**数字逻辑与处理器基础实验报告**

**——32位MIPS流水线设计**

吴昆 无58 2015010625

黄鹏志\*

李登昊\*

2017年7月

\*The information of other two authors are removed for the github copy.

I'm trying to free your mind, Neo. But I can only show you the door. You're the one that has to walk through it.

— “Morpheus” in The Matrix

目 录

[一、32位ALU设计（吴昆） 4](#_Toc488435026)

[1.1 设计方案 4](#_Toc488435027)

[1.2 加减运算器 4](#_Toc488435028)

[1.3 比较运算器 5](#_Toc488435029)

[1.4 逻辑运算器 6](#_Toc488435030)

[1.5 移位运算器 6](#_Toc488435031)

[二、32位单周期MIPS设计（吴昆，黄鹏志） 7](#_Toc488435032)

[2.1 设计目标 7](#_Toc488435033)

[2.1 设计方案 8](#_Toc488435034)

[2.2 PC 8](#_Toc488435035)

[2.3 主通路关键代码 9](#_Toc488435036)

[2.3.1 IF阶段 9](#_Toc488435037)

[2.3.2 ID阶段 9](#_Toc488435038)

[2.3.3 EX阶段 10](#_Toc488435039)

[2.3.4 MEM阶段 10](#_Toc488435040)

[2.4 控制单元（黄鹏志） 11](#_Toc488435041)

[2.5 仿真测试 12](#_Toc488435042)

[2.6 硬件调试（李登昊） 13](#_Toc488435043)

[2.7 综合情况时序分析 13](#_Toc488435044)

[2.8 综合情况面积分析 14](#_Toc488435045)

[2.7 体会（吴昆） 16](#_Toc488435046)

[三、32位流水线MIPS处理器设计（黄鹏志，吴昆） 16](#_Toc488435047)

[3.1 设计目标 16](#_Toc488435048)

[3.2 流水线的基本设计（黄鹏志） 16](#_Toc488435049)

[3.2.1 阶段寄存器 16](#_Toc488435050)

[3.2.2 其它主要元件及其逻辑 19](#_Toc488435051)

[3.3 转发、停和取消指令（吴昆） 21](#_Toc488435052)

[3.3.1 冒险和异常中断 21](#_Toc488435053)

[3.3.2 转发 22](#_Toc488435054)

[3.3.3 阶段寄存器清空 23](#_Toc488435055)

[3.3.4 冒险检测单元 24](#_Toc488435056)

[3.4 编码中遇到的bug 24](#_Toc488435057)

[3.5 流水线的中断和异常处理（黄鹏志） 24](#_Toc488435058)

[3.5.1 原则 25](#_Toc488435059)

[3.5.2 中断处理触发的讨论 25](#_Toc488435060)

[3.5.3 精确异常 25](#_Toc488435061)

[3.6 针对频率的优化 26](#_Toc488435062)

[3.6.1 ALU组件去除级联依赖（吴昆） 26](#_Toc488435063)

[3.6.2 B指令独立比较器（吴昆） 26](#_Toc488435064)

[3.6.3 控制信号扇出的改进（黄鹏志） 26](#_Toc488435065)

[3.7 仿真与硬件调试 27](#_Toc488435066)

[3.8 综合情况 27](#_Toc488435067)

[3.9 体会（吴昆，黄鹏志） 28](#_Toc488435068)

[四、机器码转换程序（吴昆） 29](#_Toc488435069)

[4.1 设计目标 29](#_Toc488435070)

[4.2 设计方案 29](#_Toc488435071)

[4.3 主流程 29](#_Toc488435072)

[4.4 体会 32](#_Toc488435073)

[五、汇编程序（李登昊） 32](#_Toc488435074)

[5.1 设计目标 32](#_Toc488435075)

[5.2 设计方案 32](#_Toc488435076)

[5.2.1 更相减损术 32](#_Toc488435077)

