数字逻辑与处理器基础实验
32 位 MIPS 处理器设计

漆耘含[*]
徐远帆[†]
刘树清[‡]

2018 年 8 月 8 日

[*]无 63 2016011058
[†]无 63 2016011054
[‡]无 63 2016011041

目录

1 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

2 实验原理

2.1 32 位 ALU

ALU 主要由两块构成，一个是用来实现进位加法的加法器，另一个是根据 ALUFunc 决定具体执行哪种运算的控制部分。

2.1.1 加法器

加法器接收进行运算的两个 32 位二进制数以及是否为有符号数的标志，输出运算结果以及 Z(结果为零)、V(结果溢出)、N(结果为负) 等标志位：

```
input [31:0] A,B;
input Sign;
output [31:0] R;
output Z,V,N;
```

在运算过程中，先用一个 33 位的 wire 型变量储存计算结果，显然前 32 位就是最终加法器得到的结果。

```
wire [32:0] temp;
temp <= A + B;
R <= temp[31:0];
```

对于溢出的判断，如果是有符号数，正负相加不可能溢出，正正和负负相加，如果符号位改变则溢出。对于结果为 0 的判断，可直接验证结果。对于结果为负的判断，也可直接验证结果符号位。

2.1.2 控制部分

控制部分的核心就是根据 ALUFun 来给最终结果指定不同的运算方式，但是 ALU 中需要进行所有类别的运算，只不过最后赋给结果的 wire 型变量不同，如下：

```
assign R = (ALUFun[5:4]==2'b00)?ZA: //ADD/SUB
          (ALUFun[5:4]==2'b11)?ZC: //CMP
          (ALUFun[5:4]==2'b01)?ZL:ZS; //Logic:Shift
```

1. ADD/SUB 操作的结果只需要直接实例化一个 Adder 即可。

2. CMP 操作需要根据 Adder 计算出的 Z、N 以及输入的二进制数来决定 0 或 1。

3. Shift 操作可以利用 verilog 中的 » 和 « 来实现，利用级联来实现不同位数的移动。

4. Logic 操作可直接利用逻辑运算符来实现。

2.2 单周期 MIPS 处理器

单周期处理器一共有 7 个模块, 下面依次进行原理分析:

2.2.1 InstructionMemory

在每一次 CPU 时钟上升沿到来的时候，InstructionMemory 读进 32 位 PC，输出 32 位的指令(instruction)。其内部原理，是通过输入的 PC[9:2] 来寻址，读取相应内存中的指令。

在每一次取指模块触发之前，应更新 PC。PC 的更新方式由一个多路选择器来判断，如下表:

PCSrc(控制信号)	PC	PC 的解释
000	PC_plus_4	PC+4
001	Branch_target	分支地址
010	Jump_target	跳转地址
011	PDatabus1	Jr 指令的跳转地址
100	ILLOP	中断跳转地址
101	XAND	异常跳转地址

InstructionMemory 模块关键代码:

```
module InstructionMemory(Address, data);
input [31:0] Address;
output reg [31:0] data;
reg [31:0] ROM[31:0]; //指令存储器的大小为2^32
always @(*)
case (Address[9:2])
0: data <=32'h08000003;
1: data <=32'h08000039;
2: data <=32'h08000038;
3: data <=32'h00008020;
4: data <=32'h3c104000;
5: data <=32'h22100018;
6: data <=32'h00008820;
...
```

2.2.2 Control

控制模块的设计由图1决定。

	PCSrc[1:0]	RegWr	RegDst[1:0]	MemRd	MemWr	MemtoReg[1:0]	ALUSrc1	ALUSrc2	ExtOp	LuOp	ALUFun[5:0]	Sign
lw	0	1	1	1	0	1	0	1	1	0	0	1
sw	0	0	x	0	1	x	0	1	1	0	0	1
lui	0	1	1	0	0	0	x	1	x	1	11110	1
add	0	1	0	0	0	0	0	0	x	x	0	1
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	1	0
addi	0	1	1	0	0	0	0	1	1	0	0	1
addiu	0	1	1	0	0	0	0	1	1	0	0	0
and	0	1	0	0	0	0	0	0	x	x	11000	1
or	0	1	0	0	0	0	0	0	x	x	11000	1
xor	0	1	0	0	0	0	0	0	x	x	10110	1
nor	0	1	0	0	0	0	0	0	x	x	10001	1
andi	0	1	1	0	0	0	0	1	0	0	11000	1
sll	0	1	0	0	0	0	1	0	x	x	100000	1
srl	0	1	0	0	0	0	1	0	x	x	100001	1
sra	0	1	0	0	0	0	1	0	x	x	100011	1
slt	0	1	0	0	0	0	0	0	x	x	110101	1
slti	0	1	1	0	0	0	0	1	1	0	110101	1
sltiu	0	1	1	0	0	0	0	1	1	0	110101	0
beq	1	0	X	0	0	x	0	0	1	0	110011	1
bne	1	0	X	0	0	X	0	0	1	0	110001	1
blez	1	0	X	0	0	X	0	0	1	0	111101	1
bgtz	1	0	X	0	0	X	0	0	1	0	111111	1
bltz	1	0	X	0	0	x	0	0	1	0	111011	1
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	0	x	x	x	11010	1
jalr	3	1	1	0	0	2	0	x	x	x	11010	1
中断	4	?	3	0	0	2	0	0	X	X	X	X
异常	5	?	3	0	0	2	0	0	x	x	X	X

图 1: 控制信号表

控制模块核心代码:

```
module control(Instruct,IRQ,PCSrc,RegDst,RegWr,ALUSrc1,ALUSrc2,ALUFun,MemWr,MemRd,
    MemToReg,EXTOp,LUOp,Sign,interrupt,jiandu);
input [31:0] Instruct;
input IRQ;
input jiandu;
output [2:0] PCSrc;
output [1:0] RegDst;
output RegWr,ALUSrc1,ALUSrc2,Sign,MemWr,MemRd;
output [5:0] ALUFun;
output [1:0] MemToReg;
```

```
output EXTOp;
output LUOp;
output interrupt;

wire [2:0] PCSrc;
wire [1:0] RegDst;
wire RegWr,ALUSrc1,ALUSrc2,Sign,MemWr,MemRd;
wire [5:0] ALUFun;
wire [1:0] MemToReg;
wire EXTOp;
wire LUOp;
wire interrupt;
wire [5:0] OpCode;
wire [5:0] Funct;
wire error;

assign OpCode=Instruct[31:26];
assign Funct=Instruct[5:0];
assign interrupt=(~jiandu)&&IRQ;

assign error=~(((OpCode>=6'h01&&OpCode<=6'h0c)||(OpCode==6'h0f)||(OpCode==6'h23)
    ||(OpCode==6'h2b))||
((OpCode==6'h0)&&((Funct>=6'h20&&Funct<=6'h27)||(Funct==6'h00)||(Funct==6'h02)||(
    Funct==6'h03)||
(Funct==6'h2a)||(Funct==6'h08)||(Funct==6'h09))));
...
```

2.2.3 Register File

寄存器堆是 32 个 32 位寄存器构成的，其中 $zero 始终等于 0。

寄存器堆接受 control 模块传过来的 RegDst 和 RegWr，控制着寄存器堆的读写使能以及控制写入寄存器号。输出 DatabusA 是寄存器号 Rs 对应的 32 位数据，输出 DatabusB 是寄存器号 Rt 对应的 32 位数据，写入的寄存器的输入是 MemToReg 控制的输出数据，写入寄存器号由四路选择器控制，如下：

RegDst	写入寄存器号
00	Rd=Instryction[15:11]
01	Rt=Instruction[20:16]
10	Ra=31
11	Xp=26

