clk(clk),.sig16(sig16),.reset(reset),.out(RX_DATA),.rx_status(RX_status));
    controller
con(.clk(clk),.RX_status(RX_status),.RX_DATA(RX_DATA),.TX_STATUS(TX_STATUS),.TX_EN(TX_EN),.r
eset(reset),.Add(Add),.wr(wr),.wdata(wdata));
    transmitter
tra(.clk(clk),.sig(sig),.reset(reset),.trans(tx),.tx_en(TX_EN),.tx_status(TX_STATUS),.out(UART_TXD));
```

```
endmodule

module controller(clk,reset,RX_status,RX_DATA,TX_STATUS,TX_EN,Add,wr,wdata);
input clk,RX_status,TX_STATUS,reset,wr;
input [31:0] wdata,Add;
input [7:0] RX_DATA;
output reg TX_EN;
initial begin
    TX_EN<=0;
end


always@(posedge clk) begin
    if(Add==32'h40000018 && TX_STATUS==1 && wr==1) begin
        TX_EN=1;
    end
    else TX_EN=0;
end
endmodule


module recevier(in,clk,sig16,reset,out,rx_status);
input clk,reset,sig16,in;
reg flag;
reg [7:0] out1;
reg [20:0] temp;//
output   reg[7:0] out;
output reg rx_status;
initial begin
    flag=0;
    temp=0;
    rx_status=0;
end

always@(negedge sig16 or negedge reset)//
begin
    if(~reset)
    begin
        flag=0;
        out=0;
        out1=0;
    end
    else   begin
        if(in==0&&flag==0)
```

```
            begin
                flag=1;
            end
            if(flag==1)
            begin
                temp=temp+21'b1;
                if(temp==24)
                    out1[0]=in;
                if(temp==40)
                    out1[1]=in;
                if(temp==56)
                    out1[2]=in;
                if(temp==72)
                    out1[3]=in;
                if(temp==88)
                    out1[4]=in;
                if(temp==104)
                    out1[5]=in;
                if(temp==120)
                    out1[6]=in;
                if(temp==136)
                begin
                    out1[7]=in;
                    rx_status=1;
                end
                if(temp==152)
                    rx_status=0;
                if(temp>152)
                begin
                    flag=0;
                    temp=0;
                end
            end
            if(rx_status==1) out=out1;
            else out=8'b0;
        end
    end
    endmodule

    module transmitter(clk,sig,reset,trans,tx_en,tx_status,out);
    input clk,sig,reset,tx_en;
    input [7:0] trans;
    reg flag1,flag2;
    reg [3:0]count;
```

```
output reg out;
output reg tx_status;
reg tmp;
initial begin
    flag1=0;
    flag2=0;
    out=1;
    tx_status=1;
    count=0;
end

always@(posedge tx_en or posedge tmp) begin
    if(tx_en) begin
        flag1=1;
        tx_status=0;
    end
    else begin
        if(tmp) begin
            flag1=0;
            tx_status=1;
        end
    end
end
always@(posedge sig or negedge reset)
begin
    if(~reset)
    begin
        flag2=0;
        count=0;
    end
    else
    begin
        if(flag1==1)
        begin
            if(~flag2)
            begin
                out=0;
                flag2=1;
            end
            else
            begin
                out=trans[count];
                count=count+4'b1;
            end
```

```
            end
            else
            begin
                out=1;
            end
            if(count==4'b1000)
            begin
                tmp=1;flag2=0;count=0;
            end
            else
                tmp=0;
        end
    end
    endmodule
```

## 5) 外设—魏齐辉

主要是 LED 灯和数码管的处理。在实验文件中已经给出了外设模块，这里的 LED 灯是可以直接使用的；数码管需要进行驱动，本实验中同时实验了硬驱动（作为前期调试使用）和软驱动的方法，软驱动的硬件设计和给的范例是一致的，在汇编程序中，当出现中断时，利用 jal 跳转过来（这里涉及到 cpu 运行的监督位问题，解决方案见编译器部分）：

见 code.asm

```
DECODE:addi $t3,$zero, 128
addi $t5,$v1, 0
SHIFTL:beq $t3,$v0, SHIFTR
sll $t5,$t5, 4
srl $t3,$t3, 1
j SHIFTL
SHIFTR:srl $t5,$t5, 28
sll $t3,$t3, 8
addi $t6,$zero, 0
beq $t6,$t5, ZERO
addi $t6,$t6, 1
beq $t6,$t5, ONE
addi $t6,$t6, 1
beq $t6,$t5, TWO
addi $t6,$t6, 1
beq $t6,$t5, THREE
addi $t6,$t6, 1
beq $t6,$t5, FOUR
addi $t6,$t6, 1
beq $t6,$t5, FIVE
```

```
addi $t6,$t6, 1
beq $t6,$t5, SIX
addi $t6,$t6, 1
beq $t6,$t5, SEVEN
addi $t6,$t6, 1
beq $t6,$t5, EIGHT
addi $t6,$t6, 1
beq $t6,$t5, NIGHT
addi $t6,$t6, 1
beq $t6,$t5, A
addi $t6,$t6, 1
beq $t6,$t5, B
addi $t6,$t6, 1
beq $t6,$t5, C
addi $t6,$t6, 1
beq $t6,$t5, D
addi $t6,$t6, 1
beq $t6,$t5, E
addi $t6,$t6, 1
beq $t6,$t5, F
ZERO:addi $t5,$zero, 64
j DE
ONE:addi $t5,$zero, 121
j DE
TWO:addi $t5,$zero, 36
j DE
THREE:addi $t5,$zero, 48
j DE
FOUR:addi $t5,$zero, 25
j DE
FIVE:addi $t5,$zero, 18
j DE
SIX:addi $t5,$zero, 2
j DE
SEVEN:addi $t5,$zero, 120
j DE
EIGHT:addi $t5,$zero, 0
j DE
NIGHT:addi $t5,$zero, 16
j DE
A:addi $t5,$zero, 8
j DE
B:addi $t5,$zero, 3
j DE
```

```
C:addi $t5,$zero, 70
j DE
D:addi $t5,$zero, 33
j DE
E:addi $t5,$zero, 6
j DE
F:addi $t5,$zero, 14
j DE
DE:add $t5,$t3, $t5
sw $t5,20($s5)
sll $v0,$v0, 1
addi $t3,$zero, 16
addi $t1,$zero, 3
sw $t1,8($s5)
bne $v0,$t3, JUMP
addi $v0,$zero, 1
JUMP:jr $xp
```

硬驱动的方案可见 single_hard 的 digitube_scan.v 文件和 Peripheral.v 文件，其汇编软件的对应部分要短很多，如下：

