32 位 MIPS 处理器设计实验报告

无 42 2014011050 陈佳榕

无 42 2014011057 慕思成

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

实验分工

陈佳榕	ALU，编译器，流水线部分的冒险检测及转发单元
慕思成	单周期部分（除了编译器），流水线部分（除了冒险检测及转发单元）

（注：本实验报告是我们两个人将自己负责部分的实验报告写完之后汇总而成）

A、32 bit ALU（陈佳榕）

I、ADD/SUB 模块

㈠设计方案

由于ADD和SUB的ALUFun只有ALUFun[0]这一位是不同的，故而选择ALUFun[0]作为区分两种运算的比特位。

1.对于加法，直接使用“+”运算符将A和B加起来，需要着重考虑的是标志位的产生。首先Zero(结果为0，指导书上为Z)信号的产生对于有符号数和无符号数来说是统一的，即运算结果为0时Zero=1。对于剩下的两个标志位信号，需

要区分有符号数和无符号数两种情况，具体如下:

⑴有符号数

①对于Overflow(结果溢出，指导书上为V)信号，两个32位异号数相加的结果肯定不会超出32位数的表示范围，即不会发生溢出，只有当A与B是同号的且结果的符号位与A、B的符号位不同时会发生溢出。

②对于Negative(结果为负，指导书上为N)信号，直接从结果的最高位即符号位即可知道结果的正负。

⑵无符号数

①对于Overflow(结果溢出，指导书上为V)信号，当最高位有进位输出时表明发生了溢出，因为这说明结果已经超出了32位无符号数的表示范围。

②对于Negative(结果为负，指导书上为N)信号，由于是无符号数，所以结果都认为是正数，即Negative信号总是0。

2.对于减法，首先要将第二个操作数B转变为其二补码，然后接下来的运算便可以直接对A和B的二补码进行加法运算了。从而各个标志位的产生和加法的是一样的。

㈡关键代码及文件清单

“signalInput.v”

```
case (ALUFun0)
1'b0:                       //加法
begin
   if(Sign)                 //有符号数
   begin
```

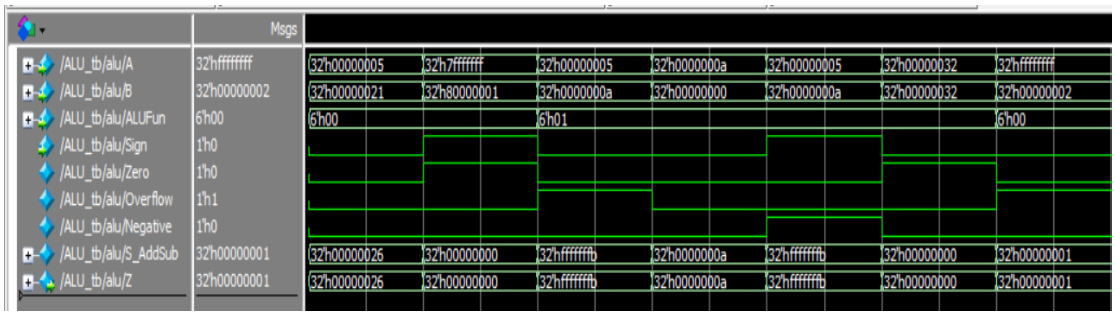
```
        S = A + B;
        Zero = (S == 0);
        Overflow = (A[31] == B[31] && A[31] != S[31]);
        Negative = S[31];
    end
    else                    //无符号数
    begin
        //Overflow 即最高位进位输出
        {Overflow,S} = {1'b0,A} + {1'b0,B};
        Zero = (S == 0);
        Negative = 0;
    end
end
1'b1:                       //减法
Begin
    //先在 B 前拼接一个 0 然后再取其二补码
    {beforeB,Bcomplement} = ~{1'b0,B} + 1'b1;
    if(Sign)
    begin
        S = A + Bcomplement;
        Zero = (S == 0);
        Overflow=(A[31]==Bcomplement[31]&&A[31]!=S[31]);
        Negative = S[31];
    end
    else
    begin
        {Overflow,S} = {1'b0,A} + {beforeB,Bcomplement};
        Zero = (S == 0);
        Negative = 0;
    end
end
default: ;
endcase
```

其中实现无符号减法时提前在 B 前拼接一个 0，然后再取其二补码是考虑到了 B 为 0 的情况下如果等到将 A 和 B 的二补码相加的时候再拼接一个 0 可能会导致错误地产生溢出信号。

具体实现及注释见 code/CPU/ALU 文件夹中的代码文件，包括***"AddSub.v"***和测试代码***"ALU_tb.v"***(内含多个测试模块)。

㈢仿真结果及分析

仿真结果如下：



图中第一个操作为无符号加，

5（0x00000005）+33(0x00000021)=38(0x00000021)，正确；

第二个操作为有符号加，

$(2^{31}-1)$(0x7fffffff)+$(1-2^{31})$(0x8000001)=0，正确；

第三个操作为无符号减，5-10，发生溢出，可以看到 Overflow 信号为 1，正确；

第四个操作为无符号减，10-0=10，正确；

第五个操作为有符号减，

5(0x00000005)-10(0x0000000a)=-5(0xfffffffb)，

同时可以看到 Negative 信号为 1，正确；

第六个操作为无符号减，50-50=0，同时可以看到 Zero 信号为 1，正确；

第七个操作为无符号加，2^{32}-1+2 发生溢出，可以看到 Overflow 信号为 1，正确。

II、CMP 模块

㈠设计方案

使用 ALUFun[3:1]来区分六个不同的关系运算--EQ、NEQ、LT、LEZ、LTZ 和 GTZ，因为是根据减法运算的结果来产生比较运算的结果(以上六个功能的 ALUFun[0]均为 1)，所以需要分析比较操作和算术运算之间的关系。

①EQ：如果两个操作数相等，那么相减的结果必为 0，即 Zero=1，所以根据这个即可判断 EQ 的输出。

②NEQ：如果两个操作数不等，那么相减的结果就不为零，即 Zero=0，所以根据这个即可判断 NEQ 的输出。

③LT：如果 A<B，当为有符号数时，一般情况下 A-B 的结果的符号位为 1，但是在发生溢出的情况下，A-B 的结果的符号位应为 0（此时应该是 A 负 B 正，符号位为 1 的情况是 A 正 B 负，不符合 A<B)。当为无符号数时，只要结果发生溢出就说明 A<B。故而当发生溢出且 Negative 为 0 或者不发生溢出且 Negative 为 1 时输出为 1。

④LEZ：当 A 为有符号数时，A-0 明显不会发生溢出，A-0 的结果的符号位即为 A 的符号位，所以当结果的符号位为 1 或者 Zero=1（即 A=0）时 A<=0。当 A 为无符号数时，明显不会发生溢出，要使得 A<=0 只可能 Zero=1(即 A=0)。故而当 Negative=1 或者 Zero=1 时输出为 1。

⑤LTZ：当 A 为有符号数时，A-0 明显不会溢出，A-0 的结果的符号位即为 A 的符号位，所以当结果的符号位为 1 时 A<0。当 A 为无符号数时，明显 A>=0，不可能符合条件。故而当 Negative 为 1 时输出为 1。

⑥GTZ：当 A 为有符号数时，A-0 明显不会溢出，A-0 的结果的符号位即为 A 的

符号位，所以当结果的符号位为 0 且结果不为 0(即 Zero=0)时 A>0。当 A 为无符号数时，明显 A>=0，当结果不为 0(Zero=0)时符合条件。故当 Negative 为 0 同时 Zero=0 时输出为 1。

㈡关键代码及文件清单

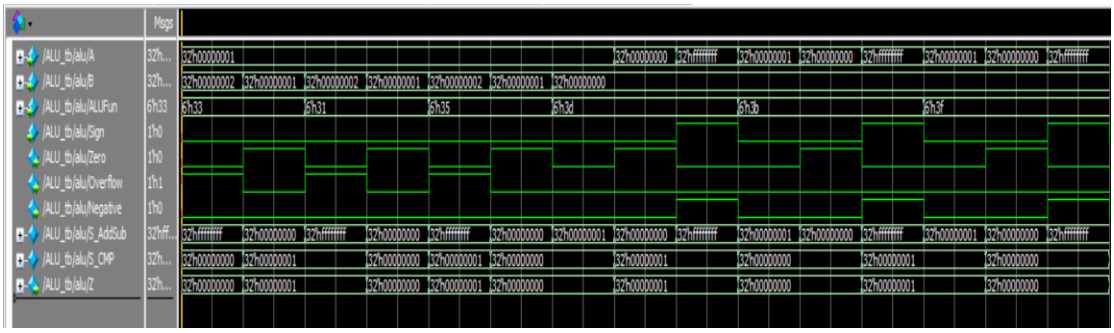
“CMP.v”

```
case (ALUFun3to1)
EQ:
   S <= (Zero == 1);
NEQ:
   S <= (Zero == 0);
LT:
   S <=((Overflow==1&&Negative==0)||(Overflow==0&&Negative==1));
LEZ:
   S <= (Negative == 1 || Zero == 1);
LTZ:
   S <= (Negative == 1);
GTZ:
   S <= (Negative == 0 && Zero == 0);
default: ;
endcase
```

具体实现及注释见 code/CPU/ALU 文件夹中的代码文件，包括***"CMP.v"***和测试代码***"ALU_tb.v"***（内含多个测试模块）。

㈢仿真结果及分析

仿真结果如下：



第 1 个操作为 EQ，A=1，B=2，A!=B,输出为 0;

第 2 个操作为 EQ，A=1，B=1，A==B,输出为 1;

第 3 个操作为 NEQ，A=1，B=2，A!=B,输出为 1;

第 4 个操作为 NEQ，A=1，B=1，A==B,输出为 0;

第 5 个操作为 LT，A=1，B=2，A<b,输出为 1;

第 6 个操作为 LT，A=1，B=1，A==B,输出为 0;

第 7 个操作为 LEZ，A=1，A>0,输出为 0;

第 8 个操作为 LEZ，A=0，输出为 1;

第 9 个操作为 LEZ，A=-1，A<0,输出为 1;

第 10 个操作为 LTZ，A=1，A>0,输出为 0;

第 11 个操作为 LTZ，A=0，输出为 0;

第 12 个操作为 LTZ，A=-1，A<0,输出为 1;

第 13 个操作为 GTZ，A=1，A>0,输出为 1;

第 14 个操作为 GTZ，A=0，输出为 0;

第 15 个操作为 GTZ，A=-1，A<0,输出为 0。

III、Logic 模块

㈠设计方案

使用 ALUFun[3:0]来区分五个不同的逻辑运算--AND、OR、XOR、NOR 和"A"（即 MOV)，直接使用 Verilog 提供的位运算符即可，像 AND 就直接利用按位与(&)，OR 就直接利用按位或（|)，XOR 就直接利用按位异或（^)，NOR 就综合利用按位或（|）与按位取反（~)，"A"(也即 MOV)则是直接赋值。