RegisterFile 模块核心代码：

```
module RegisterFile(clk,reset,RegWr,regA,regB,read_dataA,read_dataB,write_data,
    write_reg);
input clk,reset;
input RegWr;
input [4:0] regA , regB, write_reg;
input [31:0] write_data;
output [31:0]read_dataA , read_dataB;

reg [31:0] RF_DATA[31:1]; //确定输出，注意$zero
assign read_dataA = (regA == 5'b00000) ? 32'b0 : RF_DATA[regA];
assign read_dataB = (regB == 5'b00000) ? 32'b0 : RF_DATA[regB];

integer i;

always @ (posedge clk or negedge reset)
begin
if(~reset)
for (i = 1; i < 32; i = i + 1)
RF_DATA[i] <= 32'h00000000;
else
if(RegWr == 1 && write_reg != 5'b00000)
RF_DATA[write_reg] <= write_data; //写入寄存器

end
endmodule // RegisterFile
```

2.2.4 ALU

ALU 模块有两个输入 ALU_A 和 ALU_B，以及控制信号 ALUFun 和 Sign，输出 32 位的数据 ALU_out。

输入数据 ALU_A 和 ALU_B 分别由控制信号 ALUSrc1 和 ALUSrc2 来进行控制，如下表：

ALUSrc1	ALU_A	ALUSrc2	ALU_B
0	Shamt	0	DataBusB
1	DataBusA	1	Lu_out

其中：

```
Shamt={27'd0, Instruction[10:6]}
```

Lu_out 由 LU_out 控制，输出是扩展后数据 Ext_out 或者 lui 操作之后的数据，如下表：

ExtOp	Ext_out	LuOp	Lu_out
0	{16'h0000, Instruction[15:0]}	0	Ext_out
1	{16{Instruction[15]},Instruction[15:0]}	1	{Instruction[15:0], 16'h0000}

同时，在得到 Ext_out 之后，通过左移两位并与 PC+4 相加，得到分支地址 ConBA。

ALU 内部原理同第一题中 ALU 的设计

2.2.5 DataMemory

数据存储器（DataMemory）接收信号 MemRead 和 MemWrite 信号，但真正控制存储器的读取信号，是 MemRead 和 MemWrite 分别和 ALU_out[30] 做与操作，因为数据存储器访问的地址是 0x00000000-0x39999999，因此 32 位地址的第 31 位为 0. 而外设存储器（peripheral）的地址是 0x4000000-0x79999999，32 位地址的第 31 位为 1。因为 MemRead 和 MemWrite 会同时输入数据存储器和外设存储器，因此需要用 ALU_out[30] 来进行区分。

数据存储器的地址输入是 ALU_out，写入的数据是 Databus2，读取出的数据为 Read_datam。

DataMemory 模块核心代码：

```
module DataMemory(reset, clk, Address, Write_data, Read_data, MemRead, MemWrite);
input reset, clk;
input [31:0] Address, Write_data;
input MemRead, MemWrite;
output [31:0] Read_data;

parameter RAM_SIZE = 256;
parameter RAM_SIZE_BIT = 8;

reg [31:0] RAM_data[RAM_SIZE - 1: 0];

assign Read_data = MemRead? RAM_data[Address[RAM_SIZE_BIT + 1:2]]: 32'h00000000;

integer i;

always @(posedge reset or posedge clk)
```

```
if (reset)
for (i = 0; i < RAM_SIZE; i = i + 1)
RAM_data[i] <= 32'h00000000;
else if (MemWrite)
RAM_data[Address[RAM_SIZE_BIT + 1:2]] <= Write_data;
endmodule
```

2.2.6 Peripheral

外设存储器的输入控制信号如 DataMemory 中所述，控制信号为 MemRead 和 MemWrite 分别和 AlU_out[30] 做与操作。

外设存储器除了有存储的功能，还有串口接收发送的功能。具体的存储如图2：

地址范围（字节功能）	功能	描述
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，0-disable 1bit：定时器中断控制，1-enable，0-disable 2bit：定时器中断状态
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	串口发送数据 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 UART_RXD	串口接收数据寄存器，只有低 8bit 有效
0x40000020	串口状态、控制 UART_CON	0bit：发送中断使能，1-enable，0-disable 1bit：接收中断使能，1-enable，0-disable 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 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

图 2: 外设功能表

对于中断，外设中有一个定时器，在开始的时候，需要人为在汇编程序中对 TCON 进行操作，刚开始的时候，TCON=0，不允许计数，要启动定时器，需通过 sw 操作在 0x40000008 写入 3（3’ b011），使其开始计数。计数 TL 开始增加之后，当 TL 全为 1，程序需进入中断状态，外设的 irqout=1，输

入到 control 模块中使 interrrupt 控制信号 =1，改变 PCSrc 控制信号，使下一次的 PC=ILLOP，程序进入中断处理程序，同时在寄存器堆的 $ra 中保存中断位置的 PC，在中断程序执行完成之后，进行 jr $ra 指令，即返回到中断处继续执行主程序。

对于串口收发模块，基本是沿用的春季学期第四次数逻实验串口代码，不过有所修改。对于接收模块，原理是当接收到一位低电平时，开始计数，采样八位后停止，RX_STATUS 变为 1，这时，在外设模块中有一个标志位 RX_GET，当 RX_STATUS 变为 1，即接收完成之后，把 RX_GET 置位 1，同时把接收到的数据保存在 RX_DATA1 中，因为汇编代码中要求接收两个数，因此在第一次接收完成，同时把数据保存在内存中的时候，把 RX_GET 置 0，允许下一次计数。接收模块需要结合汇编代码来一起实现，在汇编代码中采用循环的方式，不断地访问外设存储器，看是否接收到数据，当没有接收到数据的时候，则一直循环，如果接收到数据，则跳出循环。

对于串口发送模块，当访问外设存储器 0x40000018 位置的时候，标志着已经计算完成，将计算结果保存在该地址，同时允许发送，将 TX_EN 置位 1，这时串口发送模块将数据发送。

同时外设模块也会输出数据：Read_datap，在确定访问存储器输出 Read_data 的时候，是用一个二路选择器来实现的，如下表所示：

ALU_out[30]	Read_data
0	Read_datam
1	Read_datap

除了这六个主要的模块，还有一点设计没有提及，即寄存器写回数据的判断。写回寄存器的数据是由 MemToReg 控制信号控制，具体如下表：

MemToReg	Data
0	ALU_out
1	Read_data
2	PC+4

外设部分核心输入输出：

```
`timescale 1ns/1ps

module Peripheral (reset,sys_clk,clk,rd,wr,addr,wdata,rdata,led,digi,irqout,
    UART_TX,UART_RX);
input reset,clk,sys_clk;
input rd,wr;
input UART_RX;
input [31:0] addr;
input [31:0] wdata;
output UART_TX;
output [31:0] rdata;
output [7:0] led;//LED信号
output [11:0] digi;//七段译码管
```

```
output irqout;//定时器是否是中断状态

reg [31:0] rdata;
reg [7:0] led;
reg [11:0] digi;
reg [31:0] TH,TL;
reg [2:0] TCON;
reg [7:0] RX_DATA1;

assign irqout = TCON[2];

wire [7:0]RX_DATA;
wire RX_STATUS;
reg [7:0]TX_DATA;
reg TX_EN;
wire TX_STATUS;
reg RX_GET;
```

