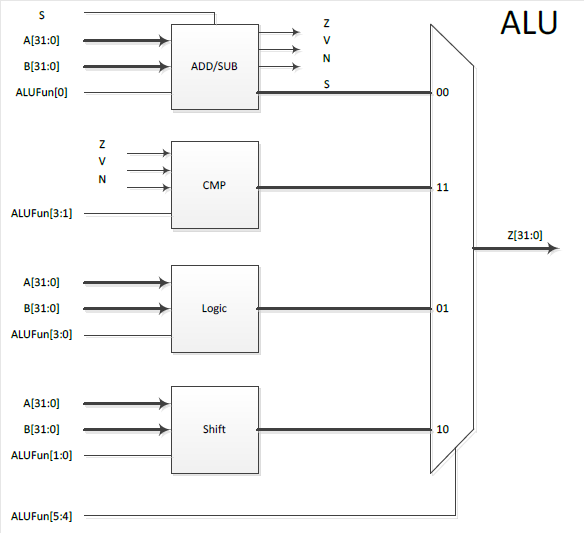
**实验名称：**32位处理器设计

**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。

1. 32位ALU的设计
2. **设计方案**

ALU用来实现基本的运算，主要分为四种运算，算术运算、关系运算、位运算以及移位运算。四种运算可以分别用四个模块实现，如下图所示：

其中算术运算模块中需要一个32位全加器，并且需要利用运算结果进行正负、是否为零或是溢出的判断，关系运算模块需要利用这些信号进行关系的判断。位运算模块可直接进行按位逻辑的运算，移位运算通过指令中五位shamt码来进行移位的操作。四种运算可以在任一指令下同时进行，最终经过一个四路选择器选出应当输出的信号。

1. **算术运算模块**

算术运算模块中只需要一个32位全加器，因为减法可以利用补码的特性，将减数取反加1后与被减数进行加法运算，是否为零通过对运算结果32位取或非来进行判断，溢出和正负的判断需要依靠输入信号Sign，在有符号数运算和无符号数运算两种情况下进行判断。对于溢出的判断，在有符号数运算下，若两符号相同的数进行加法运算产生结果负号与其不同，则产生溢出，对于无符号数运算，产生进位则代表溢出，减法中溢出判断需要使用A操作数和B取反加一后的操作数neg。关于正负的判断，还需要考虑溢出情况，在有符号数运算下，若结果为负，只要A和neg不同时为正，则为负，无符号数运算下，只有在减法运算下，若最高位进位不为1,则结果为负。

关键代码：

always @ (\*) begin

if(~ALUFun[0]) begin //加法 A与B为操作数

S00 = A + B;

neg = B;

end

else begin //减法 neg为中间变量

neg = notB + 1;

S00 = A + neg;

end

if(Sign) begin //符号、溢出及是否为零判断 V为1表示溢出

if(S00[31]) begin //N为1表示为负，Z0为1表示为0

V = (~A[31]) & (~neg[31]);

N = A[31] | neg[31];

end

else begin

V = A[31] & neg[31];

N = A[31] & neg[31];

end

end

else begin

if(~ALUFun[0]) begin

V = S00[32];

N=0;

end

else begin

V = ~S00[32];

N = ~S00[32];

end

end

Z0=~(|S00[31:0]);

end

1. **关系运算模块**

关系模块需要用到算数运算模块产生的符号和是否为零的信号进行判断，只有输出的最后一位有效。

关键代码：

always @(\*)

case(ALUFun[3:1])

3'b000:cmp <= {31'h0000000,~Z0}; //NEQ 不等于

3'b001:cmp <= {31'h0000000,Z0}; //EQ 等于

3'b010:cmp <= {31'h0000000,N}; //LT 小于

3'b110:cmp <= {31'h0000000,N|Z0}; //LEZ 小于或等于

3'b101:cmp <= {31'h0000000,N}; //LTZ 小于

3'b111:cmp <= {31'h0000000,~(N|Z0)}; //GTZ 大于

default:cmp <= 32'h0000000;

endcase

1. **位运算模块**

位运算模块可直接使用verilog中的逻辑运算符来实现。实现与、或、异或、或非及S=A逻辑运算。

关键代码：

always @(\*) begin

case(ALUFun[3:0])

4'b1000: log<=A&B;

4'b1110: log<=A|B;

4'b0110: log<=A^B;

4'b0001: log<=~(A|B);

4'b1010: log<=A;

default: log <= 32'h00000000;

endcase

end

1. **移位运算模块**

移位运算模块可以通过五个双路选择器和16位移位器、8位移位器、4位移位器、2位移位器和1位移位器级联组成。shamt的4,3,2,1,0位分别决定操作数是否经过上述五个移位器，因为其位权分别为16,8,4,2,1。可进行逻辑左移、逻辑右移及算术右移运算。