[5.2.2 开始运行 33](#_Toc488435078)

[5.2.3 中断处理 33](#_Toc488435079)

[5.2.4 异常处理 35](#_Toc488435080)

[六、外设（李登昊） 35](#_Toc488435081)

[6.1 设计目标 35](#_Toc488435082)

[6.2 设计方案 36](#_Toc488435083)

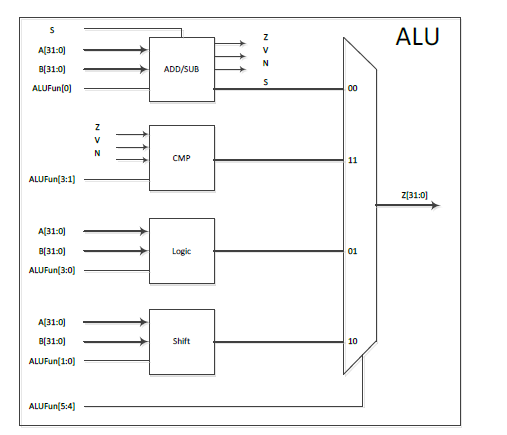
[6.2.1 定时器及开发板上外设资源 36](#_Toc488435084)

[6.2.2 UART 36](#_Toc488435085)

[七、参考文献 37](#_Toc488435086)

# 一、32位ALU设计（吴昆）

## 1.1 设计方案



设计按照实验指导书给的系统框图，分成四个模块，每个模块内部使用组合逻辑写成。根据ALUFun高2位决定最终输出是哪个模块的输出，再根据低4位对具体的模块计算进行选择。

其中，移位器按照指导书，根据指令种类分成（逻辑）左移移位器，逻辑右移移位器和算术右移移位器，分别由16、8、4、2、1位同种移位器级联形成。

其中比较器的相等判断一开始使用了加法器的Z，但后来由于优化频率采用单独电路实现，详见1.2。

我们在单周期仿真中进行了测试，覆盖了ALU的大部分计算，由单周期运行的结果正确得知ALU的实现是正确的。

下面我们分节阐述四个部件的设计；ALU源代码为ALU.v。

## 1.2 加减运算器

分别计算加法结果(\_add后缀)和减法结果(\_sub)后缀，根据ALUFun[0]来进行选择。其中减法采用将减数b取反加一得到相反数转化为加法。

区分有符号和无符号运算，当Sign为高电平，此时N,V标志位有意义；Sign为低电平N,V恒为0.

负数的判断为有符号计算（因而结果有符号）且符号位为1.

溢出的判断为有符号计算，且两个加数的符号位相同，但与结果不同，写成异或和异或取反的与。

1. module addsub(S,Z,V,N,in1,in2,ALUFun,Sign);
2. output [31:0] S;
3. output Z,V,N;
4. input [31:0] in1, in2;
5. input ALUFun,Sign;//ALUFun[0]
6. wire [31:0]s\_add,s\_sub;
7. wire z\_add,v\_add,n\_add, z\_sub,v\_sub,n\_sub;
8. wire [31:0] inv\_in2;
9. assign inv\_in2 = ~in2+1;
10. assign s\_add = in1+in2;
11. assign z\_add = (s\_add==0);
12. assign n\_add = Sign&s\_add[31];
13. assign v\_add = Sign&(s\_add[31]^in1[31])&~(in1[31]^in2[31]);
14. assign s\_sub = in1+inv\_in2;
15. assign z\_sub = (s\_sub==0);
16. assign n\_sub = Sign&s\_sub[31];
17. assign v\_sub = Sign&(s\_sub[31]^in1[31])&~(in1[31]^inv\_in2[31]);
18. assign S=ALUFun?s\_sub:s\_add;//1?sub
19. assign Z=ALUFun?z\_sub:z\_add;
20. assign N=ALUFun?n\_sub:n\_add;
21. assign V=ALUFun?v\_sub:v\_add;
22. endmodule