㈡关键代码及文件清单

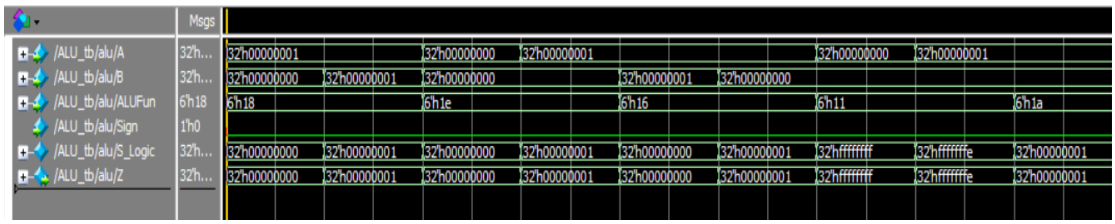
“Logic.v”

```
case(ALUFun3to0)
AND:
    S <= A & B;
OR:
    S <= A | B;
XOR:
    S <= A ^ B;
NOR:
    S <= ~(A | B);
MOV:
    S <= A;
default: ;
endcase
```

具体实现及注释见 code/CPU/ALU 文件夹中的代码文件，包括***"Logic.v"***和测试代码***"ALU_tb.v"***（内含多个测试模块）。

㈢仿真结果及分析

仿真结果如下：



第 1 个操作为 AND,A=0x00000001,B=0x00000000,输出为 0x00000000;

第 2 个操作为 AND,A=0x00000001,B=0x00000001,输出为 0x00000001;

(两个数同个比特位上的数均为 1 输出结果中这个位才为 1，否则为 0)

第 3 个操作为 OR，A=0x00000000,B=0x00000000,输出为 0x00000000;

第 4 个操作为 OR ,A=0x00000001,B=0x00000000,输出为 0x00000001;

(两个数同个比特位上的数均为 0 输出结果中这个位才为 0，否则为 1)

第 5 个操作为 XOR,A=0x00000001,B=0x00000001,输出为 0x00000000;

第 6 个操作为 XOR,A=0x00000001,B=0x00000000,输出为 0x00000001;

(两个数同个比特位上的数不同输出结果中这个位才为 1，否则为 0)

第 7 个操作为 NOR,A=0x00000000,B=0x00000000,输出为 0xffffffff;

第 8 个操作为 NOR,A=0x00000001,B=0x00000000,输出为 0xfffffffe;

(两个数同个比特位上的数均为 0 输出结果中这个位才为 1，否则为 0)

第 9 个操作为"A"（MOV）,A=0x00000001,输出为 0x00000001;

(结果与输入完全相同)

IV、Shift 模块

㈠设计方案

使用 ALUFun[1:0]来区分三种不同的移位运算，由于移动的位数为 A[4: 0]所代表的数，可以拆分为 16、8、4、2、1 位移位这五个子运算的组合，A[4]为 1 代表需要一个 16 位移位，A[3]=1 代表需要一个 8 位移位，A[2]=1 代表需要一个 4 位移位，A[1]=1 代表需要一个 2 位移位，A[0]=1 代表需要一个 1 位移位，最后将这些子运算级联起来即可得到最后的运算结果。

㈡关键代码及文件清单

“Shift.v”

```
SRA:
begin:
   if(B[31])              //negative number
   begin
      S_Shift16 = (A[4])?{16'b1111_1111_1111_1111,B[31:16]}:B;
      S_Shift8=(A[3])?{8'b1111_1111,S_Shift16[31:8]}:S_Shift16;
      S_Shift4 = (A[2])?{4'b1111,S_Shift8[31:4]}:S_Shift8;
      S_Shift2 = (A[1])?{2'b11,S_Shift4[31:2]}:S_Shift4;
```

```
        S = (A[0])?{1'b1,S_Shift2[31:1]}:S_Shift2;
    end
    else                    //postive number
    begin
        S_Shift16 = (A[4])?{16'b0,B[31:16]}:B;
        S_Shift8 = (A[3])?{8'b0,S_Shift16[31:8]}:S_Shift16;
        S_Shift4 = (A[2])?{4'b0,S_Shift8[31:4]}:S_Shift8;
        S_Shift2 = (A[1])?{2'b0,S_Shift4[31:2]}:S_Shift4;
        S = (A[0])?{1'b0,S_Shift2[31:1]}:S_Shift2;
    end
end
```

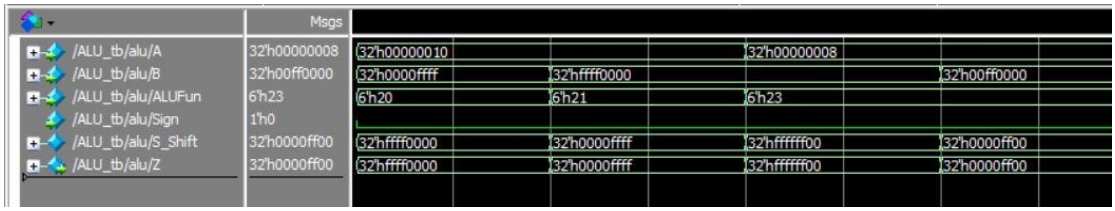
以上是 case 块的一种情形，仅以此情形为例，其他两种情形是类似的。

算术右移需要判断符号位，右移时高位补符号位；而对于逻辑右移和逻辑左移则是高位补零和低位补零。

具体实现及注释见 code/CPU/ALU 文件夹中的代码文件，包括***"Shift.v"***和测试代码***"ALU_tb.v"***（内含多个测试模块）。

㈢仿真结果及分析

仿真结果如下：



第一个操作是 SLL，A=16，B=0x0000ffff，结果为 0xffff0000，即将 B 逻辑左移了 16 位；

第二个操作是 SRL，A=16，B=0xffff0000，结果为 0x0000ffff，即将 B 逻辑右移了 16 位；

第三个操作是 SRA，A=8，B=0xffff0000，结果为 0xffffff00，即将 B 算术右移了 8 位，空出的高位补符号位 1；

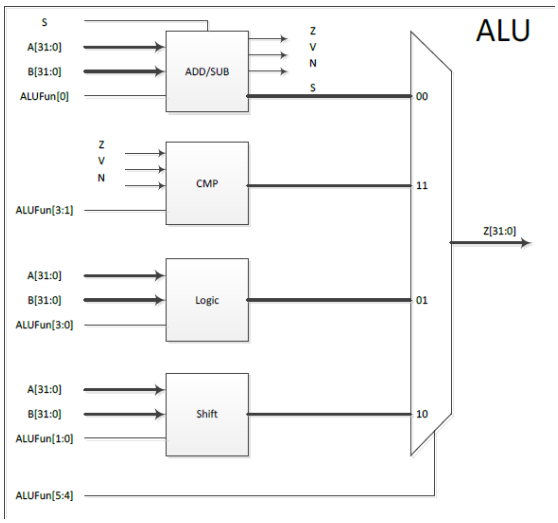
第四个操作是 SRA，A=8，B=0x00ff0000，结果为 0x0000ff00，即将 B 算术右

移了 8 位，空出的高位补符号位 0。

V、组装 ALU 模块

㈠设计方案

如下图所示组装各模块即可得到一个 ALU 模块。



㈡关键代码及文件清单

“ALU.v”

```
AddSub addsub(.A(A),.B(B),.ALUFun0(ALUFun[0]),.Sign(Sign),
              .S(S_AddSub),.Zero(Zero),.Overflow(Overflow),
              .Negative(Negative));
CMP cmp(.Zero(Zero),.Overflow(Overflow),.Negative(Negative),
          .ALUFun3to1(ALUFun[3:1]),.S(S_CMP));
Logic logic(.A(A),.B(B),.ALUFun3to0(ALUFun[3:0]),.S(S_Logic));
Shift shift(.A(A),.B(B),.ALUFun1to0(ALUFun[1:0]),.S(S_Shift));

always @(*)
begin
   case(ALUFun[5:4])
   AddSub:
      Z = S_AddSub;
   CMP:
      Z = S_CMP;
   Logic:
      Z = S_Logic;
```

```
    Shift:
        Z = S_Shift;
    endcase
end
```

具体实现及注释见 code/CPU/ALU 文件夹中的代码文件，包括***"ALU.v"***。

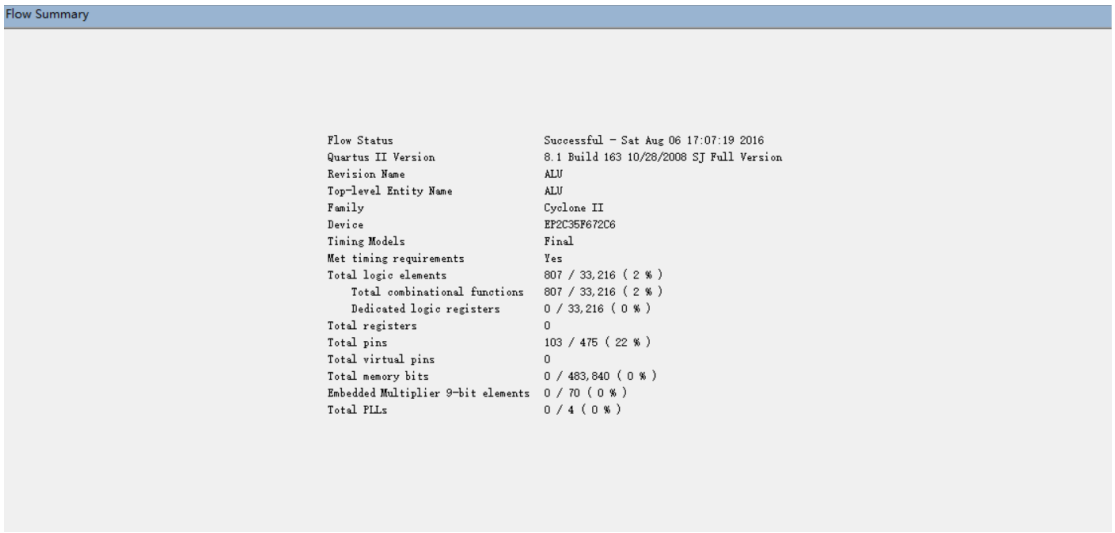
㈢仿真结果及分析

在上面四个模块的仿真结果中已经可以看出 ALU 可以根据控制信号 ALUFun[5:0]

以及 Sign 对输入 A、B 进行正确的操作，输出正确的结果，故而不再重复仿真。

㈣综合情况

面积



时序性能

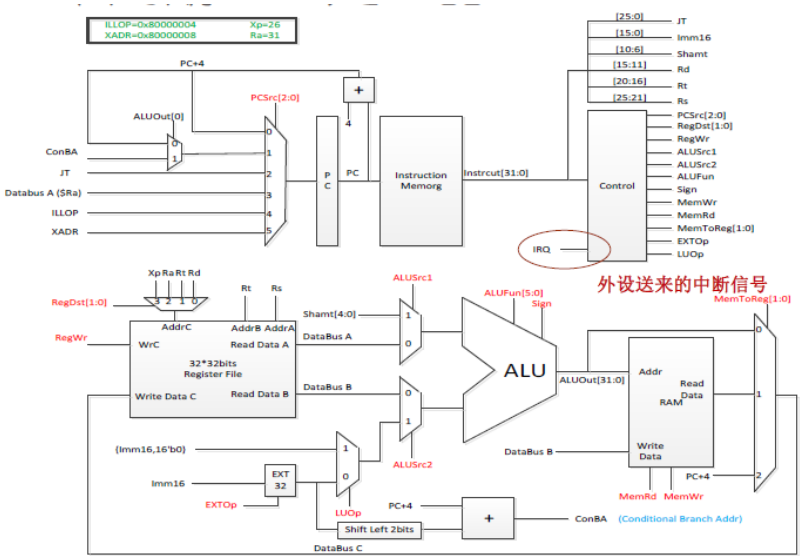
Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To	From Clock	To Clock	Failed Paths
1	Worst-case tsu	N/A	None	15.487 ns	B[6]	CMP:cmp|S[0]	--	ALUFun[1]	0
2	Worst-case tco	N/A	None	13.864 ns	Shift:shift|S[6]	Z[6]	ALUFun[1]	--	0
3	Worst-case tpd	N/A	None	20.093 ns	B[6]	Z[25]	--	--	0
4	Worst-case th	N/A	None	-0.755 ns	B[24]	Logic:logic|S[24]	--	ALUFun[0]	0
5	Total number of failed paths								0