2.2.7 Top(顶层架构)

顶层架构负责将各个模块连接起来，构成一个有机整体。具体见代码注释:

```
module CPU(reset, sys_clk, digi_o1, digi_o2, digi_o3, digi_o4, led, rx, tx);
input reset, sys_clk;
input rx;
output [6:0] digi_o1;
output [6:0] digi_o2;
output [6:0] digi_o3;
output [6:0] digi_o4;
output [7:0] led;
output tx;

//分频
divide_clk div_clk(.reset(reset), .sys_clk(sys_clk), .clk(clk));

//PC+4
reg [31:0] PC;
wire [31:0] PC_next;
always @(negedge reset or posedge clk)
```

```
if (~reset)
PC <= 32'h0000000;
else
PC <= PC_next;

wire [31:0] PC_plus_4;

assign PC_plus_4 = PC + 32'd4;

//取指令
wire [31:0] Instruction;
InstructionMemory instruction_memory1(.Address({1'b0,PC[30:0]}), .data(Instruction
    ));

//控制信号
wire [2:0] PCSrc;
wire [1:0] RegDst;
wire RegWr;
wire ALUSrc1;
wire ALUSrc2;
wire [5:0] ALUFun;
wire Sign;
wire MemWrite;
wire MemRead;
wire [1:0] MemtoReg;
wire ExtOp;
wire LuOp;
wire irqout;
wire interrupt;

control control1(.Instruct(Instruction), .IRQ(irqout),.PCSrc(PCSrc), .RegDst(
    RegDst), .RegWr(RegWr),
.ALUSrc1(ALUSrc1), .ALUSrc2(ALUSrc2), .ALUFun(ALUFun),
.MemWr(MemWrite), .MemRd(MemRead), .MemToReg(MemtoReg),
.EXTOp(ExtOp), .LUOp(LuOp),
.Sign(Sign), .interrupt(interrupt),.jiandu(PC[31]));

//寄存器堆
```

```
wire [31:0] Databus1, Databus2, Databus3;
wire [4:0] Write_register;
parameter Xp = 5'b11010;
parameter Ra = 5'b11111;
assign Write_register = (RegDst == 2'b00)? Instruction[15:11]: (RegDst == 2'b01)?
    Instruction[20:16]: (RegDst == 2'b10)? Ra: Xp;

RegisterFile register_file1(.clk(clk), .reset(reset), .RegWr(RegWr),
.regA(Instruction[25:21]), .regB(Instruction[20:16]), .read_dataA(Databus1), .
    read_dataB(Databus2), .write_data(Databus3),.write_reg(Write_register));

//扩展方式
wire [31:0] Ext_out;
assign Ext_out = {ExtOp? {16{Instruction[15]}}: 16'h0000, Instruction[15:0]};

//Lui or Ext
wire [31:0] LU_out;
assign LU_out = LuOp? {Instruction[15:0], 16'h0000}: Ext_out;

//ALU模块
wire [31:0] ALU_a;
wire [31:0] ALU_b;
wire [31:0] ALU_out;

assign ALU_a = ALUSrc1? {27'd0, Instruction[10:6]}: Databus1;
assign ALU_b = ALUSrc2? LU_out: Databus2;

ALU alu1(.A(ALU_a), .B(ALU_b), .ALUFun(ALUFun), .Sign(Sign), .R(ALU_out));

//读取数据寄存器 or 外设
wire [31:0] Read_datam;
wire [31:0] Read_datap;
wire [31:0] Read_data;
wire [11:0] digi;
//如果ALU_out[30]==0, 这说明是访问的数据段

DataMemory data_memory1(.reset(reset), .clk(clk), .Address(ALU_out), .Write_data(
    Databus2),
```

```
.Read_data(Read_datam), .MemRead(MemRead & (~ALU_out[30])), .MemWrite(MemWrite &
    (~ALU_out[30])));

//如果ALU_out[30]==1, 这说明是访问的外设段
Peripheral peripheral1(.reset(reset), .sys_clk(sys_clk), .clk(clk), .rd(MemRead &
    (ALU_out[30])), .wr(MemWrite & (ALU_out[30])),.addr(ALU_out),
.wdata(Databus2), .rdata(Read_datap), .led(led),
.digi(digi), .irqout(irqout), .UART_TX(tx), .UART_RX(rx));

//显示模块
digitube_scan digi_scan(.digi_in(digi), .digi_out1(digi_o1), .digi_out2(digi_o2),
    .digi_out3(digi_o3), .digi_out4(digi_o4));
assign Read_data = ALU_out[30]? Read_datap: Read_datam;
assign Databus3 = (MemtoReg == 2'b00)? ALU_out: (MemtoReg == 2'b01)? Read_data: ((
    MemtoReg == 2'b10) && (interrupt))?PC:PC_plus_4;

wire [31:0] Jump_target;
assign Jump_target = {PC_plus_4[31:28], Instruction[25:0], 2'b00};
wire [31:0] Branch_target;
assign Branch_target = ALU_out[0]? PC_plus_4 + {LU_out[29:0], 2'b00}: PC_plus_4;

parameter ILLOP = 32'h80000004;
parameter XADR = 32'h80000008;
assign PC_next = (PCSrc == 3'b000)? PC_plus_4: (PCSrc == 3'b001)? Branch_target: (
    PCSrc == 3'b010)? Jump_target:
(PCSrc == 3'b011)? Databus1: (PCSrc == 3'b100)? ILLOP: XADR;

endmodule
```

2.3 流水线 CPU

流水线的各个模块和单周期 CPU 的模块内部原理都是一模一样的，主要的变化是顶层模块 CPU.v，下面详细说一下设计。

流水线主要由 5 个阶段：IF、ID、EX、MEM、WB。因为每一个模块都在不停地工作，因此需要在每一个模块之后添加大量的寄存器来保存上一周期的结果，同时作为下一周期下一模块的输入。在整个顶层模块 CPU.v 中，用了多了并联形式的 always 模块，在 CPU 时钟上升沿触发，更新寄存器的值。

2.3.1 IF 阶段

IF 阶段是取指阶段，将 PC 送入 InstructionMemory 中，得到指令 instruction。

对 PC 来说，基本的思路和单周期相类似，不过有一点不同。流水线特有的是阻塞，即下一条指令需要用到上一条指令的运算结果，不过不能通过转发来实现，只能通过阻塞一个周期来实现。因此，PC 的值如下表：

控制信号	PC
reset	32'd0
PCSrc==3' d4 && is_branch	ILLOP
PCSrc==3' d5 && is_branch	XADR
load_use && is_branch	ID_EX_CONBA
PCSrc==3' d2 && load_use	JUMP_target
PCSrc==3' d3 && load_use	Databus1
else	PC_plus_4

注：is_branch 是控制是否分支跳转，load_sue 是控制是否阻塞，当阻塞的时候，PC 的值不变。

当输入 PC，得到 instruction 之后，需要对 instruction 进行处理。因为当中断或者异常的时候，需要把 instruction 清 0，当是 j 型指令或者分支指令的时候，也需要把 instruction 清 0。具体的代码实现见后。根据流水线的特性，如果不把 instruction 清零，即是 PC 改变了。后面也还是会执行下一条指令，故此必须要把 instruction 清零。

2.3.2 ID 阶段

该阶段为解指阶段，思路和单周期差不多，只不过是输入变成了从 IF_ID 阶段的寄存器中的值。

寄存器的写入数据和写入寄存器号是从 WB 阶段传回来的，因为只有 WB 阶段的才是真正该写入寄存器堆的，否则当前的 write_register 是下下下条指令的寄存器号。

同时在该阶段需要进行转发，转发一共有三种情况，是通过后面阶段的控制信号和寄存器号的比较来进行控制的，具体代码实现见后。

以及考虑到阻塞操作，如果阻塞，是需要把所有控制信号清 0 的。否则后面的阶段会进行操作。

2.3.3 EX 阶段

该阶段为 ALU 计算阶段，主要思路和单周期一样，不过进行操作的数据变为了 ID_EX 阶段寄存器中的数据。

在得到 ALU_out 之后，根据输出计算 is_branch，决定是否进行分支操作。

2.3.4 MEM

这是进行访问存储器的操作，同样分为访问数据存储器和外设存储器，功能原理和单周期一模一样。

2.3.5 WB

这是写回寄存器堆操作，同样的，写回数据是由 MEM_WB_MemToReg 控制信号来决定，这个信号是从 MemToReg 一级一级地传递过来的。写回的寄存器号也是从 ID 阶段传递过来的。