```
DECODE:addi $t3,$zero, 128
addi $t5,$v1, 0
SHIFTL:beq $t3,$v0, SHIFTR
sll $t5,$t5, 4
srl $t3,$t3, 1
j SHIFTL
SHIFTR:srl $t5,$t5, 28
sll $t3,$t3, 4
add $t5,$t3, $t5
sw $t5,20($s5)
sll $v0,$v0, 1
addi $t3,$zero, 16
addi $t1,$zero, 3
sw $t1,8($s5)
bne $v0,$t3, JUMP
addi $v0,$zero, 1
JUMP:jr $xp
```

# 四、 流水线

## 1. 实验要求

在单周期 MIPS 处理器的基础上，设计一个 5 级流水线的 MIPS 处理器（30%）。

采用如下方法解决竞争问题：

*i* 采用完全的 forwarding 电路解决数据关联问题。

*ii* 对于 Load-use 类竞争采取阻塞一个周期+Forwarding 的方法解决

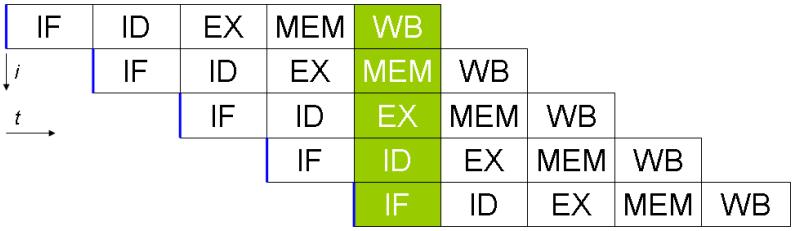
*iii* 对于分支指令在 EX 阶段判断（提前判断也可以），在分支发生时刻取消 ID 和 IF 阶段的两条指令。

*iv* 对于 J 类指令在 ID 阶段判断，并取消 IF 阶段指令。

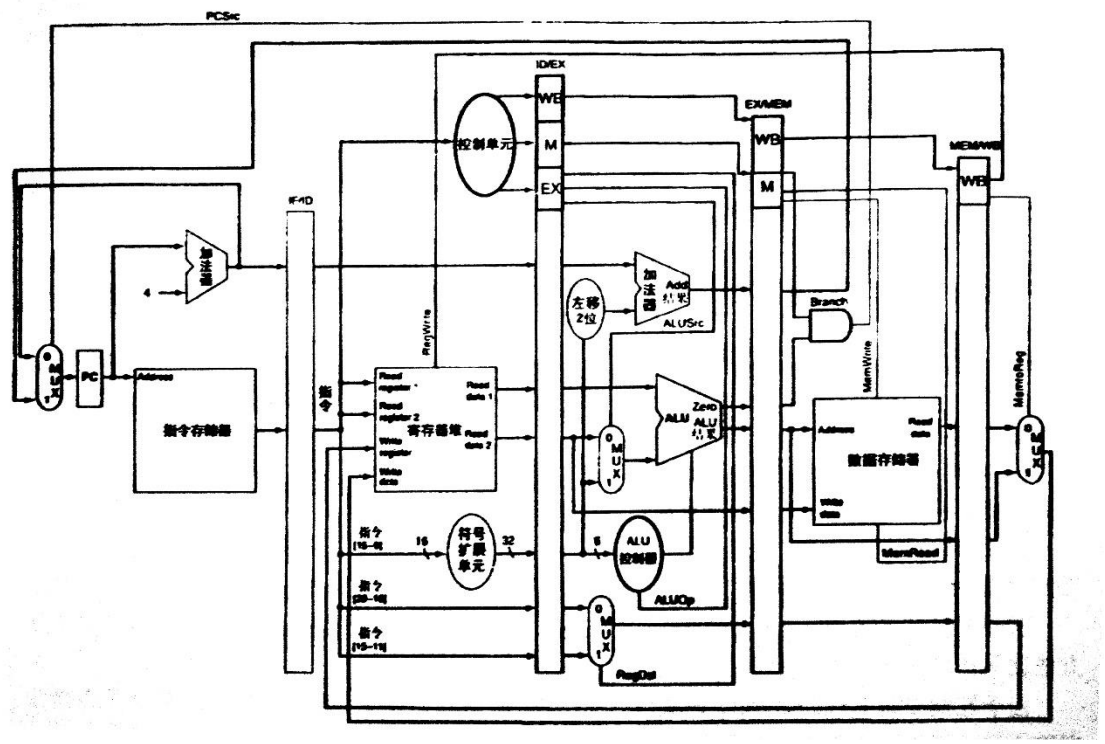
将计算最大公约数的程序在流水线 MIPS 处理器中正常运行。

## 2. 设计思路

本 CPU 指令与数据分开存储，为哈佛结构，其流水线结构如《计算机体系结构：软硬件结构》所示，为




数据通路类似下图（ALU 和 PC 处理部分有所不同）。Control 模块在 ID 部分，实验中就按此分块调试。




本部分任务主要如下：

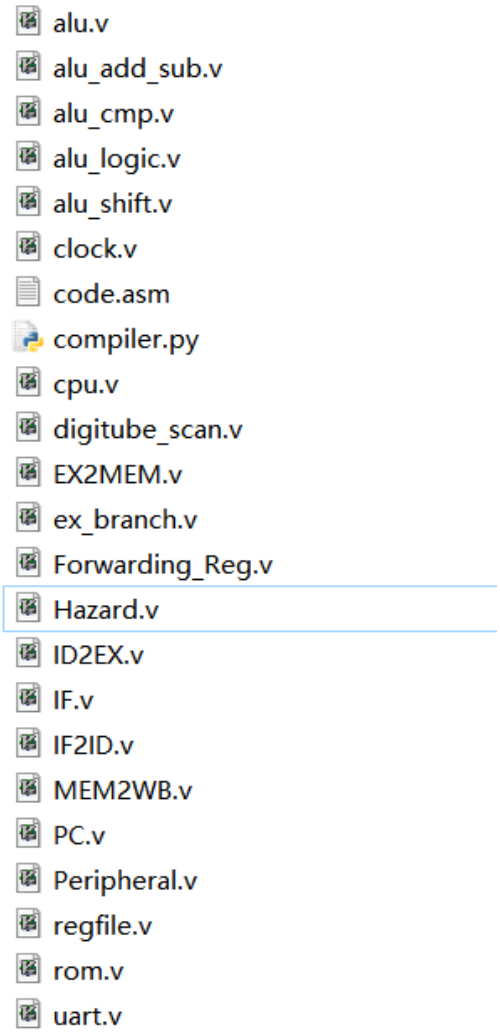
A. 流水线寄存器设计。

B. 冒险检测与转发设计

C. 原有单周期代码改写。

# 3. 设计实现

本部分按照之前的流程进行解决，代码见 pip 文件夹中.
文件清单如下




## 1) 流水线寄存器设计—聂浩

在对 clk 上升沿出发的 always 块中对值进行赋值即可延时一个周期，所有的流水线寄存器都是此原理；难题在于对转发、冒险的处理，因此在 IF，IF2ID，ID2EX 中都加入了相应的信号进行处理。

### i IF.v