B、单周期 MIPS 处理器（慕思成）

1、单周期 MIPS 处理器的设计

单周期处理器的数据通路



数据通路的连接大致按照上图所示进行，在所有多路选择器的选择中，只有 PC 单元使用模块，其余多路选择器均使用 assign 语句及三目运算符“:”实现。

PC 单元的实现

```
always@(posedge clk or negedge reset)
begin
if(~reset) pc<=32'h00000000;
else
case(PCSrc)
3'd0: pc<=pc_plus_4;
3'd1: pc<=(ALUOut0)?ConBA:pc_plus_4;
3'd2: pc<=JT;
3'd3: pc<=DataBusA;
3'd4: pc<=ILLOP;
3'd5: pc<=XADR;
endcase
end
```

为了保证 pc+4 的最高位与 pc 的最高位相同，需要：

```
assign pc_plus_4[30:0]=pc[30:0]+4;
assign pc_plus_4[31]=pc[31];
```

控制单元的实现

	PCSrc[2:0]	Sign	RegWrite	RegDst[1:0]	MemRead	MemWrite	MemtoReg[1:0]	ALUSrc1	ALUSrc2	EXTOp	LUOp	ALUFun
lw	0	0	1	0	1	0	1	0	1	1	0	000000
sw	0	0	0	×	0	1	×	0	1	1	0	000000
lui	0	0	1	0	0	0	0	0	1	×	1	000000
add	0	1	1	1	0	0	0	0	0	×	×	000000
addu	0	0	1	1	0	0	0	0	0	×	×	000000
sub	0	1	1	1	0	0	0	0	0	×	×	000001
subu	0	0	1	1	0	0	0	0	0	×	×	000001
addi	0	1	1	0	0	0	0	0	1	1	0	000000
addiu	0	0	1	0	0	0	0	0	1	1	0	000000
and	0	1	1	1	0	0	0	0	0	×	×	011000
or	0	1	1	1	0	0	0	0	0	×	×	011110
xor	0	1	1	1	0	0	0	0	0	×	×	010110
nor	0	1	1	1	0	0	0	0	0	×	×	010001
andi	0	1	1	0	0	0	0	0	1	1	0	011000
sll	0	0	1	1	0	0	0	1	0	×	×	100000
srl	0	0	1	1	0	0	0	1	0	×	×	100001
sra	0	0	1	1	0	0	0	1	0	×	×	100011
slt	0	1	1	1	0	0	0	0	0	×	×	110101
sltu	0	0	1	1	0	0	0	0	0	×	×	110101
slti	0	1	1	1	0	0	0	0	1	1	0	110101
sltiu	0	0	1	1	0	0	0	0	1	0	0	110101
beq	0	1	0	×	0	0	×	0	0	1	0	110011
j	1	1	0	×	0	0	×	×	×	×	×	000000
jal	1	1	1	2	0	0	2	×	×	×	×	000000
jr	2	0	0	×	0	0	×	×	×	×	×	000000
jalr	2	0	1	1	0	0	2	×	×	×	×	000000
bltz	1	1	0	1	0	0	0	0	0	1	0	110101
bne	1	1	0	1	0	0	0	0	0	1	0	110001
blez	1	1	0	1	0	0	0	0	0	1	0	111101
bgtz	1	1	0	1	0	0	0	0	0	1	0	111111

控制单元的信号设置参考上述表格，为了便于结构清晰，故按照每一条指令设置多个信号分块写出，don’t care 的信号不再写出。当遇到异常或中断时，

处理如下：

中断：

```
if(IRQ==1)//interrupt
begin
RegWr=1;
PCSrc=3'b100;
RegDst=2'b11;
MemToReg=2'b11;
end
```

在主程序中要判断 pc[31]是否为 1

```
assign IRQ=pc[31]?1'b0:irqout;
```

异常指令：

```
default:
begin   //Exception
RegWr=1;
PCSrc=3'b101;
RegDst=2'b11;
MemToReg=2'b11;
```

由于对于中断的处理仍需要执行当前指令，此时控制单元只改变 PCSrc,RegWr,RegDst,MemToReg 的值，其他控制信号不变，故需要在设置完当前控制信号后再判断是否中断。

寄存器的读取

修改 regfile.v 模块，以保证当一个周期中读寄存器与写寄存器编号相同时，读出的数据为写入的数据，即实现伪先读后取。

```
assign data1=(addr1==5'b0)?32'b0:(addr1==addr3)?data3:RF_DATA[addr1];
assign data2=(addr2==5'b0)?32'b0:(addr2==addr3)?data3:RF_DATA[addr2];
```

DataBusC 的选取

当跳转或分支指令与中断在同一个周期发生时，将跳转或分支的地址而不是 pc+4 写入寄存器中。

```
assign pc_next=((Instruct[31:26]==6'h01||Instruct[31:26]==6'h04||Instruct[31:26]==6'h05||Instruct[31:26]==6'h06||Instruct[31:26]==6'h07)
&&ALUOut[0])?ConBA:(Instruct[31:26]==6'h02)?JT:(Instruct[31:26]==0&&(Instruct[5:0]==6'h8||Instruct[5:0]==6'h9))?DataBusA:pc_plus_4;
assign DataBusC=(MemToReg==2'b00)? ALUOut:(MemToReg==2'b01)?ReadData:pc_next;
```

数据通路的搭建

```
//EXT
wire [31:0] EXT32;
assign EXT32=EXTOp?{{16{Instruct[15]}},Instruct[15:0]}:{16'h0000,Instruct[15:0]};

//LU
wire [31:0] LU32;
assign LU32=LUOp?{Instruct[15:0],16'h0000}:EXT32;

wire [31:0] A,B;
assign A=ALUSrc1?{27'h0,Instruct[10:6]}:DataBusA;
assign B=ALUSrc2?LU32:DataBusB;

//ConBA;
wire [31:0] pc_plus_4,ConBA;
assign pc_plus_4[30:0]=pc[30:0]+4;
assign pc_plus_4[31]=pc[31];
assign ConBA=pc_plus_4+{EXT32[29:0],2'b00};

//JT
wire [31:0] JT;
assign JT={pc_plus_4[31:28],Instruct[25:0],2'b00};
//ALU
wire[31:0] ALUOut;
ALU ALU1(.A(A),.B(B),.ALUFun(ALUFun),.Sign(Sign),.Z(ALUOut));
```

外存的访问

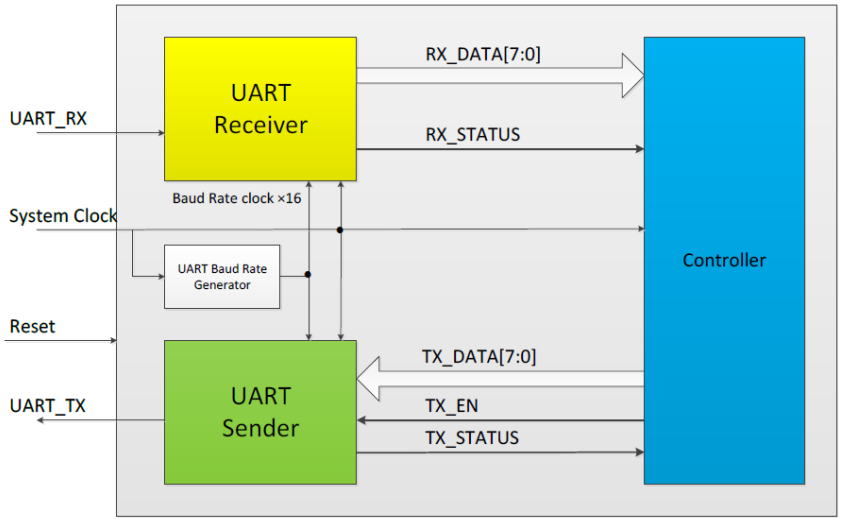
```
//ROM
ROM ROM1(.addr(pc),.data(Instruct));
//DataMem
wire [31:0] Mem_ReadData;
DataMem RAM1(.reset(reset),.clk(clk),.rd(MemRd),.wr(MemWr),.addr(ALUOut),.wdata(DataBusB),.rdata(Mem_ReadData));
//Peripheral
wire [31:0] Peripheral_ReadData;
Peripheral Ph1(.reset(reset),.clk(clk),.rd(MemRd),.wr(MemWr),.addr(ALUOut),.wdata(DataBusB),.
rdata(Peripheral_ReadData),.led(led),.switch(switch),.digi(digi),.irqout(irqout));
//UART
wire [31:0] UART_ReadData;
UART UART1(.UART_RX(UART_RX),.sys_clk(sysclk),.clk(clk),.reset(reset),.UART_TX(UART_TX),.MemRd(MemRd)
,.MemWr(MemWr),.WriteData(DataBusB),.ReadData(UART_ReadData),.Addr(ALUOut));
assign ReadData=(ALUOut<32'h40000000)? Mem_ReadData:(ALUOut<32'h40000018)?Peripheral_ReadData:UART_ReadData;
```

其中访问外存读取的数据可能来自数据存储器 RAM、外设（开关、LED、数码管）和 UART 外设三种路径，需要根据当前地址 addr（即 ALU 的输出）进行选择。其中，UART 外设通过轮询方式来访问，而 LED、开关、数码管通过中断方式来访问。

外设地址如下：

地址范围（字节地址）	功能	描述
0x4000000C	外部LEDs	0bit：LED 0 …… 7bit：LED 7
0x40000010	外部SWITCH	0bit：Switch 0 …… 7bit：Switch7
0x40000014	七段数码管	0bit：CA 1bit：CB …… 7bit：DP 8bit：AN0 9bit：AN1 10bit：AN2 11bit：AN3

UART 外设的实现



UART 在春季学期所设计的 UART 的基础上通过修改控制模块 controller 实现，需要满足以下要求：

地址范围	功能	备注
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

发送完毕信号的判断以及 UART 读取：

```
always@(*)begin
TX_DATA=UART_TXD;
end
reg temp=0;
always@(posedge clk or negedge reset)
begin
if(~reset) temp<=1;
else temp<=TX_STATUS;
end
wire OVER;
assign OVER=(~temp)&TX_STATUS;
always@(*)
begin
if(MemRd)
begin
case(Addr)
32'h40000018: ReadData={24'h0,UART_TXD};
32'h4000001C: ReadData={24'h0,UART_RXD};
32'h40000020: ReadData={27'h0,UART_CON};
default: ReadData=32'h00000000;
endcase
end
else ReadData=32'h00000000;
end
```