2.3.6 核心代码展示

其余模块与单周期相同，这里只展示顶层架构部分。

```
module CPU(reset, sys_clk, digi_o1, digi_o2, digi_o3, digi_o4, led, rx, tx);
input reset, sys_clk;
input rx;

output [6:0] digi_o1;
output [6:0] digi_o2;
output [6:0] digi_o3;
output [6:0] digi_o4;
output [7:0] led;
output tx;

//分频
divide_clk div_clk(.reset(reset), .sys_clk(sys_clk), .clk(clk));

//PC+4
reg [31:0] PC;
wire [31:0] PC_next;
wire load_use;
wire is_branch;
wire is_jump;

//IF阶段的信号
wire is_ILLOP;
wire is_XADR;
reg [31:0] IF_ID_instruction;
reg [31:0] IF_ID_PC;
reg [31:0] IF_ID_PC4;
reg IF_ID_is_branch;

//ID_EX之间的寄存器
reg ID_EX_is_jump; //保存经过control运算 (PCSrc) 传过来的jump
```

```
reg [31:0] ID_EX_JUMP_Addr;//跳转地址
reg [31:0] ID_EX_PC4;//保存PC4（注意中断和跳转的地址会保存在这里面去）
reg [4:0] ID_EX_Rt;//保存Rt
reg [4:0] ID_EX_Shamt;//保存Shamt
reg ID_EX_MemRd;//保存control出来的MemRd
reg ID_EX_MemWr;//保存control出来的MemWr
reg [1:0] ID_EX_MemToReg;//保存control出来的MemToReg
reg ID_EX_RegWr;//保存control出来的RegWr
reg [2:0] ID_EX_PCSrc;//保存control出来的PCSrc
reg ID_EX_Sign;//保存control出来的Sign
reg [5:0] ID_EX_ALUFun;//保存control出来的ALUFun
reg ID_EX_ALUSrc1;//保存control出来的MemWr
reg ID_EX_ALUSrc2;//保存control出来的MemWr
reg [4:0] ID_EX_Write_reg;//保存要写的寄存器号
reg [31:0] ID_EX_data1;//转发后的输出1
reg [31:0] ID_EX_data2;//转发后的输出2
reg [31:0] ID_EX_Imm;//经过扩展和lu选择之后的输出
reg [31:0] ID_EX_CONBA;//保存分支地址

//EX_MEM之间的寄存器，一部分是保存上一级ID_EX的控制信号
reg [31:0] EX_MEM_PC4;
reg EX_MEM_MemRd;
reg EX_MEM_MemWr;
reg [1:0] EX_MEM_MemToReg;
reg EX_MEM_RegWr;
reg [4:0] EX_MEM_Write_reg;//从ID_EX传递过来的要保存的寄存器号
reg [31:0] EX_MEM_ALU_out;//ALU的输出
reg [31:0] EX_MEM_data2;//MEM保存的数据

//MEM_WB之间的寄存器
reg MEM_WB_RegWr;
reg [4:0] MEM_WB_Write_reg;
reg [31:0] MEM_WB_dataout;//写回寄存器的值
wire [4:0] Rs;
wire [4:0] Rt;
wire [4:0] Shamt;
wire [4:0] Rd;
wire [25:0] JT;
```

```
wire [15:0] Imm16;

//控制信号
wire [2:0] PCSrc;
wire [1:0] RegDst;
wire RegWr;
wire ALUSrc1;
wire ALUSrc2;
wire [5:0] ALUFun;
wire Sign;
wire MemWrite;
wire MemRead;
wire [1:0] MemToReg;
wire ExtOp;
wire LuOp;
wire irqout;
wire interrupt;

//读取数据寄存器 or 外设
wire [31:0] Read_datam;
wire [31:0] Read_datap;
wire [31:0] Read_data;
wire [11:0] digi;
wire [31:0] Databus3;

//寄存器堆
wire [31:0] Databus1, Databus2;
wire [4:0] Write_register;

//ALU模块
wire [31:0] ALU_a;
wire [31:0] ALU_b;
wire [31:0] ALU_out;

//跳转地址
wire [31:0] Jump_target;
assign Jump_target = {IF_ID_PC4[31:28],IF_ID_instruction[25:0],2'd0};
```

```
//PC+4
wire [31:0] PC_plus_4;
assign PC_plus_4 = PC + 32'd4;

//阻塞的控制信号，即lw保存的地址等于下一条指令要进入ALU的寄存器，就阻塞（PC不变）
assign load_use =(ID_EX_MemRd && (ID_EX_Rt == Rs || ID_EX_Rt == Rt));
assign is_jump=(PCSrc==3'd2);
assign is_ILLOP=(PCSrc==3'd4);
assign is_XADR=(PCSrc==3'd5);

parameter ILLOP = 32'h80000004;
parameter XADR = 32'h80000008;

always @(negedge reset or posedge clk)
begin
if (~reset)
PC <= 32'd0;
else if (is_ILLOP && ~is_branch)//中断
PC <= ILLOP;
else if (is_XADR && ~is_branch)//异常
PC <= XADR;
else if (~load_use)//非阻塞运行，如果是阻塞，则PC不变
begin
if (is_branch)//分支
PC <= ID_EX_CONBA;
else if (PCSrc == 3'd2)//j
PC <= Jump_target;
else if (PCSrc == 3'd3)//jr
PC <= Databus1;
else//正常流程
PC <= PC_plus_4;
end
end

//取指令
wire [31:0] Instruction;
InstructionMemory instruction_memory1(.Address({1'b0,PC[30:0]}), .data(Instruction
    ));
```

```
//IF-ID，相当于是IF取完指，存入IF-ID中间的寄存器中保存
always @(posedge clk or negedge reset)
begin
if(~reset)
begin
IF_ID_instruction <= 32'd0;
IF_ID_PC4 <= 32'd0;
IF_ID_PC <= 32'd0;
end
else if (PCSrc == 3'd4 || PCSrc == 3'd5)//如果是中断或者异常，下一条指令要清0，即不
    执行
begin
IF_ID_PC <= 32'h80000000;
IF_ID_PC4 <= 32'h80000000;
IF_ID_instruction <= 32'h00000000;
end
else if (is_branch || (PCSrc == 3'd2) || (PCSrc == 3'd3))//如果是分支、j型指令，下
    一条指令清0，不执行
begin
IF_ID_PC <= IF_ID_PC[31]? 32'h80000000:32'd0;
IF_ID_PC4 <= IF_ID_PC4[31]?32'h80000000:32'd0;
IF_ID_instruction <= 32'd0;
end
else if(~load_use)//非阻塞，正常运行，阻塞，下一条指令和PC和当前指令与PC一样
begin
IF_ID_PC <= PC;
IF_ID_PC4 <= PC_plus_4;
IF_ID_instruction <= Instruction;
end
end

always@ (posedge clk or negedge reset)
begin
if(~reset)
IF_ID_is_branch <= 0;
else
IF_ID_is_branch <= is_branch;//把分支跳转指令保存到下一级里面去
```

```
end

//取指
assign Rs = IF_ID_instruction[25:21];
assign Rt = IF_ID_instruction[20:16];
assign Shamt = IF_ID_instruction[10:6];
assign Rd = IF_ID_instruction[15:11];
assign JT = IF_ID_instruction[25:0];
assign Imm16 = IF_ID_instruction[15:0];

//产生控制信号，注意，这里的输入的指令应该是IF阶段的指令，监督位也应该是IF阶段PC4的最
    高位
control control1(.Instruct(IF_ID_instruction), .IRQ(irqout),.PCSrc(PCSrc), .RegDst
    (RegDst), .RegWr(RegWr),.ALUSrc1(ALUSrc1), .ALUSrc2(ALUSrc2), .ALUFun(ALUFun),.
    MemWr(MemWrite), .MemRd(MemRead), .MemToReg(MemToReg),.EXTOp(ExtOp), .LUOp(LuOp
    ), .Sign(Sign), .interrupt(interrupt),.jiandu(IF_ID_PC4[31]));

parameter Xp = 5'b11010;
parameter Ra = 5'b11111;

//判断是保存在哪个寄存器中，然后一级一级地传递下去，最后输入寄存器堆的是MEM_WB的
    write_reg
assign Write_register = (RegDst == 2'b00)? Rd: (RegDst == 2'b01)? Rt: (RegDst ==
    2'b10)? Ra: Xp;

//访问并写回寄存器堆