```
`timescale 1ns/1ps
module IF(writeEn, reset, clk, Pc_In, Pc_Out);
```

```
input   writeEn, reset, clk;
input   [31:0]  Pc_In;
output  reg [31:0]  Pc_Out;

always @(negedge reset or posedge clk) begin
    if(!reset) begin
        Pc_Out<=32'h80000000;
    end
    else begin
        if(writeEn) begin
            Pc_Out<=Pc_In;
        end
        else begin
            Pc_Out<=Pc_Out;
        end
    end
end

endmodule
```

### ii IF2ID.v

```
module IF2ID(rst, clk,
            flush, writeEn,
            PCIn, instructionIn,
            PCOut, instructionOut
            );

input   writeEn, flush, rst, clk;
input   [31:0]  PCIn;
input   [31:0]  instructionIn;

output  reg [31:0]  PCOut;
output  reg [31:0]  instructionOut;

parameter init= 32'h0000_0000;

always @(negedge rst or posedge clk) begin
    if(!rst) begin
        PCOut<=init;
```

```
        instructionOut<=init;
    end
    else if (flush) begin
            PCOut<=init;
            instructionOut<=init;
        end
        else begin
              if(writeEn) begin
                  PCOut<=PCIn;
                  instructionOut<=instructionIn;
              end
              else begin
                  PCOut<=PCOut;
                  instructionOut<=instructionOut;
              end
        end
end

endmodule
```

### iii ID2EX.v

```
`timescale 1ns/1ps
module EX(rst, clk, flush,
          RegDstIn, ALUFunIn, ALUSrc1In, ALUSrc2In, SignIn, MemRdIn,
MemWrIn, RegWrIn, MemToRegIn,
          EX_rsContentIn, EX_rtContentIin, EX_rtIn, EX_rdIn,
imm32_in,
          RegDstOut, ALUFunOut, ALUSrc1Out, ALUSrc2Out, SignOut,
MemRdOut, MemWrOut, RegWrOut, MemToRegOut,
          EX_rsContentOut, EX_rtContentOut, EX_rtOut, EX_rdOut,
imm32_out,
          PCAdd4in,PCAdd4out
          );


input   flush, rst, clk;
input   ALUSrc1In, ALUSrc2In, SignIn, MemRdIn, MemWrIn, RegWrIn;
input   [1:0]   MemToRegIn,RegDstIn;
input   [4:0]    EX_rtIn, EX_rdIn;
input   [5:0]   ALUFunIn;
input   [31:0]  EX_rsContentIn,EX_rtContentIin;
```

```
input   [31:0]  imm32_in,PCAdd4in;

output  reg  ALUSrc1Out, ALUSrc2Out, SignOut, MemRdOut, MemWrOut,
RegWrOut;
output  reg [1:0]   MemToRegOut,RegDstOut;
output  reg [4:0]    EX_rtOut, EX_rdOut;
output  reg [5:0]   ALUFunOut;
output  reg [31:0]  imm32_out;
output  reg [31:0]  EX_rsContentOut,EX_rtContentOut,PCAdd4out;
//the flush and enable is important
always @(negedge rst or posedge clk) begin
  if(~rst) begin
  RegWrOut<=1'b0;
    SignOut<=1'b0;
    MemRdOut<=1'b0;
    MemWrOut<=1'b0;
    ALUSrc2Out<=1'b0;
    ALUSrc1Out<=1'b0;
    RegDstOut<=2'b0;
    MemToRegOut<=2'b00;
    EX_rdOut<=5'b00000;
    EX_rtOut<=5'b00000;
    ALUFunOut<=6'b000000;
    imm32_out<=32'h00000000;
    PCAdd4out<=32'h00000000;
    EX_rtContentOut<=32'h00000000;
    EX_rsContentOut<=32'h00000000;

  end
  else if (flush) begin
          MemToRegOut<=2'b00;
          RegDstOut<=2'b0;
          ALUSrc1Out<=1'b0;
          EX_rtOut<=5'b00000;
          EX_rdOut<=5'b00000;
          SignOut<=1'b0;
          ALUSrc2Out<=1'b0;
          PCAdd4out<=32'h00000000;
          MemRdOut<=1'b0;
          MemWrOut<=1'b0;
          imm32_out<=32'h00000000;
          ALUFunOut<=6'b000000;
          RegWrOut<=1'b0;
          EX_rsContentOut<=32'h00000000;
```

```
            EX_rtContentOut<=32'h00000000;
        end
    else begin
            MemRdOut<=MemRdIn;
            MemWrOut<=MemWrIn;
            EX_rtOut<=EX_rtIn;
            EX_rdOut<=EX_rdIn;
            EX_rsContentOut<=EX_rsContentIn;
            EX_rtContentOut<=EX_rtContentIin;
            MemToRegOut<=MemToRegIn;
            PCAdd4out<=PCAdd4in;
            imm32_out<=imm32_in;
            ALUFunOut<=ALUFunIn;
            RegDstOut<=RegDstIn;
            ALUSrc1Out<=ALUSrc1In;
            ALUSrc2Out<=ALUSrc2In;
            SignOut<=SignIn;
            RegWrOut<=RegWrIn;
    end
end

endmodule
```

### iv EX2MEM.v

```
module EX2MEM(rst, clk,
            MemRdIn, MemWrIn, RegWrIn, RegIn, ALUoutIn, MEM_BIn,
MEM_rdIn,
            MemRdOut, MemWrOut, RegWrOut, RegOut, ALUoutOut,
MEM_BOut, MEM_rdOut,
            PCAdd4in, PCAdd4out
            );

input   rst, clk;
input   MemRdIn, MemWrIn, RegWrIn;

input   [1:0]   RegIn;
input   [4:0]   MEM_rdIn;
input   [31:0]  ALUoutIn,MEM_BIn,PCAdd4in;
```