UART 外存地址中数据的设置：

```
always@(posedge clk or negedge reset)
begin
if(~reset) begin
TX_EN<=0;
UART_TXD<=8'b0;
UART_CON<=5'b0;
UART_RXD<=8'b0;
end
else begin
if(RX_STATUS)begin
UART_RXD=RX_DATA;
if(UART_CON[1]) UART_CON[3]<=1;
end
if(OVER&&UART_CON[0]) UART_CON[2]<=1;
if(~TX_STATUS) UART_CON[4]<=1;
else UART_CON[4]<=0;
if(MemRd&&Addr==32'h40000020) begin
UART_CON[2]<=0;
UART_CON[3]<=0;
end
if(MemWr)
begin
case(Addr)
32'h40000018:begin
UART_TXD<=WriteData[7:0];
TX_EN<=1;
end
32'h40000020: UART_CON[1:0]<=WriteData[1:0];
default:;
endcase
end
if(TX_EN) TX_EN<=0;
end
end
```

2、汇编程序的实现

汇编程序主要实现外设初始化、UART 轮询、计算最大公约数、定时器中断服务（数码管的扫描模式译码)、异常处理 5 个部分。其中 UART 访问采用轮询方式，而其他外设访问采用中断方式。

（1）外设初始化

设置各个外设中的状态，并将常用数值（0xffffffff、0x40000000）记录在寄存器中以便于后续的使用。在此步骤中，通过设置 TH 的数值可以调整定时器中断周期。

```
j Main
j Interrupt
j Exception

Main:
add $sp,$zero,$zero    #sp=0
lui $s7, 16384         #s7=0x40000000  address of peripheral
nor $s6,$zero,$zero    #s6=0xffffffff
sw $zero,12($s7)       #LED=0
sw $s6,20($s7)         #digi=0xffffffff
sw $zero,8($s7)        #TCON=0
addi $t0,$s6,-25000
addi $t0,$t0,-25000
sw $t0,0($s7)          #set the period
sw $s6,4($s7)          #TL=0xffffffff
addi $t0,$zero,3
sw $t0,8($s7)          #TCON=3
sw $t0,32($s7)
add $s0,$zero,$zero    #s0:UART Recieve?
sll $zero,$zero,0
```

（2）UART 轮询

通过对 UART_CON 中接收状态的读取以判断是否收到数据，反复轮询，并记录收到数据的个数，将接收的数据存入寄存器。当已经收到一个数据后在此收到数据时，将此数据存到另一寄存器，并进入计算最大公约数模块。计算完最大公约时候，发送结果并记录到 LED 中，接收数据状态重置，进入下一次轮询。

```
Polling:                #UART polling
lw $t0,32($s7)          #Read UART_CON
addi $t1,$zero,8
and $t2,$t1,$t0         #Read UART_CON[3]
beq $t2,$zero,Polling #UART_[3]==1,Recieve
bne $s0,$zero,MemRd
lw $a0,28($s7)          #a0=UART_RXD
addi $s0,$zero,1
j Polling
MemRd:
lw $a1,28($s7)
jal GCD
add $s0,$zero,$zero
sw $v0,12($s7)
Sending:
lw $t0,32($s7)
addi $t1,$zero,16
and $t2,$t0,$t1
srl $t2,$t2,4
bne $t2,$zero,Sending
sw $v0,24($s7)

j Polling
```

(3) 计算最大公约数

采用更相减损术计算最大公约数，即当两个参数相同时，即为最大公约数，否则反复用大数减去小数，直至两数相同。这里需要注意当被减数小于减数时需要交换两参数。

```
GCD:
add $t0,$a0,$zero
add $t1,$a1,$zero
While:
beq $t0,$t1,GCD_Exit
sub $t2,$t0,$t1
bgtz $t2,Sub
add $t2,$t0,$zero
add $t0,$t1,$zero
add $t1,$t2,$zero
Sub:
sub $t0,$t0,$t1
j While

GCD_Exit:
add $v0,$t0,$zero
jr $ra
```

（4）定时器中断服务

定时器中断软件服务程序（ISR）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. *恢复现场;*
v. *使能中断，TCON的1bit置1，TCON |= 0x00000002;*
vi. *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）.*

中断程序分为两部分，一是读出当前数码管状态及两参数的值，并计算下一数码管的数值对应的状态；二是将下一状态进行译码，并将对应的结果扫描显示在数码管上。

（5）异常处理

本次实验中，对于异常处理只进行 nop 一周期，并返回$26 中的地址。

```
Exception:
sll $zero,$zero,0
jr $k0
```

C、编译器（陈佳榕）

㈠编程语言的选择

由于我想做成一款具有图形界面的应用程序，所以选择了 JAVA 作为此次编写 MIPS 汇编器的编程语言。不仅如此，因为汇编器所要处理的汇编程序源文件的可能有很多千奇百怪的排版方式，所以需要汇编器能正确识别出正确的 MIPS 代码，同时兼具识别注释、便签的功能，这个时候 JAVA 本身提供的用于处理字符串的函数就很有用了，这也是我选择 JAVA 的另一个重要原因。

㈡设计思路

⑴由于源代码文件中可能有注释，还会有各种空格（用于排版之类的），所以第

一步所要做的就是将这些对于代码正确性不会产生影响的注释以及空格去除,这里我还用到了一些简单的正则表达式去匹配注释和空格行。

⑵对汇编程序进行编译的过程可以看成是将特定字符串映射成特定机器码的过程，为了实现这种映射过程，我选择了 HashMap 这种数据结构，具体有将指令名称映射到其 Opcode 和指令类型，将 R 型指令名称映射到其 funct，将寄存器号或寄存器名称映射到对于的寄存器地址。

⑶正式进行编译之前先处理标签，将标签翻译成对应的指令行数，同样采用 HashMap 存放这些数据，将标签名称映射到其对应的指令行数。

⑷对指令进行分类,通过观察可以看到有些指令的指令和机器码具有相同的格式,可以根据这个将指令分为以下几组:

①nop

机器码就是 32 位的 0;

②lw,sw

指令格式为 lw rt,address

机器码格式为

Opcode	rs	rt	Offset

③lui

指令格式为 lui rt,imm

机器码格式为

Opcode	0	rt	imm

④add,addu,sub,subu,and,or,xor,nor,slt

指令格式为 add rd,rs,rt

机器码格式为

Opcode	rs	rt	rd	0	funct

⑤addi,addiu,andi,slti,sltiu

指令格式为 addi rt,rs,imm

机器码格式为

Opcode	rs	rt	imm

⑥sll,srl,sra

指令格式为 sll rd,rt,shamt

机器码格式为

Opcode	rs	rt	rd	shamt	0

⑦beq,bne

指令格式为 beq rs,rt,label

机器码格式为

Opcode	rs	rt	Offset

⑧blez,bgtz,bltz

指令格式为 blez rs,label

机器码格式为

Opcode	rs	0	Offset

⑨j,jal

指令格式为 j target

机器码格式为

Opcode	target

⑩jr

指令格式为 jr rs

机器码格式为

Opcode	rs	0	funct

⑪jalr

指令格式为 jalr rd,rs

机器码格式为

Opcode	rs	0	rd	0	funct

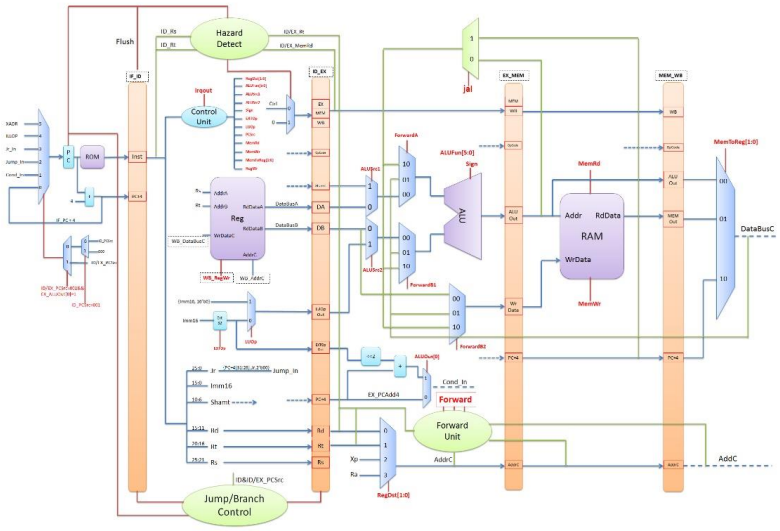
最后根据分组对指令进行相应的拆分，例如识别出是第 4 组的指令后，将除去指令名称后剩下的 rd,rs,rt 根据逗号分为 3 个部分，然后便可以利用前面的 HashMap 将这三个部分映射到相应的机器码部分，完成编译。

㈡具体代码及可执行程序

由于代码量比较多,所以将代码放在了附件里(见code/CPU/compiler 文件夹)，包括了用于实现图形界面的 Frame.java,用于实现编译功能的 Compiler.java 以及实现主程序入口的 Main.java。附件还包括了可执行的 jar 包，解压 Compiler.rar 之后直接双击 CompilerForMIPS.jar 即可运行。

D、流水线 MIPS 处理器（慕思成）

流水线 MIPS 处理器的设计



流水线数据通路的构建

流水线处理器的数据通路需要在单周期数据通路的基础上，在各个阶段之间加入流水线间寄存器，以存储在之后的阶段会用到的控制信号及数据。另外由于冒险与转发的存在，需要设计冒险检测及转发模块并对数据通路的相应部分进行调整；此外，还需要注意 PC 的取值，DataBusC 的选取中与单周期时的区别。

流水线间寄存器的实现

（1） IF/ID 寄存器

IF/ID 寄存器中存储 pc+4 和从指令存储器中取出指令，其输入中有 IF_ID_Flush 和 IF_ID_Write 两个信号，以控制寄存器的气泡和清除操作。

（2） ID/EX 寄存器

ID 阶段进行指令的译码以及寄存器的读取，故在 ID/EX 寄存器中需要存储几乎全部的控制信号，以及取出的寄存器的操作数，还有在 EX 阶段中可能会用到的立即数扩充、位移量等。此外 ID/EX 寄存器中还应存储 Rt、Rs、Rd 存储器的值，以及指令 Opcode，以供冒险检测及转发单元检测冒险的存在以及做出相应转发；pc+4 的值亦需要一直保留。