关键代码：

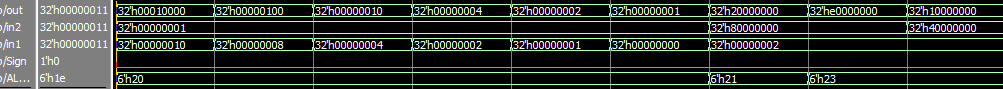
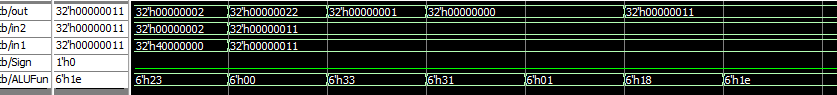
assign shamt1 = (A[0])?(ALUFun[0]?{(ALUFun[1]?B[31]:1'b0),B[31:1]}:{B[30:0],1'b0}):B;

assign shamt2 = (A[1]) ? (ALUFun[0] ? {ALUFun[1] ? {2{shamt1[31]}} : 2'b0, shamt1[31:2]} : {shamt1[29:0], 2'b0}) : shamt1;

assign shamt4 = (A[2]) ? (ALUFun[0] ? {ALUFun[1] ? {4{shamt2[31]}} : 4'b0, shamt2[31:4]} : {shamt2[27:0],4'b0}) : shamt2;

assign shamt8 = (A[3]) ? (ALUFun[0] ? {ALUFun[1] ? {8{shamt4[31]}} : 8'b0, shamt4[31:8]} : {shamt4[23:0],8'b0}) : shamt4;

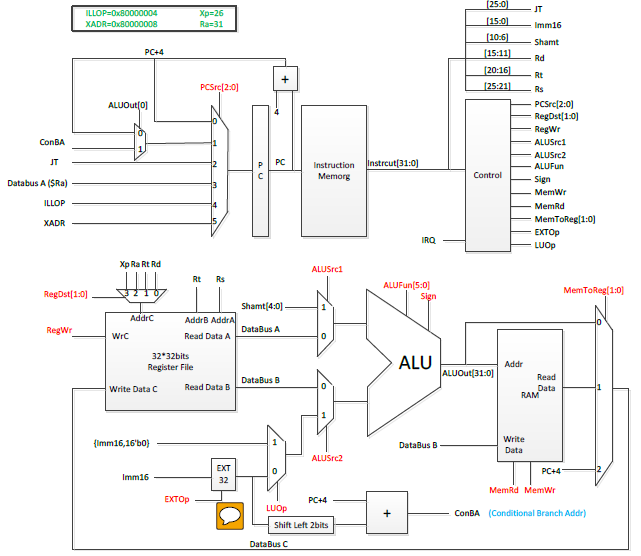
assign S10 = (A[4]) ? (ALUFun[0] ? {ALUFun[1] ? {16{shamt8[31]}} : 16'b0, shamt8[31:16]} : {shamt8[15:0],16'b0}) : shamt8;

1. **仿真结果及分析**

上面的图为移位操作仿真，分别将1进行左移16位，左移8位，左移4位，左移2位，左移1位，左移0位，输出分别为32’h00010000, 32’h00000100, 32’h00000010, 32’h00000004, 32’h00000002, 32’h00000001，再对32’h80000000进行逻辑右移2位和算术右移2位操作，输出分别为32’h20000000和32’he0000000，说明输出结果分别在高位补0和补1，最后对32’h40000000进行算术右移2位操作，输出为32’h10000000，由于31位为0，故在高位补0。移位运算模块正常。

下面的图分别对32’h00000011和32’h00000011两个操作数进行加法、相等判断、不相等判断、减法运算、相与运算、相或运算。输出结果分别为32’h00000022, 32’h00000001, 32’h00000000, 32’h00000000, 32’h00000011, 32’h00000011。输出结果正确，ALU的算术运算、关系运算、逻辑运算工作正常。

1. 单周期MIPS处理器的设计
2. **设计方案**

单周期MIPS处理器的数据通路如下图：

单周期MIPS处理器主要由PC程序计数器，指令存储器，寄存器堆，ALU，数据存储器和外设，以及加法器、多个多路复用器、32位扩展器和控制模块组成，主要需要设计的是根据不同指令产生控制信号的控制模块以及各个主体模块之间的数据通路。

数据通路基本按照PC至指令存储器ROM，至寄存器堆，至ALU运算单元，至数据存储器，再由写会的通路连接到寄存器堆组成。指令存储器由只读存储器ROM实现，数据存储器由RAM实现，外设与RAM地址分开形成地址映射，寄存器堆由32个32位寄存器构成，PC也通过多路复用器、加法器以及寄存器进行实现。根据PC进行指令的提取后，对指令进行译码，将指令的不同部分分别送至控制单元、寄存器堆、扩展单元以及ALU的源。

1. **控制信号模块设计**

控制信号模块输出的控制信号控制着寄存器堆、存储器的读写以及多路复用器信号的选取。

控制信号涉及：控制PC选取的PCSrc信号，控制寄存器写的RegWrite信号，控制写寄存器选取的RegDst信号，决定ALU运算性质的ALUFun信号，决定ALU算术运算是否带符号的Sign信号，决定ALU操作数来源的ALUSrc1和ALUSrc2信号，决定进行符号扩展还是零扩展的ExtOp信号，决定是否将立即数装入高16位的LuOp信号，控制存储器读或写的MemRead和MemWrite信号，决定写入寄存器数据来源的MemtoReg信号。不同指令所对应的操作码、功能码以及相应的控制信号如下表：