```
output  reg  MemRdOut, MemWrOut, RegWrOut;
output  reg [1:0] RegOut;
output  reg [4:0] MEM_rdOut;
output  reg [31:0] ALUoutOut,MEM_BOut,PCAdd4out;

always @(posedge clk or negedge rst) begin
    if(~rst) begin
        MemRdOut <= 1'b0;
        RegWrOut <= 1'b0;
        MemWrOut <= 1'b0;
        RegOut <= 2'b00;
        MEM_rdOut <= 5'b00000;
        PCAdd4out <= 32'h00000000;
        ALUoutOut <= 32'h00000000;
        MEM_BOut <= 32'h00000000;
    end
    else begin
        MemRdOut <= MemRdIn;
        MemWrOut <= MemWrIn;
        PCAdd4out <= PCAdd4in;
        MEM_rdOut <= MEM_rdIn;
        RegOut <= RegIn;
        MEM_BOut <= MEM_BIn;
        ALUoutOut <= ALUoutIn;
        RegWrOut <= RegWrIn;
    end
end

endmodule
```

- **MEM2WB.v**

```
`timescale 1ns/1ps
module MEM2WB(rst, clk,
            RegWrIn, MemToRegIn, ALUoutin, MEMDatain, WB_rdIn,
            RegWrOut, MemToRegOut, ALUoutOut, MEMDataOut, WB_rdOut,
            PCAdd4In, PCAdd4Out
            );

input   RegWrIn,rst, clk;
input   [1:0]   MemToRegIn;
```

```
input   [4:0]   WB_rdIn;
input   [31:0]  ALUoutin,MEMDatain,PCAdd4In;

output  reg  RegWrOut;
output  reg [1:0]   MemToRegOut;
output  reg [4:0]   WB_rdOut;
output  reg [31:0]  ALUoutOut,MEMDataOut,PCAdd4Out;


always @(posedge clk or negedge rst) begin
    if(~rst) begin
        WB_rdOut<=5'b00000;
        ALUoutOut<=32'h00000000;
        MEMDataOut<=32'h00000000;
        PCAdd4Out<=32'h00000000;
        MemToRegOut<=2'b00;
        RegWrOut<=1'b0;
    end
    else begin
        WB_rdOut<=WB_rdIn;
        MemToRegOut<=MemToRegIn;
        RegWrOut<=RegWrIn;
        ALUoutOut<=ALUoutin;
        MEMDataOut<=MEMDatain;
        PCAdd4Out<=PCAdd4In;
    end
end

endmodule
```

2) 转发与其气泡模块的设计—魏齐辉

## forward_reg

模块用于解决数据冒险，通过沿着数据通路从沿着 EX、MEM、WB 依次比对 ID 指令的中的 rt 与 rs，出现一致则通过修改 forwardA 或 forwardB 的值使得 DatabusA 或 DatabusB 的值（ID2EX 去寄存的值）变为对应值，从而实现转发，测试表明，这一部分表现良好。
代码如下

```
`timescale 1ns/1ps
module Forwarding_Reg(ID_rs, ID_rt, EX_rd,
                      EX_RegWrite,
                      MEM_rd, MEM_RegWrite, MEM_MemToReg,
```

```
                    MEM2WB_rd, MEM2WB_RegWrite,
                    ForwardA, ForwardB
                    );

input   EX_RegWrite, MEM_RegWrite, MEM2WB_RegWrite;
input   [1:0]  MEM_MemToReg;
input   [4:0]  ID_rs, ID_rt, EX_rd, MEM_rd, MEM2WB_rd;

output reg   [2:0]  ForwardA, ForwardB;

always @(*) begin
  //DatabusA
    if(EX_RegWrite && (EX_rd != 5'd0) && (EX_rd == ID_rt)) begin
        ForwardB = 3'b001;
    end
    else
        if(MEM_RegWrite && (MEM_rd != 5'd0) && (MEM_rd == ID_rt))
begin
            if(MEM_MemToReg == 2'b01) begin
                ForwardB = 3'b011;
            end
            else begin
                ForwardB = 3'b010;
            end
        end
        else begin
            if(ID_rt == MEM2WB_rd && MEM2WB_RegWrite) begin
                ForwardB = 3'b100;
            end
            else begin
                ForwardB = 3'b000;
            end
        end

  //DatabusB //the same as databusA
  if(EX_RegWrite && (EX_rd == ID_rs) && (EX_rd != 5'd0)) begin
        ForwardA = 3'b001;
    end
    else
        if(MEM_RegWrite && (MEM_rd != 5'd0) && (MEM_rd == ID_rs))
begin
            if(MEM_MemToReg == 2'b01) begin
                ForwardA = 3'b011;
            end
```

```
            else begin
                ForwardA = 3'b010;
            end
        end
        else begin
          if(ID_rs == MEM2WB_rd && MEM2WB_RegWrite) begin
              ForwardA = 3'b100;
          end
          else begin
              ForwardA = 3'b000;
          end
        end

end
endmodule
```

## ex_branch

用于 beq 等分支指令，因为之前是否进入分支是由 ALU 的结果决定的，但这样需要等到 EX 阶段才能判断转发，为了和指令一致而将这一过程转移到 ID 阶段，这一模块相当于把 ALU 的 cmp 模块单独提取了出来。工作是正常的。

```
`timescale 1ns/1ps
module forward_(databusA,databusB,BranchType,compareAB);

input  [2:0] BranchType;//BranchType
input  [31:0] databusA,databusB;
output reg compareAB;

//from the single cpu by Neil
always @ (*) begin
  case(BranchType)
    3'b001: compareAB = (databusA == databusB) ? 1'b1:1'b0;
    3'b010: compareAB = (databusA < databusB)  ? 1'b1:1'b0;
    3'b100: compareAB = (databusA > 0 || databusA == 0) ? 1'b1:1'b0;
    3'b101: compareAB = (databusA != databusB) ? 1'b1:1'b0;
    3'b110: compareAB = (databusA[31] == 1 || databusA == 0) ?
1'b1:1'b0;
    3'b111: compareAB = (databusA > 0)  ? 1'b1:1'b0;
    default: compareAB = 1'b0 ;
  endcase
end

endmodule
```

## Hazard 模块，

用于产生气泡，和 flush 信号以及控制是否读取下一个 PC，代码如下:

```
module Hazard(ID_rs, ID_rt,
              ID_Flush, ID_Write,
              EX_MemRead, EX_rt, EX_Flush,
              BranchType, J, jHazard, IFWrite
              );


input   EX_MemRead;
input   J, jHazard;

input [4:0] ID_rs, ID_rt, EX_rt;
input [2:0] BranchType;

output  reg IFWrite, ID_Write,ID_Flush,EX_Flush;

always @(*) begin
    if(EX_MemRead && ((EX_rt == ID_rs && ID_rs != 5'd0) || (EX_rt ==
ID_rt && ID_rt != 5'd0 ))) begin
        EX_Flush = 1'b1;//flush,generate a bubble
        ID_Flush = 1'b0;//hold
        IFWrite = 1'b0;//hold
        ID_Write = 1'b0;//hold
    end
    else begin
        if(J || ((BranchType[0] || BranchType[1] || BranchType[2]) &&
jHazard)) begin
            EX_Flush = 1'b0;
            ID_Flush = 1'b1;
            IFWrite = 1'b1;
            ID_Write = 1'b0;
        end
        else begin
            IFWrite = 1'b1;
            ID_Write = 1'b1;
            EX_Flush = 1'b0;
            ID_Flush = 1'b0;
        end
    end
end
```