RegisterFile register_file1(.clk(clk), .reset(reset), .RegWr(MEM_WB_RegWr),
.regA(Rs), .regB(Rt), .read_dataA(Databus1), .read_dataB(Databus2), .write_data(
    MEM_WB_dataout),
.write_reg(MEM_WB_Write_reg));

//扩展方式，ExtOp=1，符号扩展，=0，无符号扩展
wire [31:0] Ext_out;
assign Ext_out = ExtOp? {{16{Imm16[15]}},Imm16}: {16'h0000, Imm16};

//Lui or Ext，LU_out=1，输出lui之后的值，=0，输出扩展的Ext_out
wire [31:0] LU_out;
assign LU_out = LuOp? {Imm16, 16'd0}: Ext_out;
```

```
//计算分支地址
wire [31:0] CONBA;
assign CONBA = IF_ID_PC4+ {Ext_out[29:0], 2'b00};

//ID-EX阶段，更新寄存器的值
always @(posedge clk or negedge reset)
begin
if(~reset)//清0
begin
ID_EX_is_jump<= 1'b0;
ID_EX_JUMP_Addr <= 32'd0;
ID_EX_PC4 <= 32'd0;
ID_EX_Rt <= 5'd0;
ID_EX_Shamt <= 5'd0;
ID_EX_MemRd <= 1'b0;
ID_EX_MemWr <= 1'b0;
ID_EX_MemToReg <= 2'd0;
ID_EX_RegWr <= 1'b0;
ID_EX_PCSrc <= 3'd0;
ID_EX_Sign <= 1'b0;
ID_EX_ALUFun <= 6'd0;
ID_EX_ALUSrc1 <= 1'b0;
ID_EX_ALUSrc2 <= 1'b0;
ID_EX_Write_reg <= 1'b0;
ID_EX_data1 <= 32'd0;
ID_EX_data2 <= 32'd0;
ID_EX_Imm <= 32'd0;
ID_EX_CONBA <= 32'd0;
ID_EX_data1 <= 32'd0;
ID_EX_data2 <= 32'd0;
end
else
begin
//判断写入寄存器的值，如果同一条指令跳转且中断，则会先跳转再中断；如果上一条指令跳转，
    当前指令中断，则把当前的PC写入寄存器
ID_EX_PC4 <= ((IF_ID_is_branch ==1) && irqout)? PC:((ID_EX_is_jump == 1)?
    ID_EX_JUMP_Addr:
```

```
((IF_ID_instruction[31:26] == 6'b000011)?IF_ID_PC4:IF_ID_PC));
ID_EX_Rt <= Rt;//用于阻塞判断
ID_EX_Shamt <= Shamt;//用于ALUA输入的一个选项
ID_EX_PCSrc <= PCSrc;//保存PCSrc，用于ALU输出之后，计算is_branch
ID_EX_Imm <= LU_out;//保存立即数输入
ID_EX_CONBA <= CONBA;//用于保存分支指令，如果is_branch==1，则会写入PC

//转发
ID_EX_data1 <=
(EX_MEM_RegWr&&(EX_MEM_Write_reg!=0)&&(EX_MEM_Write_reg==Rs)&&(ID_EX_Write_reg!=Rs
    ||~ID_EX_RegWr))?Databus3:
(ID_EX_RegWr&&(ID_EX_Write_reg!=0)&&(ID_EX_Write_reg==Rs))?ALU_out:
(MEM_WB_RegWr&&(MEM_WB_Write_reg!=0)&&(MEM_WB_Write_reg==Rs))?MEM_WB_dataout:
    Databus1;
ID_EX_data2 <=
(EX_MEM_RegWr&&(EX_MEM_Write_reg!=0)&&(EX_MEM_Write_reg==Rt)&&(ID_EX_Write_reg!=Rt
    ||~ID_EX_RegWr))?Databus3:
(ID_EX_RegWr&&(ID_EX_Write_reg!=0)&&(ID_EX_Write_reg==Rt))?ALU_out:
(MEM_WB_RegWr&&(MEM_WB_Write_reg!=0)&&(MEM_WB_Write_reg==Rt))?MEM_WB_dataout:
    Databus2;

//如果是阻塞或者分支，则这些信号清0
if ((load_use || is_branch)&&~is_XADR&&~is_ILLOP)
begin
ID_EX_Write_reg<=5'd0;
ID_EX_MemToReg<=2'd0;
ID_EX_ALUFun<=6'd0;
ID_EX_RegWr<=1'd0;
ID_EX_ALUSrc1<=1'd0;
ID_EX_ALUSrc2<=1'd0;
ID_EX_Sign<=1'd0;
ID_EX_MemWr<=1'd0;
ID_EX_MemRd<=1'd0;
ID_EX_is_jump<=1'd0;
ID_EX_JUMP_Addr<=32'b0;
end
else
begin
```

```
ID_EX_Write_reg <= Write_register;
ID_EX_MemToReg <= MemToReg;
ID_EX_ALUFun <= ALUFun;
ID_EX_ALUSrc1 <= ALUSrc1;
ID_EX_ALUSrc2 <= ALUSrc2;
ID_EX_is_jump<= is_jump;
ID_EX_MemRd <= MemRead;
ID_EX_MemWr <= MemWrite;
ID_EX_RegWr <= RegWr;
ID_EX_Sign <= Sign;
ID_EX_JUMP_Addr <= {IF_ID_PC4[31:28],IF_ID_instruction[25:0],2'd0};
end
end
end

//判断ALU的输入
assign ALU_a = ID_EX_ALUSrc1? {27'd0,ID_EX_Shamt}:ID_EX_data1;
assign ALU_b = ID_EX_ALUSrc2? ID_EX_Imm: ID_EX_data2;

//ALU模块
ALU alu1(.A(ALU_a), .B(ALU_b), .ALUFun(ID_EX_ALUFun), .Sign(ID_EX_Sign), .R(
    ALU_out));

//根据ALU输出的最低位来判断is_branch的值
assign is_branch = (ID_EX_PCSrc == 3'd1 && ALU_out[0]);

//EX_MEM
always @ (posedge clk or negedge reset)
begin
if(~reset)
begin
EX_MEM_PC4 <= 32'd0;
EX_MEM_MemRd <= 1'd0;
EX_MEM_MemWr <= 1'd0;
EX_MEM_MemToReg <= 2'd0;
EX_MEM_RegWr <= 1'd0;
EX_MEM_Write_reg <= 5'd0;
EX_MEM_ALU_out <= 32'd0;
```

```
EX_MEM_data2 <= 32'd0;
end
else
begin
EX_MEM_PC4 <= ID_EX_PC4;
EX_MEM_MemRd <= ID_EX_MemRd;
EX_MEM_MemWr <= ID_EX_MemWr;
EX_MEM_MemToReg <= ID_EX_MemToReg;
EX_MEM_RegWr <= ID_EX_RegWr;
EX_MEM_Write_reg <= ID_EX_Write_reg;
EX_MEM_ALU_out <= ALU_out;
EX_MEM_data2 <= ID_EX_data2;
end
end

//如果ALU_out[30]==0, 这说明是访问的数据段
DataMemory data_memory1(.reset(reset), .clk(clk), .Address(EX_MEM_ALU_out),
.Write_data(EX_MEM_data2), .Read_data(Read_datam),
.MemRead(EX_MEM_MemRd & (~EX_MEM_ALU_out[30])),
.MemWrite(EX_MEM_MemWr & (~EX_MEM_ALU_out[30])));

//如果ALU_out[30]==1, 这说明是访问的外设段
Peripheral peripheral1(.reset(reset), .sys_clk(sys_clk), .clk(clk), .rd(
    EX_MEM_MemRd & (EX_MEM_ALU_out[30])),
.wr(EX_MEM_MemWr & (EX_MEM_ALU_out[30])),.addr(EX_MEM_ALU_out),
.wdata(EX_MEM_data2), .rdata(Read_datap), .led(led),
.digi(digi), .irqout(irqout), .UART_TX(tx), .UART_RX(rx));

//显示模块
digitube_scan digi_scan(.digi_in(digi), .digi_out1(digi_o1), .digi_out2(digi_o2),
    .digi_out3(digi_o3), .digi_out4(digi_o4));

//根据访问地址的第二高位来判断存储输出的是数据存储器得值还是外设存储器的值
assign Read_data = EX_MEM_ALU_out[30]? Read_datap: Read_datam;
assign Databus3 = (EX_MEM_MemToReg == 2'b00)? EX_MEM_ALU_out: (EX_MEM_MemToReg ==
    2'b01)? Read_data: (EX_MEM_MemToReg == 2'b10)?EX_MEM_PC4:EX_MEM_PC4 - 32'd12;

//MEM-WB
```

```
always @(posedge clk or negedge reset)
begin
if(~reset)
begin
MEM_WB_RegWr <= 1'b0;
MEM_WB_Write_reg <= 5'd0;
MEM_WB_dataout <= 32'd0;
end
else
begin
MEM_WB_RegWr <= EX_MEM_RegWr;
MEM_WB_Write_reg <= EX_MEM_Write_reg;
MEM_WB_dataout <= Databus3;
end
end

endmodule
```