（3） EX/MEM 寄存器

EX/MEM 寄存器中存储 ALU 运算结果、存储器写数据、寄存器写会的地址、pc+4、Opcode 以及之后阶段会用到的控制信号。

（4） MEM/WB 寄存器

MEM/WB 存储器需要存储 ALU 的结果，寄存器写会数据，写会地址，pc+4 以及 WB 阶段的控制信号，以实现寄存器写回。

数据通路的修改

由于冒险与转发的存在，需要对数据通路进行修改，将 ALU 的输入、存储器的写数据进行选择。

数据冒险产生的修改：

```
//ALU
wire [31:0] A,B;
wire [31:0] ALU_in1,ALU_in2;
wire [31:0] jal_out;
assign ALU_in1=ID_EX_ALUSrc1_out?{27'h0,ID_EX_Shamt_out}:ID_EX_DataBusA_out;
assign ALU_in2=ID_EX_ALUSrc2_out?ID_EX_LU32_out:ID_EX_DataBusB_out;
assign A=(ForwardA==2'b10)?jal_out:(ForwardA==2'b01)?DataBusC:ALU_in1;
assign B=(ForwardB1==2'b10)?jal_out:(ForwardB1==2'b01)?DataBusC:ALU_in2;
```

lw 型指令产生的冒险：

```
//WriteData
wire [31:0] WriteData;
assign WriteData=(ForwardB2==2'b10)?jal_out:(ForwardB2==2'b01)?DataBusC:ID_EX_DataBusB_out;
```

当中断与分支指令处于同一周期时，需要重新执行该指令：

```
//WB
assign DataBusC=(MEM_WB_MemToReg_out==2'b00)? MEM_WB_ALUOut_out:(MEM_WB_MemToReg_out==2'b01)?MEM_WB_ReadData_out:
(MEM_WB_MemToReg_out==2'b10)?MEM_WB_pc_plus_4_out:(EX_MEM_pc_plus_4_out-8);
```

JR 指令产生的冒险，以及在流水线下 PCSrc 信号的选择：

```
//PC
wire [31:0] PC_JR_in;
wire [2:0] PCSRC;
assign PCSRC=(ID_EX_PCSrc_out==3'b001&&ALUOut[0]==1'b1)?ID_EX_PCSrc_out:(PCSrc==3'b001)?3'b000:PCSrc;
assign PC_JR_in=(ForwardJR==2'b11)?DataBusC:(ForwardJR==2'b10)?jal_out:(ForwardJR==2'b01)?ALUOut:DataBusA;
PC PC1(.pc(pc),.reset(reset),.clk(clk),.PCSrc(PCSRC),.ConBA(ConBA),.JT(JT),.DataBusA(PC_JR_in),.ALUOut0(ALUOut[0]),.PC_Write(PC_Write));
```

外设的实现

所有外设与单周期时实现相同。

E、冒险检测&转发单元（陈佳榕）

㈠设计方案

考虑如下的几种冒险检测及转发:

1.数据关联冒险

这里采用完全的 forwarding 电路解决数据关联问题。

数据冒险分为 EX 冒险和 MEM 冒险两种情况

当冒险检测条件均满足时,产生相应的转发信号,这里的转发信号包括 ForwardA、ForwardB1 和 ForwardB2。

⑴EX 冒险的条件为:

如果前一条指令要写寄存器堆而且要写的寄存器号与 A 或 B ALU 输入的要读的寄存器号一致，只要不是寄存器 0，那么就调用多路复用选择器从流水线寄存器 EX/MEM 中读取数值。

⑵MEM 冒险的条件为:

①这条指令要写入寄存器

②写入的不是寄存器 0

③目标寄存器号 == 源寄存器号

④这两条指令之间的指令要么所写入寄存器与 ALU 输入不符要么中间的指令不写入寄存器

2.Load-use 类竞争冒险

这里采取阻塞一个周期+Forwarding 的方法解决

首先检查是否是 load 指令，然后检查 EX 阶段指令的目标寄存器号是否与 ID 阶段的两个源寄存器号之一是一样的（当然不能是寄存器 0）。当判断成立时

时，要产生相应的信号使得流水线阻塞一个周期，然后就可以由前面的数据冒险转发单元处理了。

3.分支冒险

这里在 EX 阶段判断，在分支发生时刻取消 ID 和 IF 阶段的两条指令。

判断的条件为指令是分支指令且分支成功，判断条件成立则需要清除 ID 和 IF 阶段的两条指令，进行跳转，否则继续执行相应指令。

4.J 类指令引发的冒险

这里在 ID 阶段判断，并取消 IF 阶段指令。

这里还可能需要转发，因为 Jr 指令在 ID 阶段就要得到跳转的目标地址，如果存有跳转目标地址的寄存器是前面指令的目标寄存器，这个时候就需要转发了。所以需要判断存有跳转目标地址的寄存器号是否与 EX、MEM、WB 阶段的目标寄存器号相同，然后再产生相应的转发信号 ForwardJR 以选择相应的转发源。这里比较特殊的是当存有跳转目标地址的寄存器号是否与 MEM 阶段的目标寄存器号相同时需要判断 MEM 阶段的指令是否为 jal 指令，如果是则需要转发 PC+4（因为 jal 指令的目标寄存器要存的是 PC+4），否则转发 ALU 的输出。

㈡关键代码及文件清单

1.数据关联冒险

⑴EX 冒险

① ForwardA

```
if(EX_MEM_RegWrite&&(EX_MEM_AddrC!=0)&&(EX_MEM_AddrC==ID_EX_Rs)
&&(ID_EX_ALUSrc1 == 0))
   ForwardA <= 2'b10;
```

② ForwardB1 和 ForwardB2

```
if(EX_MEM_RegWrite&&(EX_MEM_AddrC!=0)&&(EX_MEM_AddrC==ID_EX_Rt
))
   begin
      ForwardB1 <= (ID_EX_ALUSrc2)?2'b00:2'b10;
      ForwardB2 <= 2'b10;
   end
```

⑵MEM 冒险

① ForwardA

```
if(MEM_WB_RegWrite&&(MEM_WB_AddrC!=0)&&(EX_MEM_AddrC!=ID_EX_Rs
||~EX_MEM_RegWrite)&&(MEM_WB_AddrC==ID_EX_Rs)
&&(ID_EX_ALUSrc1==0))
   ForwardA <= 2'b01;
```

② ForwardB1 和 ForwardB2

```
if(MEM_WB_RegWrite&&(MEM_WB_AddrC!=0)&&(EX_MEM_AddrC!=ID_EX_Rt
||~EX_MEM_RegWrite)&&MEM_WB_AddrC==ID_EX_Rt)
   begin
      ForwardB1 <= (ID_EX_ALUSrc2)?2'b00:2'b01;
      ForwardB2 <= 2'b01;
   end
```

2. Load-use 类竞争冒险

```
if (ID_EX_MemRead&&((ID_EX_Rt==IF_ID_Rs&&IF_ID_Rs!=5'd0)
||(ID_EX_Rt==IF_ID_Rt&&IF_ID_Rt!=5'd0)))
   begin
      IF_ID_Flush_load <= 0;
      ID_EX_Flush_load <= 1;
      PC_Write_load <= 0;
      IF_ID_Write_load <= 0;
   end
```

3. 分支冒险

```
if((ID_EX_PCSrc == 3'b001) && EX_ALUOut0)
```

```
   begin
      IF_ID_Flush_branch <= 1;
      ID_EX_Flush_branch <= 1;
      PC_Write_branch <= 1;
      IF_ID_Write_branch <= 1;
   end
```

4.J 类指令引发的冒险

```
if(ID_PCSrc==3'd2||ID_PCSrc==3'd3||ID_PCSrc==3'd4
||ID_PCSrc==3'd5)
   begin
      IF_ID_Flush_jump <= 1;
      ID_EX_Flush_jump <= 0;
      PC_Write_jump <= 1;
      IF_ID_Write_jump <= 1;
   end

if((ID_PCSrc==3'd3)&&(ID_Rs==EX_AddrC)&&(EX_AddrC!=0)&&
(ID_EX_RegWrite))
   ForwardJR <= 2'b01;
else if((ID_PCSrc==3'd3)&&(ID_Rs==EX_MEM_AddrC)&&
(EX_MEM_AddrC != 0)&&EX_MEM_RegWrite&&(ID_Rs!=EX_AddrC))
   ForwardJR <= 2'b10;
else
if((ID_PCSrc==3'd3)&&(ID_Rs==MEM_WB_AddrC)&&(MEM_WB_AddrC!=0)
&&MEM_WB_RegWrite&&(ID_Rs!=EX_AddrC)&&(ID_Rs!=EX_MEM_AddrC))
   ForwardJR <= 2'b11;
else
   ForwardJR <= 2'b00;
```

具体实现及注释见 code/CPU/CPU_pipeline 文件夹中的代码文件

"Hazard_Forwarding.v"。

F、关键代码及文件清单（慕思成）

（1） 单周期文件清单

文件名	描述
Logic.v	逻辑运算单元
CMP.v	关系运算单元
AddSub.v	算术运算单元
Shift.v	移位运算单元
ALU.v	ALU 顶层单元
PC.v	程序计数器单元
Control.v	控制信号发生模块
regfile.v	寄存器模块
rom.v	指令存储器
DataMem.v	数据存储器
Peripheral.v	外设（开关、LED、数码管）
UART_Receiver.v	串口接收器
UART_Sender.v	串口发送器
Controller.v	串口控制模块
Baud_Rate_Generator.v	波特率发生器
UART.v	串口收发器顶层模块
digitube_scan.v	数码管扫描显示模块
CPU_single.v	单周期处理器顶层模块
CPU_singletb.v	单周期处理器测试平台
UARTtb.v	串口测试平台
code.asm	汇编代码

CPU_single.qsf	引脚绑定文件
CPU_single Clock Hold_ 'sysclk'.txt	时序分析文件
CPU_single Clock Setup_ 'sysclk'.txt	
CPU_single tco.txt	
CPU_single tsu.txt	

（2） 流水线文件清单

文件名	描述
Logic.v	逻辑运算单元
CMP.v	关系运算单元
AddSub.v	算术运算单元
Shift.v	移位运算单元
ALU.v	ALU 顶层单元
PC.v	程序计数器单元
Control.v	控制信号发生模块
regfile.v	寄存器模块
rom.v	指令存储器
DataMem.v	数据存储器
Peripheral.v	外设（开关、LED、数码管）
UART_Receiver.v	串口接收器
UART_Sender.v	串口发送器
Controller.v	串口控制模块
Baud_Rate_Generator.v	波特率发生器

UART.v	串口收发器顶层模块
digitube_scan.v	数码管扫描显示模块
IF_ID.v	IF/ID 寄存器
ID_EX.v	ID/EX 寄存器
EX_MEM.v	EX/MEM 寄存器
MEM_WB.v	MEM/WB 寄存器
Hazard_Forwarding.v	冒险探测及转发单元
CPU_pipeline.v	单周期处理器顶层模块
CPU_pipelinetb.v	单周期处理器测试平台
UARTtb.v	串口测试平台
code.asm	汇编代码
CPU_pipeline.qsf	引脚绑定文件
CPU_pipeline Clock Hold_ 'sysclk'.txt	时序分析
CPU_pipeline Clock Setup_ 'sysclk'.txt	
CPU_pipeline tco.txt	
CPU_pipeline tsu.txt	