```
endmodule
```

3) 单周期代码的修改—聂浩

i. 顶层模块修改

将流水线模块加入其中，这里需要注意的是各个变量的命名，很容易乱掉
代码如下

```
//by Neil 2015/9/6
//CPU
module
cpu(sysclk,reset,switch,digi_out1,digi_out2,digi_out3,digi_out4,led,U
ART_RXD,UART_TXD);
input sysclk,reset;
input [7:0]switch;
input UART_RXD;
output [6:0]digi_out1,digi_out2,digi_out3,digi_out4;
output [7:0]led;
output UART_TXD;


wire clk;
wire sig,sig16;
wire [11:0]digi;
wire [31:0]PC_add4;
wire   [31:0]  IFInst;
wire [31:0]Pc;
reg [31:0]ConBA;
//IF
wire [25:0]JT;
wire [31:0]Pc_in;
//ID
wire   [31:0]  IDInst;
wire   [31:0]  ID_PCAdd4;
wire irq;
wire [31:0]instruct;
wire [15:0]Imm16;
wire [4:0]Shamt;
wire [4:0]Rd,Rt,Rs;
wire [5:0]Opcode;
wire [5:0]Funct;
```

```
wire [2:0] PCSrc;
wire   [1:0]  RegDstIn;
wire   RegWrIn, ALUSrc1In, ALUSrc2In;
wire   [5:0]  ALUFunIn;
wire   SignIn, MemWrIn, MemRdIn;
wire   [1:0]  MemtoRegIn;
wire   [2:0]  BranchType;
wire ExtOp;
wire LuOp;
wire [4:0]RegAddrWr;
wire [31:0]DataBusA,DataBusB;
wire   [31:0]  RegFileA;
wire   [31:0]  RegFileB;
wire [31:0]lu,Ext_out;
wire   [4:0]  ID2EX_rt,ID2EX_rd;
//EX
wire [31:0]AluA,AluB,EX2MEM_ALUoutOut;
wire   [31:0]  imm32_out;
wire   [5:0]  ALUFunOut;
wire   ALUSrc1Out, ALUSrc2Out, SignOut;
wire   [1:0]  ID2EX_RegDstOut;
wire   ID2EX_MemRdOut, ID2EX_MemWrOut, ID2EX_RegWrOut;
wire   [1:0]  ID2EX_MemToRegOut;
wire   [31:0]  EX2PCAdd4_out;
wire   [31:0]  EX_ALUout;
wire   [4:0]  EX2MEM_rt,EX2MEM_rd;
wire   [4:0]  EX2MEM_RegAddrWr;
//MEM
wire   [31:0]  DataBusB_out;
wire   [31:0]  EX2MEM_rsData, EX2MEM_rtData;
wire   [1:0]  EX2MEM_MemToRegOut;
wire   EX2MEM_MemRdOut, EX2MEM_MemWrOut, EX2MEM_RegWrOut;
wire   [31:0]  MemReadData;
wire   [31:0]  MEM_PCAdd4Out;
wire [31:0]pout,dmout,uout;
wire   [4:0]  MEM2WB_RegAddrWr;

//WB
wire [1:0]MEM2WB_MemToRegOut;
wire   [31:0]  MEM2WB_ALUoutOut;
wire   [31:0]  MEM2WB_MEMDataOut;
wire   [31:0]  RegWriteData,WB2PCAdd4Out;
wire   MEM2WB_RegWrOut;
//forward||haz
```

```
wire    [2:0]   ForwardA;
wire    [2:0]   ForwardB;
wire    PcNextEnable;
wire    IF2ID_Flush;
wire    IF2ID_Write;
wire    ID2EX_Flush;
wire JHazardCom;
wire J;//J 指令的具体值

assign  JT=IDInst [25:0];
always@(posedge clk) ConBA=(Ext_out<<2)+Pc;

cloc ck(sysclk,clk,sig,sig16);

PC pc(clk,reset,ConBA,DataBusA,JT,PCSrc,Pc_in,PC_add4,JHazardCom);


ROM rom(Pc,IFInst);
assign  Opcode  =IDInst[31:26];
assign  Imm16   =IDInst[15:0];
assign  Shamt   =IDInst[10:6];
assign  Rd      =IDInst[15:11];
assign  Rt      =IDInst[20:16];
assign  Rs      =IDInst[25:21];
assign  Funct   =IDInst[5:0];

IF PCIF(PcNextEnable,reset,clk,Pc_in,Pc);
IF2ID IFID(reset,clk,
          IF2ID_Flush,//if beq, flush all
          IF2ID_Write,PC_add4,
          IFInst,
          ID_PCAdd4,
          IDInst
       );



Control control(Opcode,  Funct, irq , reset,
             PCSrc,
             RegDstIn,  RegWrIn,
             ALUSrc1In, ALUSrc2In,    ALUFunIn,
             SignIn,
             MemRdIn, MemWrIn,   MemtoRegIn,
             ExtOp, LuOp,J,BranchType);
```

```
assign RegAddrWr=(ID2EX_RegDstOut==2'b0)?EX2MEM_rd:
             (ID2EX_RegDstOut==2'b01)?EX2MEM_rt:
             (ID2EX_RegDstOut==2'b10)?5'd31:
             5'd26;

RegFile
rf(reset,clk,Rs,RegFileA,Rt,RegFileB,MEM2WB_RegWrOut,MEM2WB_RegAddrWr
,RegWriteData);
assign
Ext_out=(Imm16[15]==1&&ExtOp==1)?{16'b1111_1111_1111_1111,Imm16}:{16'
b0000_0000_0000_0000,Imm16};
assign lu=(LuOp==1'b1)?{Imm16,16'b0}:Ext_out;
ID2EX IDtoEX(
           reset,clk,
           ID2EX_Flush,RegDstIn,//if beq, flush all
           ALUFunIn, ALUSrc1In, ALUSrc2In,
           SignIn, MemRdIn, MemWrIn,
           RegWrIn, MemtoRegIn,
           DataBusA, DataBusB,
           Rt, Rd,
           lu,
           ID2EX_RegDstOut,ALUFunOut,
           ALUSrc1Out,ALUSrc2Out,
           SignOut,ID2EX_MemRdOut,
           ID2EX_MemWrOut,
           ID2EX_RegWrOut,
           ID2EX_MemToRegOut,
           EX2MEM_rsData,EX2MEM_rtData,
           EX2MEM_rt,EX2MEM_rd,
           imm32_out,
           ID_PCAdd4,
           EX2PCAdd4_out
           );

assign AluA=(ALUSrc1Out==1'b1)?{27'b0,imm32_out[10:6]}:EX2MEM_rsData;
assign AluB=(ALUSrc2Out==1'b1)?imm32_out:EX2MEM_rtData;