2.4 汇编器及汇编代码原理

2.4.1 汇编程序

本次实验中的汇编程序由两部分组成，一部分是从串口读入数据并运行算法计算最大公约数，另一部分是数码管的译码和显示，由于实验中定时器的中断只用来完成数码管的扫描，因此数码管的译码和显示代码就是中断处理代码。

第一部分代码首先启动定时器，而后检查串口标志位，当串口标志位有效时读串口数据。待输入的两个数据均读取完成后，计算结果并访问外设以显示结果。当完成任务后，CPU 将进入无限循环的状态以便观察结果。

第二部分的代码首先将处理与定时器相关的中断，而后保护现场，由于程序比较简单，主程序与中断处理程序未使用相同的寄存器，因此这一步在代码中未体现。中断处理代码首先检查数码管对应外设数据，并移动扫描位，而后进行软件译码，软件译码的过程即 case 块语法转换成汇编语法，较为繁琐，因此在报告中删去部分。软件译码完成后，修改数码管外设对应地址值，重新启动定时器，回到主程序继续执行。

部分关键代码见下：

```
//求最大公约数
dif:
add $t2, $a0, $0
sub $t4, $t2, $a1
```

```
blez $t4, Swap
sub $t2, $t2, $a1
Swap:
add $a0, $a1, $0
add $a1, $t2, $0
Euclidean:
bne $a0, $a1, dif
add $v0, $a0, $0

//中断显示
InterruptProcess:
addi $t5, $0, -7 #t5=0xfffffff9

add $s7, $0, $0
lui $s7, 0x4000
addi $s7, $s7, 0x0008 #s7=0x40000008
lw $t6, 0($s7)

and $t5, $t5, $t6
sw $t5, 0($s7) #Tcon&=0xfffffff9

addi $s7, $s7, 12 #s7=0x40000014

lw $t5, 0($s7) #t5=数码管
andi $s6, $t5, 0x0F00 #111100000000 s6记录数码管信息

addi $t6, $0, 0x0100 #t6=000100000000
beq $s6, $0, LabelA #AN全为0
beq $t6, $s6, LabelB #AN=0001
sll $t6, $t6, 1
beq $t6, $s6, LabelC
sll $t6, $t6, 1
beq $t6, $s6, LabelD
sll $t6, $t6, 1
beq $t6, $s6, LabelA
```

2.4.2 汇编器

汇编器使用 python 语言编写。汇编器依照以下步骤执行工作：

1. 遍历代码，计算 Label 名称对应的地址值
2. 遍历代码，将 Label 名称表示的地址转换成相对地址或绝对地址
3. 遍历代码，将汇编程序转换成机器码

匹配和替换工作主要使用正则表达式完成，核心转换代码见下：

```
def MachineCodeTranslate(assemblecode):
    try:
        a =re.match(r'(.*)[ ]+\$(.*)[ ]*,[ ]*\$(.*)[ ]*,[ ]*\$(.*)', assemblecode) #add $rd,$rs,$rt
        if a: return opcode[a.group(1)] +register_translated(a.group(3)) +register_translated(a.group(4)) +
                                                                register_translated(a.group(2)) +'00000' +funccode
                                                                [a.group(1)]
        a =re.match(r'(.*)[ ]+\$(.*)[ ]*,[ ]*\$(.*)[ ]*,[ ]*([^\$]+)', assemblecode) #addi $rt,$rs,imm
        if a: return opcode[a.group(1)] +register_translated(a.group(3)) +register_translated(a.group(2)) +
                                                                imme_translated((lambda x: int(x, 16) if x.
                                                                startswith('0x') else int(x, 10))(a.group(4)))
        a =re.match(r'(.*)[ ]+\$(.*)[ ]*,[ ]*(.*)\(\$(.*)\)', assemblecode) #lw/sw $rt,offset($rs)
        if a: return opcode[a.group(1)] +register_translated(a.group(4)) +register_translated(a.group(2)) +
                                                                imme_translated((lambda x: int(x, 16) if x.
                                                                startswith('0x') else int(x, 10))(a.group(3)))
        a =re.match(r'(sll|srl|sra)[ ]+\$(.+)[ ]*,[ ]*\$(.+)[ ]*,[ ]*(.+)', assemblecode) #sll/srl/sra $rd,$rt,shamt
        if a: return opcode[a.group(1)] +'00000' +register_translated(a.group(3)) +register_translated(a.group(2))
                                                                +sha_translate(int(a.group(4))) +funccode[a.group(
                                                                1)]
        a =re.match(r'(beq|bne)[ ]+\$(.*)[ ]*,[ ]*\$(.*)[ ]*,[ ]*([^\$]+)',assemblecode) #beq/bne $rs,$rt,label
        if a:return opcode[a.group(1)] +register_translated(a.group(2)) +register_translated(a.group(3)) +
                                                                imme_translated((lambda x:int(x,16) if x.
                                                                startswith('0x') else int(x,10))(a.group(4)))
        a =re.match(r'lui[ ]+\$(.*)[ ]*,[ ]*([^\$]+)', assemblecode) #lui $rt,imm
        if a: return opcode['lui'] +'00000' +register_translated(a.group(1)) +imme_translated((lambda x: int(x, 16)
                                                                 if x.startswith('0x') else int(x, 10))(a.group(2)
                                                                ))
        a =re.match(r'(.*)[ ]+\$(.*)[ ]*,[ ]*([^\$]+)', assemblecode) #blez/bgtz/bltz $rs,label
        if a: return opcode[a.group(1)] +register_translated(a.group(2)) +'00000' +imme_translated((lambda x: int(x
                                                                , 16) if x.startswith('0x') else int(x, 10))(a.
                                                                group(3)))
        a =re.match(r'(jal|j)[ ]+(.+)', assemblecode)
        if a: return opcode[a.group(1)] +imme_translated_26((lambda x: int(x, 16) if x.startswith('0x') else int(x,
                                                                10))(a.group(2)))
        a =re.match(r'(jr)[ ]+\$(.+)', assemblecode)
        if a: return opcode['jr'] +register_translated(a.group(2)) +'0' *15 +funccode['jr']
        a =re.match(r'jalr[ ]+\$(.+)[ ]*,[ ]*\$(.+)', assemblecode)
        if a: return opcode['jalr'] +register_translated(a.group(1)) +'0' *5 +register_translated(a.group(2)) +'0'
                                                                *5 +funccode['jalr']
    except ValueError as e:
        print("Error in line:" +e.args[0].split()[0] +"is not a register name!")
        return None
    except KeyError as e:
```

```
        print("Error in line:" +e.args[0].split()[0] +" is not a instruction or the way using it is unproper")
        return None
    except Exception as e:
        print("Unknown Error!",e)
        return None
```