续上页

此模块只需要依靠条件语句来实现，关键代码如下：

assign exception = ~(OpCode == 6'h23 || OpCode == 6'h2b || OpCode == 6'h0f || OpCode == 6'h08 || OpCode == 6'h09 || OpCode == 6'h0c || OpCode == 6'h20 || OpCode == 6'h0a || OpCode == 6'h0b || OpCode == 6'h04 || OpCode == 6'h05 || OpCode == 6'h06 || OpCode == 6'h07 || OpCode == 6'h01 || OpCode == 6'h02 || OpCode == 6'h03 || (OpCode == 6'h00 && (Funct == 6'h00 || Funct == 6'h02 || Funct == 6'h03 || Funct == 6'h22 || Funct == 6'h23 || Funct == 6'h08 || Funct == 6'h09 || Funct == 6'h20 || Funct == 6'h21 || Funct == 6'h24 || Funct == 6'h25 || Funct == 6'h26 || Funct == 6'h27 || Funct == 6'h2a || Funct == 6'h2b))); //用于异常的判断：未定义指令

assign XADR = (~PC\_31)&exception; //异常

assign ILLOP = (~PC\_31)&IRQ; //中断

assign PCSrc[2:0] =

(ILLOP)? 3'b100:

(XADR)? 3'b101:

(OpCode == 6'h04 || OpCode == 6'h05 || OpCode == 6'h06 || OpCode == 6'h07 || OpCode == 6'h01)? 3'b001:

(OpCode == 6'h02 || OpCode == 6'h03)? 3'b010:

(OpCode == 6'h00 && (Funct == 6'h08 || Funct == 6'h09))? 3'b011:3'b000;

assign RegWrite =

(ILLOP)?1:

(OpCode == 6'h2b || OpCode == 6'h04 || OpCode == 6'h02 || OpCode == 6'h05 || OpCode == 6'h06 || OpCode == 6'h07 || OpCode == 6'h01)? 0:

(OpCode == 6'h00 && Funct == 6'h08)? 0: 1;

assign RegDst[1:0] =

(ILLOP || XADR)? 2'b11:

(OpCode == 6'h00)? 2'b00:

(OpCode == 6'h03)? 2'b10: 2'b01;

assign MemRead = (OpCode == 6'h23)? 1: 0;

assign MemWrite = (OpCode == 6'h2b)? 1: 0;

assign MemtoReg =

(ILLOP || exception)? 2'b10:

(OpCode == 6'h23)? 2'b01:

(OpCode == 6'h03 || OpCode == 6'h20)? 2'b10:

(OpCode == 6'h00 && Funct == 6'h09)? 2'b10: 2'b00;

assign ALUSrc1 = (OpCode == 6'h00 && (Funct == 6'h00 || Funct == 6'h02 || Funct == 6'h03))? 1:0;

assign ALUSrc2 = (OpCode == 6'h00 || OpCode == 6'h04 || OpCode == 6'h05 || OpCode == 6'h06 || OpCode == 6'h07 || OpCode == 6'h01)? 0:1;

assign ExtOp = (OpCode == 6'h0c)? 0:1;

assign LuOp = (OpCode == 6'h0f)? 1:0;

assign Sign = (OpCode == 6'h09 || OpCode == 6'h0b)? 0:

(OpCode == 6'h00 && (Funct == 6'h2b || Funct == 6'h05 || Funct == 6'h21))? 0:1;

assign ALUFun[5:0] =

(OpCode == 6'h0f || (OpCode == 6'h00 && Funct == 6'h25))? 6'b011110: //lui or

(OpCode == 6'h00 && (Funct == 6'h22 || Funct == 6'h23))? 6'b000001: //sub subu

(OpCode == 6'h0c || (OpCode == 6'h00 && Funct == 6'h24))? 6'b011000: //and andi

(OpCode == 6'h00 && Funct == 6'h26)? 6'b010110: //xor

(OpCode == 6'h00 && Funct == 6'h27)? 6'b010001: //nor

(OpCode == 6'h00 && Funct == 6'h00)? 6'b100000: //sll

(OpCode == 6'h00 && Funct == 6'h02)? 6'b100001: //srl

(OpCode == 6'h00 && Funct == 6'h03)? 6'b100011: //sra

(OpCode == 6'h0a || OpCode == 6'h0b || (OpCode == 6'h00 && (Funct == 6'h2a || Funct == 6'h2b)))? 6'b110101: //slt slti sltiu sltu

(OpCode == 6'h04)? 6'b110011: //beq

(OpCode == 6'h05)? 6'b110001: //bne

(OpCode == 6'h06)? 6'b111101: //blez

(OpCode == 6'h07)? 6'b111111 : //bgtz

(OpCode == 6'h01)? 6'b111011: 6'b000000;

1. **PC的设计**

PC相当于为多路复用器、寄存器及加法器的组合。多路复用器选择PC的来源。在此数据通路中，PC值有6个来源，分别是PC+4，分支指令中的Branch\_Target，跳转指令中的Jump\_Target，跳转至寄存器指令中来源于寄存器的PC值以及中断时的0x80000004和异常时的0x80000008.