G、仿真结果及分析（慕思成）

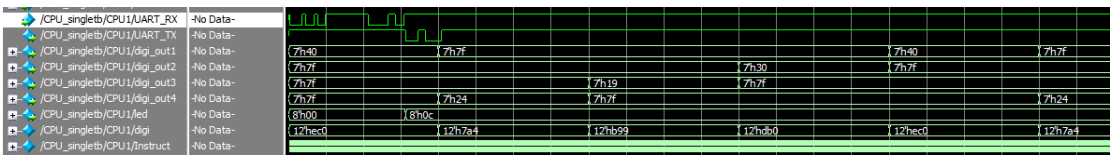
(1) 单周期处理器仿真

利用“CPU_singletb.v”文件进行仿真

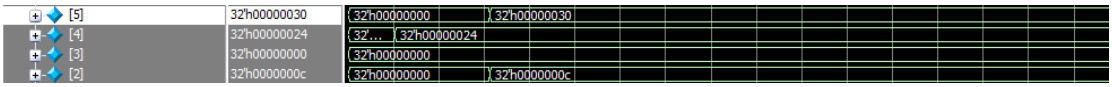
```
`timescale 1ns/1ns
module CPU_pipelinetb;
reg reset,sysclk,UART_RX;
wire [7:0] led;
reg [7:0] switch;
wire [6:0] digi_out1,digi_out2,digi_out3,digi_out4;
wire UART_TX;
reg [7:0] a0,a1;
CPU_pipeline CPU1(sysclk,reset,led,switch,digi_out1,digi_out2,digi_out3,digi_out4,UART_RX,UART_TX);
initial begin
reset=1;
sysclk=0;
UART_RX=1;
a0=8'b00100100;//32
a1=8'b00110000;//48
UART_RX=1;
#23 reset=0;
#23 reset=1;
end
wire sam_clk;
reg Baud_clk;
reg [5:0]count;
reg [8:0] Count;
initial begin
Baud_clk<=0;
count<=0;
Count<=0;
end
Baud_Rate_Generator B1(.sys_clk(sysclk),.reset(reset),.sam_clk(sam_clk));
always@(posedge sam_clk or negedge reset)
begin
if(~reset) begin
Baud_clk<=0;
count<=0;
end
else begin
count<=count+1;
if(count==7) begin
Baud_clk<=~Baud_clk;
count<=0;
end
end
end
always @(posedge Baud_clk) begin
Count <= Count + 1;
if((Count >=10&&Count<20)||Count>=30) begin
UART_RX <= 1;
if(Count == 40)
Count <= 0;
end
else if(Count == 0)
UART_RX <= 0;
else if(Count == 9)
UART_RX <= 1;
else if(Count>0&&Count<9)
UART_RX <= a0[Count - 1];
else if(Count == 20)
UART_RX <= 0;
else if(Count == 29)
UART_RX <= 1;
else if(Count>20&&Count<29)
UART_RX <= a1[Count - 21];
end

always #50 sysclk<=~sysclk;
endmodule
```

此程序可以循环产生 0x24 和 0x30 两个数据流，其最大公约数为 0x0c，汇编代码 code.asm 经编译器编译后存入 rom.v 中，仿真结果如下：



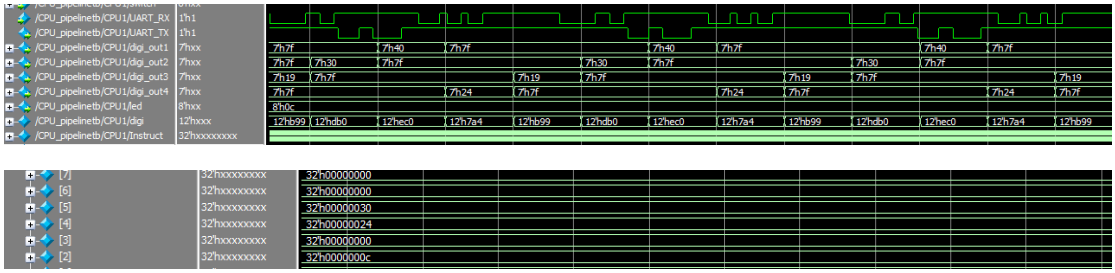
可以看到，当接收到 2 个完整的数据后，系统发送了一个数据，并向 LED 中输出结果 0x0c，与理论结果一致，经查表可知数码管上显示的数字为 2430，与理论一致。



由图可知，$a0,$a1 寄存器中存储了 2 个操作数 0x24 和 0x30，$v0 中存储了结果 0x0c，与理论结果一致，说明设计符合理论预期。

（2）流水线处理器

利用“CPU_pipeline.tb”进行仿真，该模块与单周期时模块一致，rom.v 中指令与单周期时相同。仿真结果如下：



结果与单周期时相同，说明设计符合理论预期。

H、综合情况（慕思成）

（1）单周期处理器

Flow Summary

Flow Status	Successful - Sun Aug 07 06:13:38 2016
Quartus II Version	8.1 Build 163 10/28/2008 SJ Full Version
Revision Name	CPU_single
Top-level Entity Name	CPU_single
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Met timing requirements	No
Total logic elements	13,261 / 33,216 (40 %)
Total combinational functions	9,104 / 33,216 (27 %)
Dedicated logic registers	9,368 / 33,216 (28 %)
Total registers	9368
Total pins	48 / 475 (10 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To	From Clock	To Clock	Failed Paths
1	Worst-case tsu	N/A	None	1.154 ns	switch[1]	RegFile:RegFile1|RF_DATA[1][1]	--	sysclk	0
2	Worst-case tco	N/A	None	22.455 ns	Peripheral:Ph1|digi[10]	digi_out2[2]	sysclk	--	0
3	Worst-case th	N/A	None	8.683 ns	switch[6]	RegFile:RegFile1|RF_DATA[4][6]	--	sysclk	0
4	Clock Setup: 'sysclk'	22.088 ns	10.00 MHz (period = 100.000 ns)	17.91 MHz (period = 55.824 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[28][10]	sysclk	sysclk	0
5	Clock Hold: 'sysclk'	-8.058 ns	10.00 MHz (period = 100.000 ns)	N/A	Control:Control1|ALUFun[3]	ALU:ALU1|CMP:cmp|S[0]	sysclk	sysclk	343
6	Total number of failed paths								343

Clock Setup: 'sysclk'

	Slack	Actual fmax (period)	From	To	From Clock	To Clock	Required Setup Relationship	Required Longest P2P Time	Actual Longest P2P Time
1	22.088 ns	17.91 MHz (period = 55.824 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[28][10]	sysclk	sysclk	50.000 ns	47.726 ns	25.638 ns
2	22.089 ns	17.91 MHz (period = 55.822 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[14][10]	sysclk	sysclk	50.000 ns	47.726 ns	25.637 ns
3	22.504 ns	18.18 MHz (period = 54.992 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[1][10]	sysclk	sysclk	50.000 ns	47.720 ns	25.216 ns
4	22.505 ns	18.19 MHz (period = 54.990 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[5][10]	sysclk	sysclk	50.000 ns	47.720 ns	25.215 ns
5	22.508 ns	18.19 MHz (period = 54.984 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[25][10]	sysclk	sysclk	50.000 ns	47.718 ns	25.210 ns
6	22.511 ns	18.19 MHz (period = 54.978 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[27][10]	sysclk	sysclk	50.000 ns	47.718 ns	25.207 ns
7	22.586 ns	18.24 MHz (period = 54.828 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[27][14]	sysclk	sysclk	50.000 ns	47.718 ns	25.132 ns
8	22.591 ns	18.24 MHz (period = 54.818 ns)	Control:Control1|ALUSrc2	RegFile:RegFile1|RF_DATA[27][14]	sysclk	sysclk	50.000 ns	49.273 ns	26.682 ns
9	22.695 ns	18.31 MHz (period = 54.610 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[17][10]	sysclk	sysclk	50.000 ns	47.744 ns	25.049 ns
10	22.696 ns	18.31 MHz (period = 54.608 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[19][10]	sysclk	sysclk	50.000 ns	47.744 ns	25.048 ns
11	22.704 ns	18.32 MHz (period = 54.592 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[4][10]	sysclk	sysclk	50.000 ns	47.723 ns	25.019 ns
12	22.706 ns	18.32 MHz (period = 54.588 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[6][10]	sysclk	sysclk	50.000 ns	47.737 ns	25.031 ns
13	22.707 ns	18.32 MHz (period = 54.586 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[22][10]	sysclk	sysclk	50.000 ns	47.737 ns	25.030 ns
14	22.708 ns	18.32 MHz (period = 54.584 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[8][10]	sysclk	sysclk	50.000 ns	47.723 ns	25.015 ns
15	22.720 ns	18.33 MHz (period = 54.560 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[31][10]	sysclk	sysclk	50.000 ns	47.738 ns	25.018 ns
16	22.721 ns	18.33 MHz (period = 54.558 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[29][10]	sysclk	sysclk	50.000 ns	47.738 ns	25.017 ns
17	22.821 ns	18.40 MHz (period = 54.358 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[10][14]	sysclk	sysclk	50.000 ns	47.746 ns	24.925 ns
18	22.821 ns	18.40 MHz (period = 54.358 ns)	Control:Control1|ALUSrc1	RegFile:RegFile1|RF_DATA[18][14]	sysclk	sysclk	50.000 ns	47.746 ns	24.925 ns

Clock Hold: 'sysclk'

	Minimum Slack	From	To	From Clock	To Clock	Required Hold Relationship	Required Shortest P2P Time	Actual Shortest P2P Time
1	-8.058 ns	Control:Control1|ALUFun[3]	ALU:ALU1|CMP:cmp|S[0]	sysclk	sysclk	0.000 ns	9.344 ns	1.286 ns
2	-7.890 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Shift:shift|S[0]	sysclk	sysclk	0.000 ns	9.763 ns	1.873 ns
3	-7.807 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Shift:shift|S[15]	sysclk	sysclk	0.000 ns	9.620 ns	1.813 ns
4	-7.790 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Shift:shift|S[24]	sysclk	sysclk	0.000 ns	9.592 ns	1.802 ns
5	-7.775 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Shift:shift|S[12]	sysclk	sysclk	0.000 ns	9.755 ns	1.980 ns
6	-7.760 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Logic:logic|S[30]	sysclk	sysclk	0.000 ns	9.582 ns	1.822 ns
7	-7.747 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Logic:logic|S[5]	sysclk	sysclk	0.000 ns	9.602 ns	1.855 ns
8	-7.719 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Logic:logic|S[0]	sysclk	sysclk	0.000 ns	9.586 ns	1.867 ns
9	-7.697 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Logic:logic|S[25]	sysclk	sysclk	0.000 ns	9.586 ns	1.889 ns
10	-7.695 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Logic:logic|S[31]	sysclk	sysclk	0.000 ns	9.583 ns	1.888 ns
11	-7.578 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Shift:shift|S[20]	sysclk	sysclk	0.000 ns	9.593 ns	2.015 ns
12	-7.538 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Logic:logic|S[27]	sysclk	sysclk	0.000 ns	9.466 ns	1.928 ns
13	-7.514 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Logic:logic|S[18]	sysclk	sysclk	0.000 ns	9.467 ns	1.953 ns
14	-7.482 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Shift:shift|S[30]	sysclk	sysclk	0.000 ns	9.596 ns	2.114 ns
15	-7.464 ns	Control:Control1|ALUFun[1]	ALU:ALU1|CMP:cmp|S[0]	sysclk	sysclk	0.000 ns	9.342 ns	1.878 ns
16	-7.441 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Logic:logic|S[17]	sysclk	sysclk	0.000 ns	9.585 ns	2.144 ns
17	-7.398 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Shift:shift|S[19]	sysclk	sysclk	0.000 ns	9.593 ns	2.195 ns
18	-7.391 ns	Control:Control1|ALUFun[0]	ALU:ALU1|Logic:logic|S[1]	sysclk	sysclk	0.000 ns	9.583 ns	2.192 ns