3 仿真结果与分析

3.1 ALU 仿真

ALU 仿真即使用不同的数来运行不同的指令，观察是否符合现象。

图 3: ADD/SUB/AND

图 4: OR/XOR/NOR

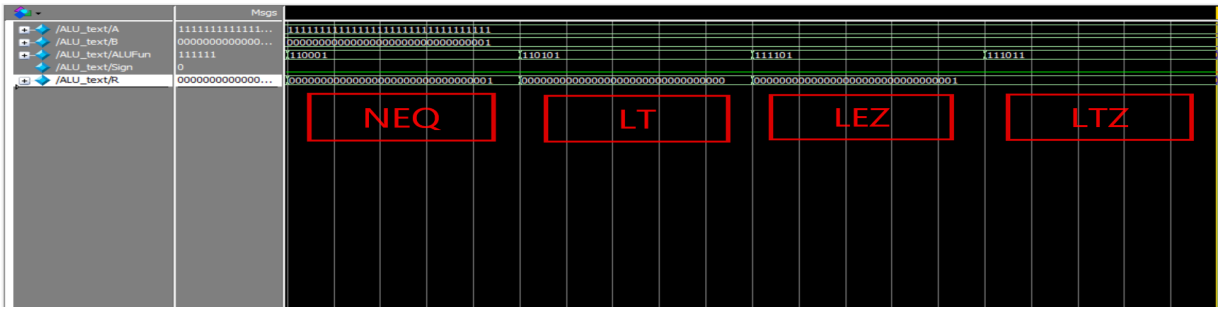

图 5: NEQ/LT/LEZ/LTZ

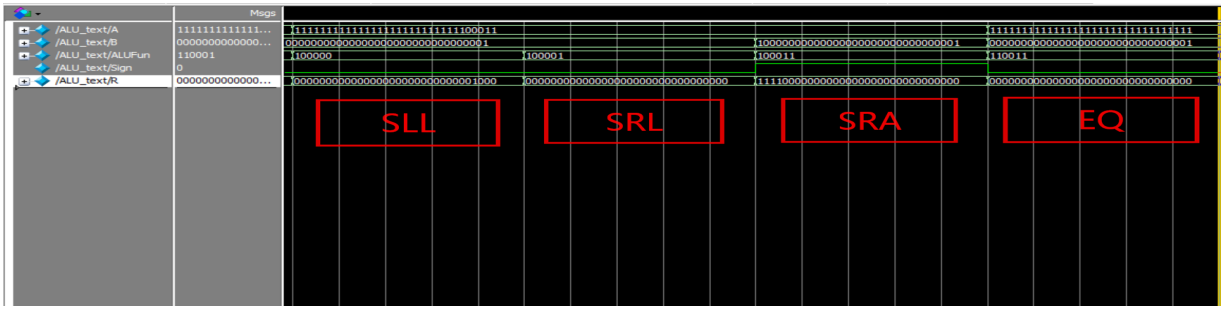

图 6: SLL/SRL/SRA/EQ

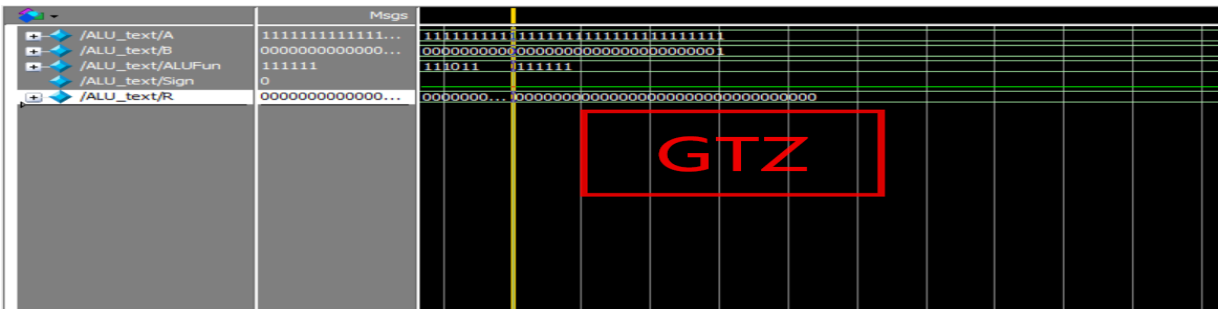

图 7: GTZ

3.2 单周期 CPU 仿真

这是对整个 CPU 进行的完整仿真，从仿真结果来看，每隔相同的时间，就会中断一次（见 interrupt 信号）；Rx 明显输入了两个数据（3 和 15），当两个数输入完成之后，主程序算出最大公约数，通过串口发送出去，现象见图8。

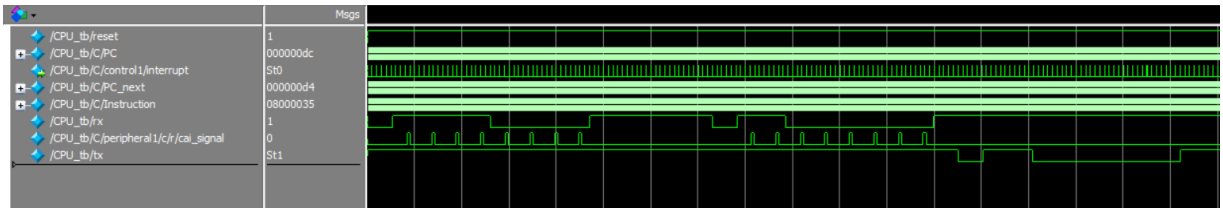

图 8: 单周期 CPU 基本运算波形

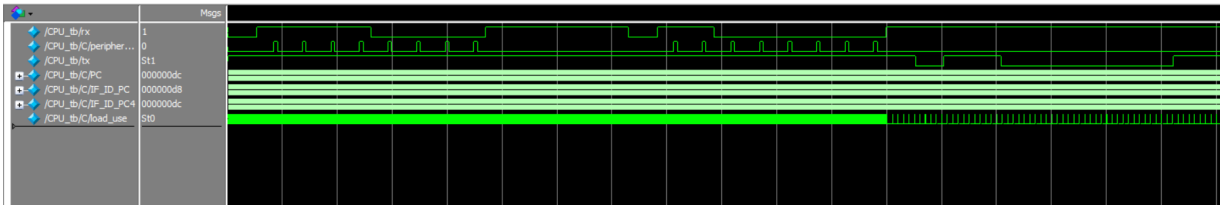

图 9: 流水线 CPU 基本运算波形

3.3 流水线 CPU 仿真

这也是对整个 CPU 进行仿真的，从仿真结果来看很好的完成了设计要求。

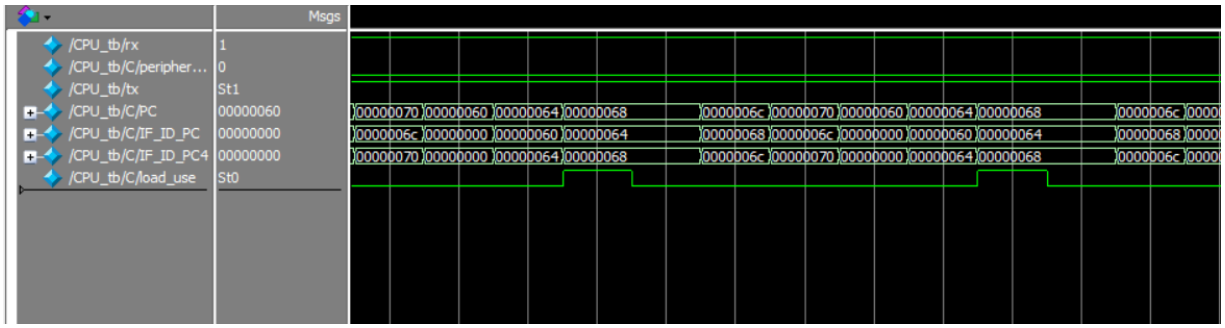

图 10: 阻塞仿真情况

图10是 load_use 阻塞的示例仿真，当阻塞的时候，则需要将 PC 保持一个周期。

4 综合情况

4.1 单周期 CPU

单周期 CPU 最高频率为 32.97MHz，不过我们实际在板子上跑的时候，50MHz 也是没有问题的，后来我们分析一下，我们的汇编代码并没有使用全部的指令集，因此可能有一条指令占得时间比较长，所以会导致整体的时钟频率比较低。