ALU alu(AluA,AluB,SignOut,ALUFunOut,EX_ALUout);
EX2MEM EXMEM(
           reset,clk,
           ID2EX_MemRdOut, ID2EX_MemWrOut,
           ID2EX_RegWrOut,
           ID2EX_MemToRegOut,
```

```
            EX_ALUout,
            EX2MEM_rtData,RegAddrWr,
            EX2MEM_MemRdOut,
            EX2MEM_MemWrOut, EX2MEM_RegWrOut,
            EX2MEM_MemToRegOut,
            EX2MEM_ALUoutOut,
            DataBusB_out,
            EX2MEM_RegAddrWr,
            EX2PCAdd4_out,
            MEM_PCAdd4Out
            );


Peripheral
per(reset,clk,EX2MEM_MemRdOut,EX2MEM_MemWrOut,EX2MEM_ALUoutOut,DataBu
sB_out,pout,led,switch,digi,irq);
digitube_scan ds(digi,digi_out1,digi_out2,digi_out3,digi_out4);

DataMem
dm(reset,clk,EX2MEM_MemRdOut,EX2MEM_MemWrOut,EX2MEM_ALUoutOut,DataBus
B_out,dmout);

UART
ua(clk,reset,EX2MEM_ALUoutOut,uout,EX2MEM_MemRdOut,DataBusB_out,EX2ME
M_MemWrOut,UART_RXD,UART_TXD,sig,sig16);

assign
MemReadData=(EX2MEM_ALUoutOut<32'h40000000)?dmout:(EX2MEM_ALUoutOut>=
32'h40000018)?uout:pout;


assign RegWriteData=(MEM2WB_MemToRegOut[1])?
                     ((MEM2WB_MemToRegOut[0])? 32'b0:WB2PCAdd4Out):
                     ((MEM2WB_MemToRegOut[0])?
MEM2WB_MEMDataOut:MEM2WB_ALUoutOut);


MEM2WB MEMWB(
            reset,clk,
            EX2MEM_RegWrOut,
            EX2MEM_MemToRegOut,
            EX2MEM_ALUoutOut,
            MemReadData,
            EX2MEM_RegAddrWr,
```

```
            MEM2WB_RegWrOut,
            MEM2WB_MemToRegOut,
            MEM2WB_ALUoutOut,
            MEM2WB_MEMDataOut,
            MEM2WB_RegAddrWr,
            MEM_PCAdd4Out,WB2PCAdd4Out
           );



Forwarding_Reg Forwardingr(
                        IDInst[25:21],IDInst[20:16],
                        RegAddrWr,
                        ID2EX_RegWrOut,
                        EX2MEM_RegAddrWr,
                        EX2MEM_RegWrOut,
                        EX2MEM_MemToRegOut,
                        MEM2WB_RegAddrWr,
                        MEM2WB_RegWrOut,
                        ForwardA, ForwardB
                        );
ex_branch eb(DataBusA,DataBusB,BranchType,JHazardCom);

Hazard Hazard_Detection(
                    IDInst[25:21],IDInst[20:16],
                    IF2ID_Flush,
                    IF2ID_Write,
                    ID2EX_MemRdOut,
                    EX2MEM_rt,
                    ID2EX_Flush,
                    BranchType,J,JHazardCom,
                    PcNextEnable
              );



   assign DataBusA=(ForwardA==3'b000)?RegFileA:
       (ForwardA==3'b001)?EX_ALUout:
       (ForwardA==3'b011)?MemReadData:
       (ForwardA==3'b100)?RegWriteData:
       (ForwardA==3'b010)?EX2MEM_ALUoutOut:
       RegFileA;
```

```
    assign DataBusB =(ForwardB==3'b000)?RegFileB:
    (ForwardB==3'b010)?EX2MEM_ALUoutOut:
    (ForwardB==3'b011)?MemReadData:
    (ForwardB==3'b001)?EX_ALUout:
    (ForwardB==3'b100)?RegWriteData:
     RegFileB;
endmodule
```

### control 模块修改

因为阻塞信号，提前判断信号（ex_branch）需要 J 型指令和 branch 指令的具体信息，所以 control 增加 J 和 BranchType 信号，因为改动不大，这里不再贴代码。

### 汇编程序的修改

阻塞模块还存在一些问题（没有对 IF 模块进行 flush 操作），导致阻塞一个周期后还要经过一个周期才能跳转，这一周期会读取跳转指令之后的第二条指令，由此产生一些问题。

这里采取了在跳转指令后增加两条空指令的方式进行解决，最终得以完成设计目标。具体见 code.asm

## 五、 编译器的设计与优化—王璞瑞

最开始我们希望设计的编译器是这样的，能够将简单的指令集转换为 32 位机器码，后来发现由于跳转指令和分支指令（如 jal， ,beq 等）的存在，使得我们必须能够计算内存地址，对照 rom.v 指令参考后发现，可以利用 python 强大的字符串处理功能实现生成内存地址的功能。

在编译器中我们定义了这样的函数，方便后续使用。

首先是 findname 函数，由于 MIPS 指令格式大致相同，如 add $S1,$s2,$s3，与此相同格式的指令还有很多，我们希望能够提取出其中的关键字，注意到每个关键字两侧的的分隔符不同，如第一个指令名 add，其两端的分隔符分别是是开始符和‘ $’，第一个寄存器名字 s1 两端的分割符是’ $’和‘,$’，而第三个 s3 的分隔符为’,$’与句末的’\n’，因此我们迫切的需要一个函数，这个函数可以实现这样的功能，输入一个字符串和待求字符串的起止符，输出中间截取出的字符串，有了这样的思路，findname 函数很快就出来了，如下所示：

```
def findname(str_begin,str_end,html):
    begin = html.find(str_begin)
    if begin >= 0:
        begin += len(str_begin)
        str_end = html.find(str_end,begin)
        if str_end >= 0:
            return html[begin:str_end].strip()
```

这个函数虽然好用，但也会面临问题，很快就能发现。

在有了这个函数之后，我们接下来该如何处理收集到的寄存器名字的字符串呢？经过思考，我们想到了这样的方法，按照 0~31 号寄存器的顺序，构造了一个寄存器字符串数组，如下所示：