同时，实验中要求PC的最高位PC[31]为监督位。当该位为‘1’时，处理器处于内核态，此时异常和中断被禁止；当该位为‘0’时，处理器处于普通态，此时允许发生中断和异常。PC[31]不能作为地址最高位去索引指令存储器，取指令时应当固定将地址最高位置零。只有RESET、异常、中断等有可能将PC[31]设置为‘1’，其他指令不能设置该位为‘1’，JR和JALR指令可以使监督位清零。数据通路需要保证，PC+4逻辑电路实现时PC[31]应不变，分支语句和J、JAL语句不应该改变PC[31]。

PC实现的关键代码如下：

parameter ILLOP = 32'h80000004;

parameter XADR = 32'h80000008;

// Program Counter

reg [31:0] PC;

wire [31:0] PC\_next;

always @(negedge reset or posedge clk)

if (~reset)

PC <= 32'h80000000;

else

PC <= PC\_next;

wire [31:0] PC\_plus\_4;

//PC+4

assign PC\_plus\_4[30:0] = PC[30:0] + 4;

assign PC\_plus\_4[31] = PC[31];

// J

wire [31:0] Jump\_target;

assign Jump\_target = {PC\_plus\_4[31:28], Instruction[25:0], 2'b00};

// branch

wire [31:0] Branch\_target;

assign Branch\_target = (ALU\_out[0])? PC\_plus\_4 + {Ext\_out[29:0], 2'b00}: PC\_plus\_4;

// PC Source

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:

(PCSrc == 3'b101)? XADR: 32'h80000000;

1. **数据存储器及外设设计**

数据存储地址分为两部分，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，对其地址的读写与相应的外设资源形成映射。

对于数据RAM需要进行修改，由于RAMDATA中每一个元素声明的大小为一个字，故在进行寻址时需要对齐到字，去掉地址的后两位再进行寻址，关键代码如下：

assign rdata = (rd && (addr[RAM\_SIZE\_BIT+1:2] < RAM\_SIZE))? RAMDATA [addr[RAM\_SIZE\_BIT+1:2]] : 32'h00000000;

always@(posedge clk) begin

if(wr && (addr[RAM\_SIZE\_BIT+1:2] < RAM\_SIZE)) RAMDATA[addr[RAM\_SIZE\_BIT+1:2]] <= wdata;

end

1. **UART串口设计**

UART串口使用了串口实验中的设计，其中控制模块包含在外设模块之中。在接收到一个数据后，UART串口接收单元发送一个系统时钟周期时长的rx\_status有效信号到外设模块，使得外设模块中串口数据接收状态位UART\_CON[0]置1，地址0x4000001c映射至接收的数据。当第二个数据到来后，串口接收单元再发送一个系统时钟周期时长的rx\_status有效信号到外设模块使UART\_CON[1]置1。当整个程序运行结束后，会往UART\_CON写入0，并将运算结果存入发送数据对应地址，在进行该写操作的同时，在发送模块空闲的状态下，向UART发送模块发送1个16倍9600波特率时钟周期长度的发送使能信号。

由于进行串口实验时我的各个模块都是由16倍的波特率时钟触发，故在此试验中需要进行更改，使rx\_status为1的状态只持续1个系统时钟周期。

UART接收模块关键代码如下：uart\_receiver.v

always @(posedge baudclk or negedge reset) begin

if(~reset) begin

state<=2'b0; rstatus<=0; data<=8'b0; cnt<=4'b0;

end

else begin

case(state)

2'b0:begin

rstatus<=0;

if(~uart\_rx) begin

cnt<=4'b0; state<=2'b01; num<=4'b0;

end

end

2'b1:begin

if(cnt==8 && num<=8) begin

if(num!=4'b0) begin

data[num-1]<=uart\_rx;

end

cnt<=cnt+1; num<=num+1;

end

else if(cnt==8 && num==9) begin

state<=2'b10; rstatus<=1; rx\_data<=data;

end

else

cnt<=cnt+1;

end

2'b10:begin

state<=2'b0;

end

endcase

end

end

always @(posedge clk or negedge reset) begin

if(~reset) begin

rx\_status <= 0; cnt1 <= 9'd0;

end

else begin

if(rstatus && ~(|cnt1))

rx\_status <= 1;

else rx\_status <= 0;

if(rstatus)

cnt1 <= cnt1+1;

if(~rstatus)

cnt1 <= 0;

end

end

UART发送模块关键代码如下：uart\_sender.v

always @(posedge baudclk or negedge reset) begin

if(~reset) begin

tx\_status<=1; uart\_tx<=1; cnt<=4'b0; num<=4'b0; state<=2'b0;

end

else begin

case(state)

2'b0:begin

tx\_status<=1;

if(tx\_en) begin

state<=1; tx\_status<=0; cnt<=4'b0; num<=4'b0;

end

end

2'b1:begin

if(cnt==0 && num==0) begin

uart\_tx<=0; num<=num+1;

end

else if(cnt==0 && num<=8) begin

uart\_tx<=tx\_data[num-1]; num<=num+1;

end

else if(cnt==0 && num==9) begin

state<=2'b10; uart\_tx<=1;

end

cnt<=cnt+1;

end

2'b10:begin

if(cnt==14) begin

state<=2'b0;

end

cnt<=cnt+1;

end

endcase

end

end

外设模块中UART控制部分关键代码如下：Periphery.v

always@(\*) begin //读模块

if(rd) begin

case(addr)

……

32'h40000018: rdata <= {24'b0,UART\_TXD};