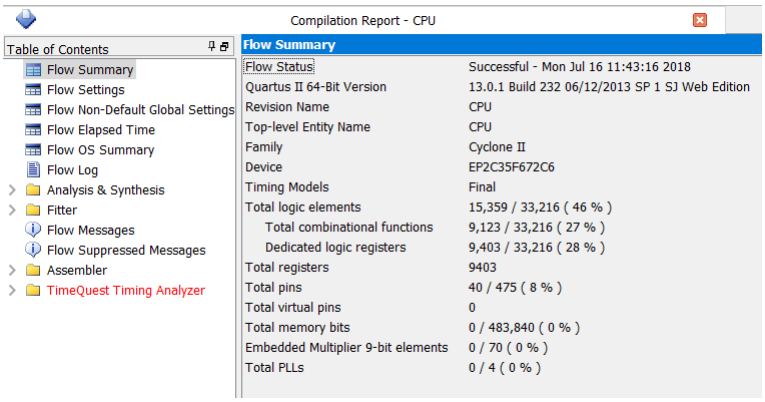

图 11: 单周期综合情况

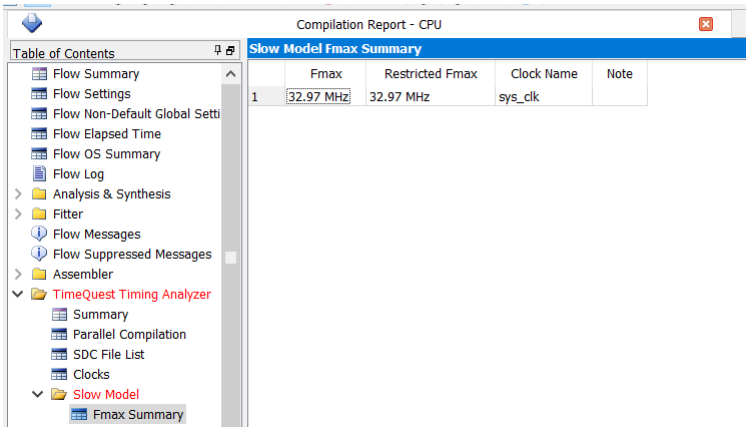

图 12: 单周期时序情况

4.2 流水线 CPU

流水线 CPU 最高时钟频率为 75.75MHz，运行速度还是比较快的。资源使用率中等。

5 思想体会

5.1 单周期 CPU

在写单周期 CPU 的时候，因为在春季学期做过单周期 CPU 的大作业，因此有一些经验，比较容易地完成了一些模块的设计，最难的部分是外设部分，即是如何通过外设部分来实现串口收发功能，刚开始我们并没有意识到需要通过软件来译码，同时也需要软件来打开计时 TCON，所以刚开始的时候我们仿真并没有任何的现象，后来通过不断地讨论和调试，发现这是需要和汇编代码结合起来的实现的。在完成相关代码编写之后，又陆陆续续出现了很多 bug，通过这次实验，提高了我们对通过仿真现象来推断问题并改正的能力。

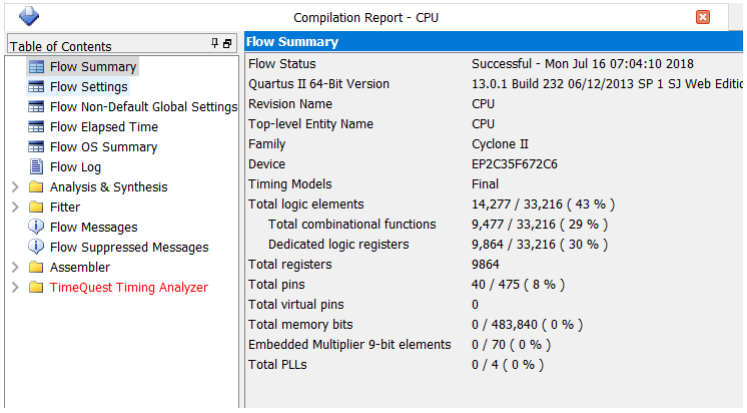

图 13: 流水线综合情况

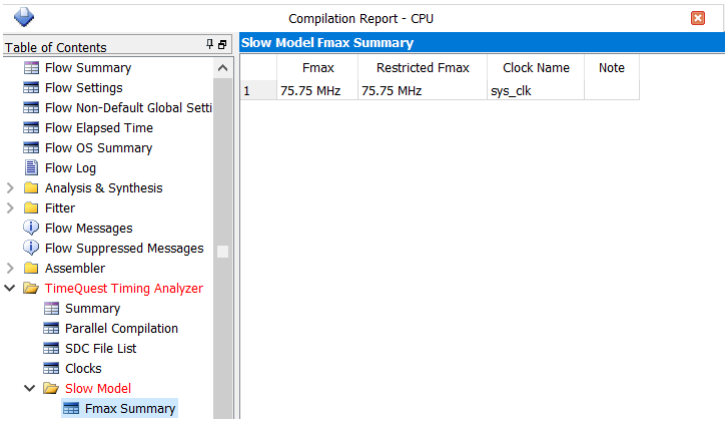

图 14: 流水线时序情况

5.2 流水线 CPU

流水线 CPU 和单周期 CPU 比较起来，流水线 CPU 在调试的时候更难。特别是在 IF 和 ID 阶段，对 PC 和 instruction 的操作花了我们很多时间去讨论和调试。因为在单周期 CPU 中已经完成各个模块功能的调试，在编写流水线 CPU 的时候，只用修改顶层模块，而不用管其余的的子模块。

在调试中，我们碰到最难的一个问题，是有些 ALU 的输出有问题，输出的是不定值，我们挨个排查，用了一整天的时间来找问题，后来发现是 beq 的分支地址 PC 有问题，改正这个问题之后，流水线很快就调试完成了。

5.3 汇编器及汇编代码

汇编器的编写让我熟悉了正则表达式的运用，是一次拓展技能点的好机会。汇编代码让我们更加深入的理解了如何用软件操作实现硬件功能，很大程度的提高了我 MIPS 汇编代码的编写能力。

通过这次 CPU 大作业，让我们对单周期和流水线 CPU 的认识更加深刻，同时也提升了通过仿

真和硬件结果来调试的能力，收获满满!

6 任务分工

任务	完成人
ALU	徐远帆
控制信号	刘树清
单周期 CPU	漆耘含
流水线 CPU	漆耘含
汇编器及汇编代码	徐远帆
仿真	漆耘含、刘树清
实验报告撰写	漆耘含、刘树清、徐远帆
实验报告排版	徐远帆

7 关键代码与文件清单

```
assemble
      assembly.py  汇编器程序
      GCD.asm 求最大公约数的汇编程序

singlecycle_CPU
     Adder.v 全加器
     ALU.v 单周期ALU
     control.v 单周期控制信号生成文件
     CPU.v 单周期CPU结构文件
     Data_Memory.v 单周期Data Memory
     digitube_scan.v
     receiver.v      串口接收
     sender.v  串口发射
     divide_clk.v 分频模块
     Peripheral.v 外设模块
     Register_file.v 寄存器
     Instruction_Memory.v 指令存储器
     chuankou.v 串口相关电路实现


Pipeline_CPU
```

Adder.v 全加器
ALU.v 单周期ALU
control.v 单周期控制信号生成文件
CPU.v 单周期CPU结构文件
Data_Memory.v 单周期Data Memory
digitube_scan.v
receiver.v 串口接收
sender.v 串口发射
divide_clk.v 分频模块
Peripheral.v 外设模块
Register_file.v 寄存器
Instruction_Memory.v 指令存储器
chuankou.v 串口相关电路实现