```
reg=['zero','at','v0','v1','a0','a1','a2','a3',
    't0','t1','t2','t3','t4','t5','t6','t7',
    's0','s1','s2','s3','s4','s5','s6','s7',
    't8','t9','xp','k1','gp','sp','fp','ra']
```

设我们找到的寄存器名字为 rs，那么，我们发现，只要能够在 reg 数组中找到与 rs 相同的值并取出其下标，这个下标就是我们希望的 reg 的编号，这个可以利用 python 中针对数组的 reg.index(rs)来实现，但是为了使其变为固定长度的二进制码，还需要利用 bin 函数使其变为二进制，并截掉二进制前面自带的‘0b’，然后利用 zfill 函数使其扩充为指定位数，基于以上思考，我们设计了这样的函数 bin1，他可以实现将寄存器名字字符串变为 num 位长得二进制字符串。函数如下：

```
def bin1(str,num):
    return (bin(reg.index(str))[2:]).zfill(num)
```

在得到了这样两个很好的函数后，我们可以对 add 同样的函数做相应的处理了，在此不全部列举，仅挑选对 add 进行处理的一段代码来展示。

```
    if (line[0:3] == "add") & (line[3] == ' '):
        rd = findname(' $',',',line)
        rs = findname(',$',',',line)
        rt = findname(', $','\n',line)
        code = '000000' + bin1(rs,5) + bin1(rt,5) + bin1(rd,5) +
'00000' + '100000'
```

细心的助教一定已经发现了我们在提取 rt 的时候第一个分隔符有一些异样，是的，由于’,$’这样的字符串不仅存在于 rt 之前，也存在于 rs 之前，而我们的 findname 函数是由左至右遍历寻找的，这就导致如果以‘,$’和‘\n’为分隔符来截取字符串的时候，会将 rs 与 rt 一起截取出来，导致 bin1 函数内的 index 函数无法正确寻找到指定下标，从而使程序错误。为了解决这样一个问题，我们决定在 MIPS 上去做一些改变，当我们输入 MIPS 指令时，总会在第二个逗号之后紧跟一个空格，这样就能够区分’,$’和’, $’了。除了这个问题以外 findname 还会给我们带来另一个问题，由于提取最后一个字符串的时候我们都是采用了’\n’作为其结束符，所以需要在整个程序的末尾也打一个回车符，这样才能够保证最后一行的末尾是以’\n’结束的。

经过以上步骤，这个编译器已经能够将简单的 MIPS 指令如 add 等转换为 32 位的机器码了，但是这仅仅只是这个编译器的一个很小很基础的功能。仔细研究了指令集的多个函数后发现，最难处理的一类函数就是跳转指令与分支指令，它不仅需要如同 add 等 R 型指令的译码步骤，还需要一类特殊的步骤，就是跳转到指定 label，为了记录 MIPS 程序中出现的所有 label 位，我们首先对 label 出现的地方进行了分析，发现冒号是其出现的标志，只要有冒号出现，该行就有 label 存在，于是我们先对 code.asm 文件进行第一次扫描，寻找其中的 label，并将其 label 名称和出现位置记录在相应数组内。具体程序如下：

```
fp = open("code.asm","r")
i = 0
j = 0
flag = 0
```

```
for line in fp:
    if line[0:4] == 'INT2':
        flag = 1
        j -= 1
    if line[0:4] == 'END2':
        flag = 0
        j -= 1
    place = line.find(':')
    if place >= 0:j
        templabel = line[:place]
        branch += [templabel,j]
   j += 1
fp.close()
```

有了这个过程之后我们便可以对含有 label 的函数如 beq 等进行处理了。如下列程序：

```
    if line[0:3] == "beq":
        rs = findname(' $',',',line)
        rt = findname(',$',',',line)
        label = findname(', ','\n',line)
        offset = branch[branch.index(label)+1] - i
        offset = str(offset)
        code = '000100' + bin1(rs,5) + bin1(rt,5) + bin2(offset,16)
```

以上程序中 i 为当前行数，利用第一次遍历生成的 branch 数组和 i 生成了相对位移量 offset，然后将数字 offset 转换为字符串，并利用此处新出现的 bin2 函数将其转变为指定位数的二进制。bin2 也是自己编写的函数，其作用机理与 bin1 大致相同，但 bin1 是专用于转换寄存器名称字符串的函数，而 bin2 则用于将数字字符串（如'-100'等）转换为指定位数的二进制（正数则转为二进制，负数则转为其二补码），具体函数如下所示：

```
def bin2(str,num):
    dec = string.atoi(str)
    if dec >= 0:
        return (bin(dec)[2:j]).zfill(num)
    else:
        return bin(2**num+dec)[2:j].zfill(num)
```

对于中断的处理，我们做了如下设计：

由于中断的执行块内需要将内存地址最高位置为 1，所以需要将中断跳转后所执行的那一块命令用标志位标识出来，在我们的程序中采用了这样的方法，给中断指令前加 INT1，结束后加 END1，在中断跳转后的执行块开始时加 INT2，结束时加 END2，然后通过遍历令 flag 变量在执行块内为 1，最后向 rom.v 文件内写入内存地址时进行一次判断，若 flag 为 1 则将内存地址最高位置为 1。由此可以实现以上要求。

当遇到中断和异常处理时需要做一些处理，即中断或者异常处理时会将内存地址改为其本身的 ILLOP 和 XADR 地址上去，然后在此基础上进行+4 操作，当遇到 END 时重新继续在原内存地址上去进行+4 操作，利用 tempadd 做一个暂时的存储，具体程序如下：

```
    if line[0:4] == 'INT1':
        tempadd = add
        add = ILLOP
```

```
    if line[0:3] == 'ERR':
        tempadd = add
        add = XADR
    if line[0:4] == 'END1':
        add = tempadd
```

接下来在程序的最后判断,若此时读到的指令不为中断或者异常处理则将对应的内存地址和对应的机器码写入 rom.v 文件内，同时在两个字符串间加上相应字符。

具体程序如下:

```
    if (line[0:3] != 'INT') &  (line[0:3] != 'ERR') & (line[0:3] != 'END'):
        hat = bin(add)[2:j].zfill(32)
        fo = open('rom.v','a')
        if flag == 1:j
            hat = '1' + hat[1:j]
        if udge == 1:j
            fo.write('// ' + templabel + ':\n' )
        fo.write("32'b%s: data <= 32'b" % (hat))
        fo.write(code + ';' + '// '+ line)
        add += 4
        fo.close()
```

此编译器还能实现这样的功能，支持 MIPS 代码 code.asm 中添加以'#'开头的注释，并能够在生成的 rom.v 文件内自动生成注释，标注这一行机器码所代表的 MIPS 码，方便阅读和查错，同时此编译器可以直接由 MIPS 源代码 code.asm 直接生成 cpu 可使用的 rom.v，节省了人力，正确率较高。由于采用 python 语言对程序进行编写，故此编译器在程序规模上较小，仅有两百余行，与 c++编写的编译器相比规模小了很多，且功能完备，支持 MIPS 注释，能够自动添加注释，可读性很强，编译器内部使用了几个函数，识别每个命令的过程基本一致，可读性很强，遇到错误容易找出问题。这一切都源自于 python 强大的字符串处理功能和多样化的功能函数。

现对如下所示的 code.asm 进行处理。