32'h4000001c: begin

rdata <= {24'b0,UART\_RXD};

end

32'h40000020: rdata <= {29'b0,UART\_CON};

default: rdata <= 32'b0;

endcase

end

else

rdata <= 32'b0;

end

always@(negedge reset or posedge clk) begin

if(~reset) begin

……

tx\_en <= 1'b0;

UART\_RXD <= 8'b0;

UART\_TXD <= 8'b0;

UART\_CON <= 3'b0;

cnt <= 9'b0;

end

else begin

……

if(rx\_status) begin

UART\_RXD <= rx\_data;

if(UART\_CON[0])

UART\_CON[1] <= 1;

else

UART\_CON[0] <= 1;

end

if(tx\_en) begin //使tx\_en能够维持一个16倍9600波特率时钟的时钟周期

if(cnt == 9'd326)

begin tx\_en <= 1'b0; end

else

cnt <= cnt+1;

end

if(wr) begin

case(addr)

……

32'h40000018:

begin

UART\_TXD <= wdata[7:0];

if(tx\_status)

tx\_en <= 1'b1;

else

tx\_en <= 1'b0;

cnt <= 9'b0;

end

32'h40000020: UART\_CON <= wdata[2:0];

default: ;

endcase

end

end

end

assign tx\_data = UART\_TXD;

endmodule

1. **数据通路设计**

对于将16位立即数扩展为32位，需要分别考虑符号扩展、零扩展以及将立即数装入高位。关键代码如下：CPU.v（顶层模块）

// Extension 16-32

wire [31:0] Ext\_out;

assign Ext\_out = {ExtOp? {16{Instruction[15]}}: 16'h0000, Instruction[15:0]};

// lui

wire [31:0] LU\_out;

assign LU\_out = LuOp? {Instruction[15:0], 16'h0000}: Ext\_out;

ALU的操作数需要进行选取，关键代码如下：

assign ALU\_in1 = ALUSrc1? {27'h00000, Instruction[10:6]}: Databus1;

assign ALU\_in2 = ALUSrc2? LU\_out: Databus2;

ALU alu1(.A(ALU\_in1), .B(ALU\_in2), .ALUFun(ALUFun), .Sign(Sign), .Z(ALU\_out));

由于数据存储器地址分为两部分，分别为数据RAM和外设所使用，且两者未被访问时数据输出都为0，若只使用一个net型变量会造成冲突，输出会变为不定态，故需要加一个双路选择器。由于二者的地址范围分别为0至0x3fffffff和0x40000000至0x7fffffff，故可以利用地址的第30位进行判断。第30位为0则访问数据RAM，为1则访问外设。关键代码如下：

DataMem DataMem1(.reset(reset), .clk(clk), .addr(ALU\_out), .wdata(Databus2), .rdata(Read\_data1), .rd(MemRead), .wr(MemWrite));

Peripheral Peripheral1(.reset(reset), .clk(clk), .rd(MemRead), .wr(MemWrite), .addr(ALU\_out), .wdata(Databus2), .rdata(Read\_data2), .led(led), .switch(switch), .digi(digi), .irqout(IRQ), .tx\_en(tx\_en), .tx\_status(tx\_status), .rx\_status(rx\_status), .rx\_data(rx\_data), .tx\_data(tx\_data)); //

assign Read\_data = (ALU\_out[30])? Read\_data2: Read\_data1;

输出至寄存器堆的数据也需要进行选择，分别来源于ALU运算结果、数据存储器和PC+4。关键代码如下：

assign Databus3 = (MemtoReg == 2'b00)? ALU\_out: (MemtoReg == 2'b01)? Read\_data: PC\_plus\_4;

1. **汇编代码设计**

进行最大公约数运算并显示的汇编语言代码如下;

j main # 0x80000000 复位

j interrupt # 0x80000004 中断

jr $k0 # 0x80000008 $26 异常处理（由于只涉及到未定义指令，直接进行下一条指令）

main:

jal Initial #跳转至初始化部分

num1: #接收串口发送的第一个数

lw $t1,8($s0)

andi $t1,$t1,1

beq $t1,$zero,num1 #采用轮询方式进行接收

lw $a0,4($s0)

num2: #接收串口发送的第二个数

lw $t1,8($s0)

andi $t1,$t1,2

beq $t1,$zero,num2

lw $a1,4($s0)

#对两个操作数进行BCD译码

andi $t0,$a0,15 #对第一个数的低四位译码

addi $a2,$t0,256 #在HEX0进行显示

jal Decode

add $s4,$a2,$zero

srl $t0,$a0,4 #对第一个数的高四位译码

andi $t0,$t0,15

addi $a2,$t0,512 #在HEX1进行显示

jal Decode

add $s5,$a2,$zero

andi $t0,$a1,15 #对第二个数的低四位译码

addi $a2,$t0,1024 #在HEX2进行显示

jal Decode

add $s6,$a2,$zero

srl $t0,$a1,4 #对第二个数的高四位译码

andi $t0,$t0,15 #在HEX3进行显示

addi $a2,$t0,2048

jal Decode

add $s7,$a2,$zero #decode

#求解最大公约数

slt $t4,$a0,$a1

beq $t4,$zero,Great

add $t4,$a1,$zero

sub $t5,$a1,$a0

j Loop

Great:

add $t4,$a0,$zero

sub $t5,$a0,$a1

Loop:

beq $t5,$zero,End

sub $t6,$t4,$t5

slt $t7,$t5,$t6

beq $t7,$zero,Great1

add $t4,$t6,$zero

add $t5,$t5,$zero

j Loop

Great1:

add $t4,$t5,$zero

add $t5,$t6,$zero

j Loop

End:

add $v0,$t4,$zero #将运算结果存入v0

sw $v0,12($s1) #将运算结果存入外设中led所对应地址进行显示

sw $v0,0($s0) #将运算结果存入外设中UART\_TXD，使用串口进行发送

sw $zero,8($s0) #将串口数据接收状态清零

add $s2,$zero,$zero

addi $s3,$zero,4

#定时器的初始化

sw $zero,8($s1) #关闭定时器，TCON写入0;

lui $t0,65535

sra $t0,$t0,9 #设置定时器周期，TH取值决定定时器的计数周期;

sw $t0,0($s1)

lui $t1,65535

sra $t1,$t1,16 #设置定时器TL为0xFFFFFFFF;

addi $t2,$zero,3 #启动定时器，TCON写入3

sw $t1,4($s1)

sw $t2,8($s1)

Loop1:

jal Loop1 #进入死循环，等待进入中断

interrupt: #进入中断执行程序

lw $t2,8($s1)

andi $t4,$t2,9 #定时器中断禁止，同时中断状态清零，TCON的1-2bit清零

sw $t4,8($s1)

addi $s2,$s2,1 #开始中断处理

bne $s2,$s3,display

add $s2,$zero,$zero

display:

beq $s2,$zero,display0 #进行HEX0的显示

addi $t9,$zero,1

beq $s2,$t9,display1 #进行HEX1的显示

addi $t9,$zero,2

beq $s2,$t9,display2 #进行HEX2的显示

addi $t9,$zero,3

beq $s2,$t9,display3 #进行HEX3的显示

display0:

sw $s4,20($s1)

lw $t4,8($s1)

addi $t8,$zero,2

or $t4,$t4,$t8 #使能中断，TCON的1bit置1，TCON |= 0x00000002

sw $t4,8($s1)

addi $k0,$k0,-4 #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

jr $k0

display1:

sw $s5,20($s1)

lw $t4,8($s1)

addi $t8,$zero,2

or $t4,$t4,$t8 #使能中断，TCON的1bit置1，TCON |= 0x00000002

sw $t4,8($s1)

addi $k0,$k0,-4 #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

jr $k0

display2:

sw $s6,20($s1)

lw $t4,8($s1)

addi $t8,$zero,2

or $t4,$t4,$t8 #使能中断，TCON的1bit置1，TCON |= 0x00000002

sw $t4,8($s1)

addi $k0,$k0,-4 #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

jr $k0

display3:

sw $s7,20($s1)

lw $t4,8($s1)

addi $t8,$zero,2

or $t4,$t4,$t8 #使能中断，TCON的1bit置1，TCON |= 0x00000002

sw $t4,8($s1)

addi $k0,$k0,-4 #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

jr $k0

Decode: #BCD译码

andi $t8,$a2,15

addi $t9,$zero,0

beq $t8,$t9,zero

addi $t9,$zero,1

beq $t8,$t9,one

addi $t9,$zero,2

beq $t8,$t9,two

addi $t9,$zero,3

beq $t8,$t9,three

addi $t9,$zero,4

beq $t8,$t9,four

addi $t9,$zero,5

beq $t8,$t9,five

addi $t9,$zero,6

beq $t8,$t9,six

addi $t9,$zero,7

beq $t8,$t9,seven

addi $t9,$zero,8

beq $t8,$t9,eight

addi $t9,$zero,9

beq $t8,$t9,nine

addi $t9,$zero,10

beq $t8,$t9,ten

addi $t9,$zero,11

beq $t8,$t9,eleven

addi $t9,$zero,12

beq $t8,$t9,twelve

addi $t9,$zero,13

beq $t8,$t9,thirteen

addi $t9,$zero,14

beq $t8,$t9,fourteen

addi $t9,$zero,15

beq $t8,$t9,fifteen

zero:

addi $a2,$a2,64 #显示0,7‘d64

jr $ra

one:

addi $a2,$a2,120 #显示1,7‘d121

jr $ra

two:

addi $a2,$a2,34 #显示2,7‘d36

jr $ra

three:

addi $a2,$a2,45 #显示3,7‘d48

jr $ra

four:

addi $a2,$a2,21 #显示4,7‘d25

jr $ra

five:

addi $a2,$a2,13 #显示5,7‘d18

jr $ra

six:

addi $a2,$a2,-4 #显示6,7‘d2

jr $ra

seven:

addi $a2,$a2,113 #显示7,7‘d120

jr $ra

eight:

addi $a2,$a2,-8 #显示8,7‘d0

jr $ra

nine:

addi $a2,$a2,7 #显示9,7‘d16

jr $ra

ten:

addi $a2,$a2,-2 #显示a,7‘d8

jr $ra

eleven:

addi $a2,$a2,-8 #显示b,7‘d3

jr $ra

twelve:

addi $a2,$a2,58 #显示c,7‘d70

jr $ra

thirteen:

addi $a2,$a2,20 #显示d,7‘d33

jr $ra

fourteen:

addi $a2,$a2,-8 #显示e,7‘d6

jr $ra

fifteen:

addi $a2,$a2,-1 #显示f,7‘d14

jr $ra

Initial: #初始化部分

sll $ra,$ra,1 #将PC监督位置为0

srl $ra,$ra,1

addi $sp,$sp,1024 #根据数据RAM大小初始化$sp

lui $s1,16384 #0x40000000 0c led 00 TH 04 TL 08 TCON

lui $s0,16384

addi $s0,$s0,24 #0x40000018 uart\_tx 0x4000001c uart\_rx 0x40000020 uart\_con

jr $ra

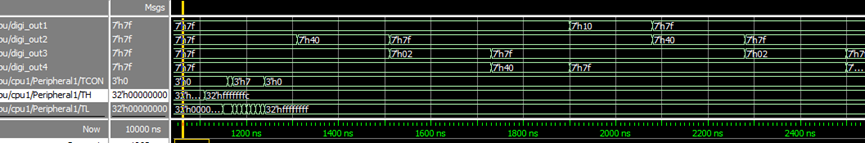
此汇编代码有一点缺陷，即在计算完两个操作数的最大公约数后，想要计算新的两个数的最大公约数时，需要进行复位，才能够输入两个新的整数。若希望能够连续输入，需要对此汇编代码做修改：

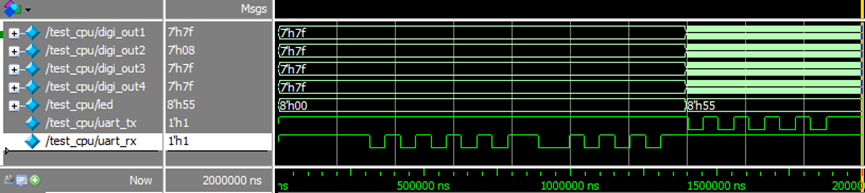
对于定时器初始化后的Loop1出，将jal Loop1改为j num1，进入接收第一个操作数的轮询循环中，并在 beq $t1,$zero,num1语句后加一条语句sw $zero,8($s1)，关闭定时器，避免在运算过程中不断进入中断。此外，需要在进入中断后对$t1寄存器中的值进行保护，存入数据RAM之中，并在中断程序执行结束后再使用lw指令将该值取出，恢复现场。

1. **仿真结果及分析**

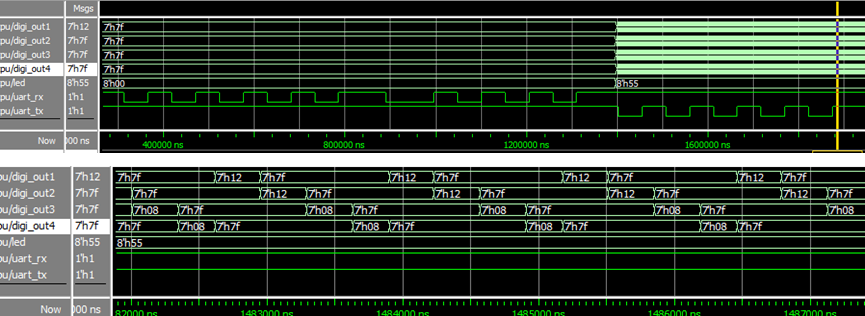
我将单周期CPU的仿真分为了两部分：定时器及终端的仿真，和串口的仿真。

对于定时器及中断的仿真，我去掉了汇编代码中串口接收部分，将求最大公约数程序的两个操作数直接赋为了6和9，进行数码管显示的仿真。仿真结果如下;

可见在进行完中断的初始化后，四个数码管开始交替显示数字，分别为9,0,6,0，通过视觉暂留效应，DE2开发板上从左至右会显示0609，正确地显示了最大公约数运算程序的两个操作数。

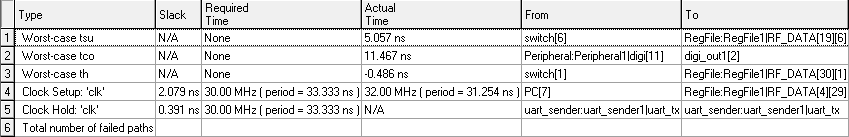
对于串口的仿真，在testbench文件中，设置先后向串口发送8’b10101010和8’b01010101（16进制aa和55,10进制170和85，大端在后），仿真结果如下：

串口发送8’b01010101（大端在后），16进制即为55，十进制即为85，结果正确。

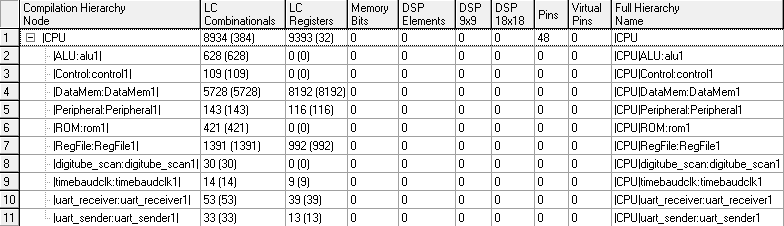
总体仿真结果如下：

可见在串口接收到数据8’haa和8’h55后，led灯显示01010101，即8’h55，数码管显示aa，55，串口发送回电脑的数据为8’h55，结果正确。

1. **综合情况**

时序分析结果如下：

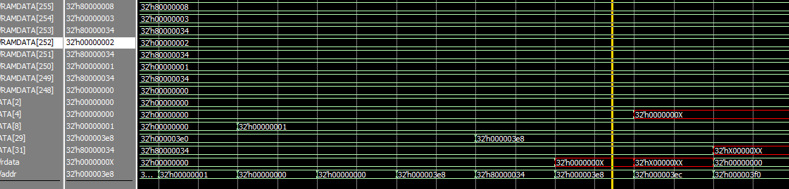
单周期32位MIPS处理器的最高时钟频率为32.00MHz。

面积综合情况如下：

除去数据存储器占用的寄存器，处理器所占用的寄存器为1201，占用的逻辑单元（查找表）为3206.