tco

	Slack	Required tco	Actual tco	From	To	From Clock
1	N/A	None	22.455 ns	Peripheral:Ph1|digi[10]	digi_out2[2]	sysclk
2	N/A	None	22.314 ns	Peripheral:Ph1|digi[11]	digi_out2[2]	sysclk
3	N/A	None	22.147 ns	Peripheral:Ph1|digi[10]	digi_out2[6]	sysclk
4	N/A	None	22.006 ns	Peripheral:Ph1|digi[11]	digi_out2[6]	sysclk
5	N/A	None	21.735 ns	Peripheral:Ph1|digi[9]	digi_out3[3]	sysclk
6	N/A	None	21.700 ns	Peripheral:Ph1|digi[9]	digi_out3[2]	sysclk
7	N/A	None	21.689 ns	Peripheral:Ph1|digi[9]	digi_out3[4]	sysclk
8	N/A	None	21.680 ns	Peripheral:Ph1|digi[9]	digi_out3[6]	sysclk
9	N/A	None	21.677 ns	Peripheral:Ph1|digi[9]	digi_out3[0]	sysclk
10	N/A	None	21.662 ns	Peripheral:Ph1|digi[9]	digi_out3[1]	sysclk
11	N/A	None	21.655 ns	Peripheral:Ph1|digi[9]	digi_out4[5]	sysclk
12	N/A	None	21.644 ns	Peripheral:Ph1|digi[9]	digi_out4[0]	sysclk
13	N/A	None	21.635 ns	Peripheral:Ph1|digi[10]	digi_out2[5]	sysclk
14	N/A	None	21.601 ns	Peripheral:Ph1|digi[10]	digi_out2[3]	sysclk
15	N/A	None	21.502 ns	Peripheral:Ph1|digi[10]	digi_out2[4]	sysclk
16	N/A	None	21.494 ns	Peripheral:Ph1|digi[11]	digi_out2[5]	sysclk
17	N/A	None	21.460 ns	Peripheral:Ph1|digi[11]	digi_out2[3]	sysclk
18	N/A	None	21.434 ns	Peripheral:Ph1|digi[9]	digi_out4[2]	sysclk

tsu

	Slack	Required tsu	Actual tsu	From	To	To Clock
1	N/A	None	1.154 ns	switch[1]	RegFile:RegFile1|RF_DATA[1][1]	sysclk
2	N/A	None	1.150 ns	switch[1]	RegFile:RegFile1|RF_DATA[5][1]	sysclk
3	N/A	None	1.131 ns	switch[1]	RegFile:RegFile1|RF_DATA[15][1]	sysclk
4	N/A	None	1.126 ns	switch[1]	RegFile:RegFile1|RF_DATA[11][1]	sysclk
5	N/A	None	1.122 ns	switch[1]	RegFile:RegFile1|RF_DATA[9][1]	sysclk
6	N/A	None	0.884 ns	switch[1]	RegFile:RegFile1|RF_DATA[30][1]	sysclk
7	N/A	None	0.883 ns	switch[1]	RegFile:RegFile1|RF_DATA[24][1]	sysclk
8	N/A	None	0.883 ns	switch[1]	RegFile:RegFile1|RF_DATA[3][1]	sysclk
9	N/A	None	0.881 ns	switch[1]	RegFile:RegFile1|RF_DATA[7][1]	sysclk
10	N/A	None	0.697 ns	switch[1]	RegFile:RegFile1|RF_DATA[2][1]	sysclk
11	N/A	None	0.694 ns	switch[1]	RegFile:RegFile1|RF_DATA[31][1]	sysclk
12	N/A	None	0.675 ns	switch[1]	RegFile:RegFile1|RF_DATA[25][1]	sysclk
13	N/A	None	0.673 ns	switch[1]	RegFile:RegFile1|RF_DATA[27][1]	sysclk
14	N/A	None	0.609 ns	switch[1]	RegFile:RegFile1|RF_DATA[26][1]	sysclk
15	N/A	None	0.569 ns	switch[1]	RegFile:RegFile1|RF_DATA[29][1]	sysclk
16	N/A	None	0.566 ns	switch[1]	RegFile:RegFile1|RF_DATA[16][1]	sysclk
17	N/A	None	0.556 ns	switch[1]	RegFile:RegFile1|RF_DATA[13][1]	sysclk
18	N/A	None	0.543 ns	switch[1]	RegFile:RegFile1|RF_DATA[10][1]	sysclk

(2) 流水线处理器

Flow Summary

Flow Status	Successful - Thu Jul 14 10:58:53 2016
Quartus II Version	8.1 Build 163 10/28/2008 SJ Full Version
Revision Name	CPU_pipeline
Top-level Entity Name	CPU_pipeline
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Met timing requirements	Yes
Total logic elements	3,539 / 33,216 (11 %)
Total combinational functions	3,473 / 33,216 (10 %)
Dedicated logic registers	1,675 / 33,216 (5 %)
Total registers	1675
Total pins	48 / 475 (10 %)
Total virtual pins	0
Total memory bits	16,128 / 483,840 (3 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To	From Clock	To Clock	Failed Paths
1	Worst-case tsu	N/A	None	6.382 ns	reset	Peripheral:Ph1|digi[9]	--	sysclk	0
2	Worst-case tco	N/A	None	10.354 ns	Peripheral:Ph1|digi[10]	digi_out2[1]	sysclk	--	0
3	Worst-case th	N/A	None	-1.440 ns	switch[6]	MEM_WB:MEM_WB1|ReadData_out[6]	--	sysclk	0
4	Clock Setup: 'sysclk'	7.299 ns	20.00 MHz (period = 50.000 ns)	28.25 MHz (period = 35.402 ns)	Control:Control1|PCSrc[0]	PC:PC1|pc[29]	sysclk	sysclk	0
5	Clock Hold: 'sysclk'	3.293 ns	20.00 MHz (period = 50.000 ns)	N/A	Control:Control1|EXTOp	ID_EX:ID_EX1|LU32_out[31]	sysclk	sysclk	0
6	Total number of failed paths								0

Clock Setup: 'sysclk'

	Slack	Actual fmax (period)	From	To	From Clock	To Clock	Required Setup Relationship	Required Longest P2P Time	Actual Longest P2P Time
1	7.299 ns	28.25 MHz (period = 35.402 ns)	Control:Control1|PCSrc[0]	PC:PC1|pc[29]	sysclk	sysclk	25.000 ns	13.716 ns	6.417 ns
2	7.370 ns	28.36 MHz (period = 35.260 ns)	Control:Control1|PCSrc[0]	PC:PC1|pc[27]	sysclk	sysclk	25.000 ns	13.716 ns	6.346 ns
3	7.395 ns	28.40 MHz (period = 35.210 ns)	Control:Control1|PCSrc[0]	PC:PC1|pc[16]	sysclk	sysclk	25.000 ns	13.717 ns	6.322 ns
4	7.472 ns	28.53 MHz (period = 35.056 ns)	Control:Control1|PCSrc[0]	PC:PC1|pc[2]	sysclk	sysclk	25.000 ns	13.713 ns	6.241 ns
5	7.489 ns	28.55 MHz (period = 35.022 ns)	Control:Control1|PCSrc[1]	PC:PC1|pc[29]	sysclk	sysclk	25.000 ns	13.714 ns	6.225 ns
6	7.560 ns	28.67 MHz (period = 34.880 ns)	Control:Control1|PCSrc[1]	PC:PC1|pc[27]	sysclk	sysclk	25.000 ns	13.714 ns	6.154 ns
7	7.585 ns	28.71 MHz (period = 34.830 ns)	Control:Control1|PCSrc[1]	PC:PC1|pc[16]	sysclk	sysclk	25.000 ns	13.715 ns	6.130 ns
8	7.604 ns	28.74 MHz (period = 34.792 ns)	Control:Control1|PCSrc[0]	PC:PC1|pc[18]	sysclk	sysclk	25.000 ns	13.725 ns	6.121 ns
9	7.684 ns	28.88 MHz (period = 34.632 ns)	Control:Control1|PCSrc[1]	PC:PC1|pc[2]	sysclk	sysclk	25.000 ns	13.711 ns	6.027 ns
10	7.777 ns	29.03 MHz (period = 34.446 ns)	Control:Control1|PCSrc[0]	PC:PC1|pc[5]	sysclk	sysclk	25.000 ns	13.717 ns	5.940 ns
11	7.794 ns	29.06 MHz (period = 34.412 ns)	Control:Control1|PCSrc[1]	PC:PC1|pc[18]	sysclk	sysclk	25.000 ns	13.723 ns	5.929 ns
12	7.814 ns	29.09 MHz (period = 34.372 ns)	Control:Control1|PCSrc[0]	PC:PC1|pc[12]	sysclk	sysclk	25.000 ns	13.713 ns	5.899 ns
13	7.880 ns	29.21 MHz (period = 34.240 ns)	Control:Control1|PCSrc[0]	PC:PC1|pc[14]	sysclk	sysclk	25.000 ns	13.720 ns	5.840 ns
14	7.967 ns	29.35 MHz (period = 34.066 ns)	Control:Control1|PCSrc[1]	PC:PC1|pc[5]	sysclk	sysclk	25.000 ns	13.715 ns	5.748 ns
15	7.974 ns	29.37 MHz (period = 34.052 ns)	Control:Control1|PCSrc[0]	PC:PC1|pc[22]	sysclk	sysclk	25.000 ns	13.716 ns	5.742 ns
16	7.976 ns	29.37 MHz (period = 34.048 ns)	Control:Control1|PCSrc[0]	PC:PC1|pc[20]	sysclk	sysclk	25.000 ns	13.724 ns	5.748 ns
17	8.026 ns	29.46 MHz (period = 33.948 ns)	Control:Control1|PCSrc[1]	PC:PC1|pc[12]	sysclk	sysclk	25.000 ns	13.711 ns	5.685 ns
18	8.028 ns	29.46 MHz (period = 33.944 ns)	Control:Control1|PCSrc[0]	PC:PC1|pc[21]	sysclk	sysclk	25.000 ns	13.719 ns	5.691 ns

Clock Hold: 'sysclk'