```
lui $s5,16384
addi $t0,$zero, 1
add $t1,$t0, $t0
sub $s0,$t1, $t0
and $t2,$t0, $s0
sll $t2,$t2, 2
srl $t2,$t2, 1
bne $t1,$s0, Loop1
Loop1:add $t1,$t1, $t0
addi $a0,$zero, 3
jal sum
Loop:beq $zero,$zero, Loop
sum:addi $sp,$sp, -8
sw $ra,4($sp)
sw $a0,0($sp)
```

```
slti $t0,$a0, 1
beq $t0,$zero, L1
xor $v0,$zero, $zero
addi $sp,$sp, 8
jr $ra
L1:addi $a0,$a0, -1
jal sum
lw $a0,0($sp)
lw $ra,4($sp)
addi $sp,$sp, 8
add $v0,$a0, $v0
jr $ra
```

得到的 rom.v 如下所示：

```
`timescale 1ns/1ps
module ROM (addr,data);
input [31:0] addr;
output [31:0] data;
reg [31:0] data;
localparam ROM_SIZE = 32;
reg [31:0] ROM_DATA[ROM_SIZE-1:0];
always@(*)
   case(addr)  //Address Must Be Word Aligned.
32'b00000000000000000000000000000000: data <=
32'b00111100000101010100000000000000;// lui $s5,16384
32'b00000000000000000000000000000100: data <=
32'b00100000000010000000000000000001;// addi $t0,$zero, 1
32'b00000000000000000000000000001000: data <=
32'b00000001000010000100100000100000;// add $t1,$t0, $t0
32'b00000000000000000000000000001100: data <=
32'b00000001001010001000000000100010;// sub $s0,$t1, $t0
32'b00000000000000000000000000010000: data <=
32'b00000001000100000101000000100100;// and $t2,$t0, $s0
32'b00000000000000000000000000010100: data <=
32'b00000000000010100101000010000000;// sll $t2,$t2, 2
32'b00000000000000000000000000011000: data <=
32'b00000000000010100101000001000010;// srl $t2,$t2, 1
32'b00000000000000000000000000011100: data <=
32'b00010101001100000000000000000000;// bne $t1,$s0, Loop1
// Loop1:
32'b00000000000000000000000000100000: data <=
32'b00000001001010000100100000100000;// add $t1,$t1, $t0
32'b00000000000000000000000000100100: data <=
32'b00100000000001000000000000000011;// addi $a0,$zero, 3
```

```
32'b00000000000000000000000000101000: data <=
32'b00001100000000000000000000001100;// jal sum
// Loop:
32'b00000000000000000000000000101100: data <=
32'b00010000000000001111111111111111;// beq $zero,$zero, Loop
// sum:
32'b00000000000000000000000000110000: data <=
32'b00100011101111011111111111111000;// addi $sp,$sp, -8
32'b00000000000000000000000000110100: data <=
32'b10101111101111110000000000000100;// sw $ra,4($sp)
32'b00000000000000000000000000111000: data <=
32'b10101111101001000000000000000000;// sw $a0,0($sp)
32'b00000000000000000000000000111100: data <=
32'b00101000100010000000000000000001;// slti $t0,$a0, 1
32'b00000000000000000000000001000000: data <=
32'b00010001000000000000000000000011;// beq $t0,$zero, L1
32'b00000000000000000000000001000100: data <=
32'b00000000000000000001000000100110;// xor $v0,$zero, $zero
32'b00000000000000000000000001001000: data <=
32'b00100011101111010000000000001000;// addi $sp,$sp, 8
32'b00000000000000000000000001001100: data <=
32'b00000011111100000000000000001000;// jr $ra
// L1:
32'b00000000000000000000000001010000: data <=
32'b00100000100001001111111111111111;// addi $a0,$a0, -1
32'b00000000000000000000000001010100: data <=
32'b00001100000000000000000000001100;// al sum
32'b00000000000000000000000001011000: data <=
32'b10001111101001000000000000000000;// lw $a0,0($sp)
32'b00000000000000000000000001011100: data <=
32'b10001111101111110000000000000100;// lw $ra,4($sp)
32'b00000000000000000000000001100000: data <=
32'b00100011101111010000000000001000;// addi $sp,$sp, 8
32'b00000000000000000000000001100100: data <=
32'b00000000100000100001000000100000;// add $v0,$a0, $v0
32'b00000000000000000000000001101000: data <=
32'b00000011111100000000000000001000;// jr $ra

      default:    data <= 32'h0000_0008;
endcase
endmodule
```

可见，编译器很好的完成了其工作，并且由于注释的加入，使得原本难以辨识的机器码和内存地址拥有了较好的可读性。

# 六、 实验总结—魏齐辉

这次实验是一次完整的大型的数字逻辑设计工程，工作量非常的大，内部的模块非常的多，我们小组主力是聂浩同学，身为系科协的硬件部部长的他很好地发挥了领导作用以及充分利用自身的知识技能，我们组在他的带领下脚踏实地，一点一点地攻克遇到的问题，我们总结了一下几点教训与经验：由于工程较大，需要多人协作，分工很重要，同时每人负责的相应模块接口要一致，这就需要协调以及及时的反馈；文档较多，对于各个版本的文档进行适当的版本管理也变得很重要，中间我们就出现过改过的程序忘记保存或者修改后无法退回之前代码的问题，这真的很让人无奈，虽然期间利用 git 进行了版本管理，但效果不是特别好。

通过这次作业，我们的综合与协作能力都有了很大的进步与提高，对 CPU 也有了更深层次的理解与掌握，对流水线有了更加深刻的认识，王璞瑞同学自学 python 编写了编译器，用 200 多行就完成了该工作，很大程度上提高了效率。

遗憾的是，由于本次实验的时间较短，流水线的阻塞问题解决的并不完美，但使得我们对于流水线时序有了深刻的理解。

在实验中我们遇到了不少问题，但我们终能逐一解决，这离不开我们组内成员的相互激励，一起刷夜、一起吃外卖，这期间的汗水与欢乐都令人难忘。最后感谢助教和老师的辛勤劳动。