1. **硬件调试情况**

在进行硬件和汇编代码的调试过程中，为了不让二者的问题互相影响，我将UART串口和外设中断分开进行了调试。在硬件的调试过程中主要出现了如下问题：

1. ALU单元在进行溢出的判断中，在进行减法的判断时，应当在B操作数进行完取反加一的操作后，再与A相加进行判断。否则，在B取反加一的过程中，也有可能会产生溢出。
2. ALU单元为组合逻辑，涉及到了很多的条件语句，条件语句的分支必须完全，否则会形成锁存器。一开始在位运算模块的case语句中，我忘记了加default语句，使得该处形成锁存。在进行综合时，单周期CPU的最高时钟频率只有16.67MHz。
3. 在进行数据RAM地址的操作时，需要注意到RAM的地址存在着一个范围，在寄存器初始化后$sp内值为0时不可以直接使用，而需要软件进行初始化。硬件实现中声明数据RAM的大小为256个字，因此$sp需要初始化为1024，之后才可进行数据RAM空间地址声明的操作。
4. 由于外设中的地址映射和数据RAM的地址访问使用的是同一种模式的地址，只是分区不同，两者的访问也都是通过lw和sw指令，因而在二者数据通路的接线上容易出现问题。两者的输出不可连接到一个net型变量，否则会发生冲突，（同时Quartus在综合时也不允许出现这样的情况）。当我将两个模块的输出连接到同一个net型变量时仿真结果如图：

数据RAM读出的数据凡是为1的位最终结果都显示为不定态，这是由于在读取数据RAM时，外设模块输出恒为32’h00000000，两个部分输出不相等的部分为不定态。

为了解决这个问题，我在数据RAM和外设输出数据后加一双路选择器根据访问地址的范围进行数据的选取，解决了这一问题。

1. 在进行UART串口的设计时，由于接收器的时钟是由16倍9600波特率的时钟控制，与外设部分不同，导致了接收器输出的rx\_status信号要延续326个系统时钟的长度。在执行程序的过程中，在接收完第一个数据后的下一时钟周期，由于rx\_status信号还未变零，程序会当作第二个数据也已经接收完毕，导致接收数据的过程直接执行完毕，第二个数据却没有接收到。

这种情况下需要对接收模块做修改，使得rx\_status为1的情况只能延续一个时钟周期，与外设模块同步。需要添加如下模块来实现此功能：

always @(posedge clk or negedge reset) begin

if(~reset) begin

rx\_status <= 0; cnt1 <= 9'd0;

end

else begin

if(rstatus && ~(|cnt1))

rx\_status <= 1;

else rx\_status <= 0;

if(rstatus)

cnt1 <= cnt1+1;

if(~rstatus)

cnt1 <= 0;

end

end

其中cnt1为9位，可以保证在326个系统时钟周期内不再变为0.

1. 在寄存器中写入一个32位立即数的标准操作应当是lui指令和ori指令进行组合，但此次实现的CPU中并没有ori指令，在使用addiu或addi指令时，若低16位的第16位为1，使用这两条指令会产生错误。因而，在初始化计时器的过程中，需要使用sra指令来实现装入立即数0xffffffff的操作。

思想体会：

1. 通过这次试验，让我更加熟悉了verilog的语法，也更加熟悉了逻辑电路的组成以及一些规范，特别是一些逻辑电路设计方面的常识，例如不可以将两个不同模块的输出连接到同一个net型变量，以及不存在电路能够通过两个触发器对同一个变量进行赋值的操作，还有组合逻辑的条件语句中一定要将分支写全等，具备了这些基础的常识才能真正具备一个电子人的素养。
2. CPU是我们数字逻辑实验中所做过的规模最大的实验，在这样的实验中，模块化是非常重要的，分别完成每个模块，实现每个模块的功能，再在顶层模块中对各个功能模块进行连接，可以更有条理地完成整个工程。
3. 对于这个比较大的实验，仿真也需要分模块进行。一开始对整体进行仿真时，只能够看到什么结果也无法进行输出，UART模块和中断部分的问题交织在一起，让我感觉我从下手。于是我修改了汇编程序，将UART与中断的调试分开进行，很快找到了问题，并进行了修改，使得模块的功能变得正常。
4. 对于这个涉及到的变量很多的实验，需要对一些具有特别意义的变量进行关注，如PC值，PC\_next值以及特别的寄存器等，通过PC值可以判断程序执行得正不正常，CPU工作得正不正常，可以同时调试硬件和软件。当通过这些关键变量判断出某一步出现问题时，可以再对可能出现问题的部分变量的取值进行观察，特别是一些中间变量。