	Minimum Slack	From	To	From Clock	To Clock	Required Hold Relationship	Required Shortest P2P Time	Actual Shortest P2P Time
1	3.293 ns	Control:Control1|EXTOp	ID_EX:ID_EX1|LU32_out[31]	sysclk	sysclk	0.000 ns	-1.979 ns	1.314 ns
2	3.345 ns	Control:Control1|EXTOp	ID_EX:ID_EX1|EXT32_out[29]	sysclk	sysclk	0.000 ns	-1.979 ns	1.366 ns
3	3.550 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[9]	sysclk	sysclk	0.000 ns	-2.125 ns	1.425 ns
4	3.551 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[31]	sysclk	sysclk	0.000 ns	-2.125 ns	1.426 ns
5	3.682 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[0]	sysclk	sysclk	0.000 ns	-2.125 ns	1.557 ns
6	3.803 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[7]	sysclk	sysclk	0.000 ns	-2.111 ns	1.692 ns
7	3.813 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[29]	sysclk	sysclk	0.000 ns	-2.112 ns	1.701 ns
8	3.816 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[30]	sysclk	sysclk	0.000 ns	-2.112 ns	1.704 ns
9	3.817 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[27]	sysclk	sysclk	0.000 ns	-2.112 ns	1.705 ns
10	3.818 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[3]	sysclk	sysclk	0.000 ns	-2.109 ns	1.709 ns
11	3.821 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[5]	sysclk	sysclk	0.000 ns	-2.109 ns	1.712 ns
12	3.914 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[4]	sysclk	sysclk	0.000 ns	-2.112 ns	1.802 ns
13	3.927 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[24]	sysclk	sysclk	0.000 ns	-2.112 ns	1.815 ns
14	3.928 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[26]	sysclk	sysclk	0.000 ns	-2.112 ns	1.816 ns
15	3.930 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[22]	sysclk	sysclk	0.000 ns	-2.112 ns	1.818 ns
16	3.930 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[21]	sysclk	sysclk	0.000 ns	-2.112 ns	1.818 ns
17	3.930 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[16]	sysclk	sysclk	0.000 ns	-2.112 ns	1.818 ns
18	3.933 ns	Control:Control1|LUOp	ID_EX:ID_EX1|LU32_out[18]	sysclk	sysclk	0.000 ns	-2.112 ns	1.821 ns

tco

	Slack	Required tco	Actual tco	From	To	From Clock
1	N/A	None	10.354 ns	Peripheral:Ph1|digi[10]	digi_out2[1]	sysclk
2	N/A	None	10.079 ns	Peripheral:Ph1|digi[11]	digi_out2[1]	sysclk
3	N/A	None	9.882 ns	Peripheral:Ph1|digi[10]	digi_out2[2]	sysclk
4	N/A	None	9.843 ns	Peripheral:Ph1|digi[10]	digi_out2[3]	sysclk
5	N/A	None	9.841 ns	Peripheral:Ph1|digi[10]	digi_out2[5]	sysclk
6	N/A	None	9.832 ns	Peripheral:Ph1|digi[10]	digi_out2[0]	sysclk
7	N/A	None	9.820 ns	Peripheral:Ph1|digi[8]	digi_out3[0]	sysclk
8	N/A	None	9.797 ns	Peripheral:Ph1|digi[8]	digi_out3[5]	sysclk
9	N/A	None	9.788 ns	Peripheral:Ph1|digi[8]	digi_out3[1]	sysclk
10	N/A	None	9.784 ns	Peripheral:Ph1|digi[8]	digi_out3[2]	sysclk
11	N/A	None	9.759 ns	Peripheral:Ph1|digi[8]	digi_out4[0]	sysclk
12	N/A	None	9.699 ns	Peripheral:Ph1|digi[0]	digi_out2[0]	sysclk
13	N/A	None	9.692 ns	Peripheral:Ph1|digi[9]	digi_out3[0]	sysclk
14	N/A	None	9.684 ns	Peripheral:Ph1|digi[5]	digi_out2[5]	sysclk
15	N/A	None	9.669 ns	Peripheral:Ph1|digi[9]	digi_out3[5]	sysclk
16	N/A	None	9.660 ns	Peripheral:Ph1|digi[9]	digi_out3[1]	sysclk
17	N/A	None	9.656 ns	Peripheral:Ph1|digi[9]	digi_out3[2]	sysclk
18	N/A	None	9.631 ns	Peripheral:Ph1|digi[9]	digi_out4[0]	sysclk

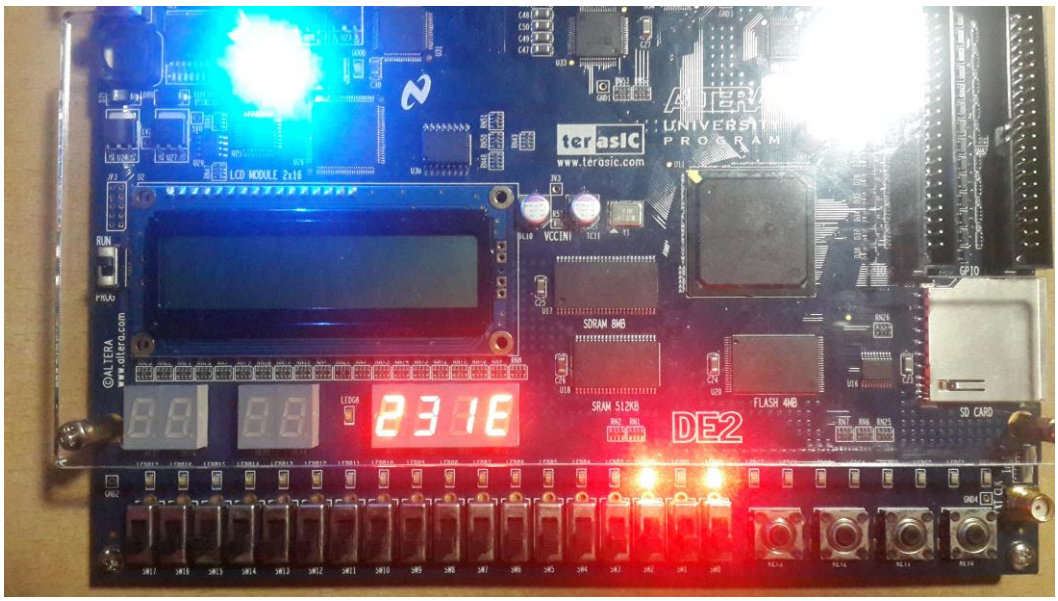
tsu

	Slack	Required tsu	Actual tsu	From	To	To Clock
1	N/A	None	6.382 ns	reset	Peripheral:Ph1|digi[8]	sysclk
2	N/A	None	6.382 ns	reset	Peripheral:Ph1|digi[9]	sysclk
3	N/A	None	6.232 ns	reset	Peripheral:Ph1|digi[11]	sysclk
4	N/A	None	6.232 ns	reset	Peripheral:Ph1|digi[10]	sysclk
5	N/A	None	5.788 ns	reset	Peripheral:Ph1|led[0]	sysclk
6	N/A	None	5.788 ns	reset	Peripheral:Ph1|led[6]	sysclk
7	N/A	None	5.788 ns	reset	Peripheral:Ph1|led[5]	sysclk
8	N/A	None	5.788 ns	reset	Peripheral:Ph1|led[7]	sysclk
9	N/A	None	5.788 ns	reset	Peripheral:Ph1|led[2]	sysclk
10	N/A	None	5.788 ns	reset	Peripheral:Ph1|led[4]	sysclk
11	N/A	None	5.788 ns	reset	Peripheral:Ph1|led[1]	sysclk
12	N/A	None	5.788 ns	reset	Peripheral:Ph1|led[3]	sysclk
13	N/A	None	5.720 ns	reset	Peripheral:Ph1|digi[0]	sysclk
14	N/A	None	5.720 ns	reset	Peripheral:Ph1|digi[6]	sysclk
15	N/A	None	5.720 ns	reset	Peripheral:Ph1|digi[5]	sysclk
16	N/A	None	5.720 ns	reset	Peripheral:Ph1|digi[7]	sysclk
17	N/A	None	5.720 ns	reset	Peripheral:Ph1|digi[2]	sysclk
18	N/A	None	5.720 ns	reset	Peripheral:Ph1|digi[4]	sysclk

I、硬件调试情况（慕思成、陈佳榕）

将模块烧至开发板上，利用串口调试助手进行调试，16 进制发送、显示。

如下图：



任意输入两组数据并手动发送，如 23 1E（也即十进制的 35 30），数码管可以显示操作数（23 1E），LED 点亮显示结果（00000101，即 5），串口可接受到数据（05）；更换数据后仍能正确显示结果，说明设计正确。

自动发送两组数据，数码管显示，LED 显示及串口接收结果均可以一直正确，大约 1000 组数据可能随机出现 1 组误码（大部分情况下不出现），说明设计符合要求。

J、思想体会（慕思成、陈佳榕）

慕思成：

本次实验是我第一次使用 verilog 语言来编写这么大的一个程序。我们组只有 2 个人，我在组内的分工主要为单周期、流水线中数据通路的搭建（控制模块、流水线间寄存器、连线等）和汇编代码。这使得我对于处理器的数据通路及各个部分的意义有了更深一步的认识，并进一步熟悉和巩固了理论课所学的知识。

本次实验的调试中我主要着眼 pc 的变化，各种奇奇怪怪的错误中首先会反应到 pc 的值出现反常的变化。尤其是在流水线中，由于冒险和转发的存在以及各个阶段的分开，使得我在考虑 pc 和指令时必须要清晰的了解哪个阶段需要用到哪个阶段的信号，而在这次分支/跳转/中断后 pc 会调到哪里？进而我就会发现我在某个模块中又犯了低级错误、或是没有理清之一模块地思路。这一点在程序计数器的多路选择器上尤为明显，它的输入在各个阶段的 PCSrc 中进行选择，一旦稍微没想清楚，就会导致整个程序出现很大的错误。

由于本次实验对于我们 Verilog 初学者十一次很大的工程，所以他对我们编写、调试代码能力也是一个很大的考验，bug 基本是肯定会出现的，有时候

思路觉得对了却仍然出错，而且经常是想了很久都找不出来原因，令人抓狂，这就要我要有足够的耐心，静下心来去考虑代码中的细节，从一个小错开始一步步往前推，寻找答案。还有时候，自己怎么也想不出来的 bug，问一问同组的队友，与不同的思路交流一下就能解决问题。

最后感谢队友的支持与帮助使我能顺利完成本次实验。

陈佳榕：

因为本次实验我们队只有两个人，所以平均下来个人的工作量会比其他那些三个人一组的队伍的个人工作量更多，能够按时完成也蛮不容易的，不过这也让我们收获更多，因为负责的部分多，对 CPU 的理解就更加全面深刻。

我负责的主要是 ALU 部分，编译器部分及流水线的冒险检测及转发单元。

ALU 部分相对来说比较简单，但是却是对理论课上所学内容极好的复习与巩固，认认真真地写完 ALU，仿真然后 debug，不仅加深了对可是所学东西比如补码、有符号运算与无符号运算的区别等的认识理解，更深刻体会到模块化思想的好处，把一个复杂的 ALU 模块再分为若干小模块，不仅方便理解实现，实际上更加速了工作的进度。

写编译器的时候可以说是将课上学到的知识运用于实际了，而且还有更深入的认识理解，对于指令格式与机器码的认识更进一层。最后我还将整个 Java 程序打成包，可以在很多电脑上运行，这种成就感也是无与伦比的。

写冒险检测与转发的时候基本上按照理论课上所讲的加上自己的一点理解来写的，将学到的东西亲自动手实现出来也是记住这些知识最好的方法，也对流水线的工作方式有更深刻的理解。

总而言之，这次实验给了我们实践理论课所学知识的一个很好的机会，在写代码、

仿真、debug 并与小伙伴沟通交流的过程中不断加深对 CPU 的理解，经过一个个废寝忘食的日夜，终于调试完在板子上跑了起来，看着结果出来的那一瞬间真是无比的兴奋与喜悦，成就感顿然于心中。