## 1.3 比较运算器

比较运算器使用同样的操作，先求得各个结果，再根据ALUFun相关位选择其中的结果。

加法器的Z就是相等判断，一开始我们就直接用了，也符合实验指导书框图的设计。但是在优化时，我们发现，关键路径包括了ALU输入->减法->比较这一段，也就是说从加减器输出到比较器的级联增加了关键路径的长度。为优化速度，我们为比较器单独设计了相等运算（也就是给予单独的电路），降低了处理器的worst negative slack.我们在单独的优化报告中将详细阐述。

小于判断时，ALUFun[0]==1，加法器会做减法运算，如果结果小于0（N==1），那么就是小于，这是没有溢出的情况。如果溢出，那么说明a正b负或者a负b正，只有后一种a<b.判断方法为a[31]。

与0的各种比较，同样需要注意符号位的问题，其余的和jiajia呢运算器差不多。

1. module cmp(S,V,N,ALUFun,in1,in2,Sign);
2. input V,N,Sign;
3. input [2:0] ALUFun;
4. input [31:0] in1,in2;
5. output [31:0] S;
6. //EQ001,NEQ000,LT010,LEZ110,LTZ101,GTZ111
7. wire EQ,NEQ,LT,LEZ,LTZ,GTZ;
8. assign EQ=in1==in2;
9. assign NEQ=~(in1==in2);
10. assign LT=(~V&N)|(V&in1[31]);
11. assign EQZ=(in1==0);
12. assign LEZ=Sign?(in1[31]||EQZ):EQZ;
13. assign LTZ=Sign?(in1[31]):0;
14. assign GTZ=Sign?(~in1[31]):~EQZ;
15. assign S=(ALUFun[2]==0)?
16. (
17. (ALUFun[1]==0)?
18. ((ALUFun[0]==0)?NEQ:EQ)//000,001
19. :
20. ((ALUFun[0]==0)?LT:0)//010,011
21. )
22. :
23. (
24. (ALUFun[1]==0)?
25. ((ALUFun[0]==0)?0:LTZ)//100,101
26. :
27. ((ALUFun[0]==0)?LEZ:GTZ)//110,111
28. );
29. endmodule1.3 逻辑运算器

## 1.4 逻辑运算器

逻辑运算器涉及了常见的位逻辑操作，都是verilog内置的，直接使用就好。

1. module logic(out,in1,in2,ALUFun,Z);
2. output [31:0] out;
3. input [31:0] in1,in2;
4. input [3:0] ALUFun;
5. input Z;
6. wire [31:0] \_and,\_or,\_xor,\_nor;
7. assign \_and=in1&in2;
8. assign \_or=in1|in2;
9. assign \_xor=in1^in2;
10. assign \_nor=~(in1|in2);
11. assign out=(ALUFun[3]==1)?(
12. (ALUFun[2]==1)?\_or:(
13. (ALUFun[1]==1)?in1:\_and))
14. :
15. ((ALUFun[2]==1)?\_xor:\_nor);
16. endmodule

## 1.5 移位运算器

移位运算器采用了实验指导书中的操作，将8、4、2、1移位器级联得到，下面是算数右移运算器的代码。

1. module arshift(out,in,shamt);
2. output [31:0] out;
3. input [31:0] in;
4. input [4:0] shamt;
5. wire [31:0] \_out1,\_out2,\_out4,\_out8,\_out16;
6. wire [31:0] \_in1,\_in2,\_in4,\_in8,\_in16;
7. \_arshift16 ar16(\_out16,\_in16);
8. \_arshift8 ar8(\_out8,\_in8);
9. \_arshift4 ar4(\_out4,\_in4);
10. \_arshift2 ar2(\_out2,\_in2);
11. \_arshift1 ar1(\_out1,\_in1);
12. assign \_in16=in;
13. assign \_in8=(shamt[4]==1)?\_out16:in;
14. assign \_in4=(shamt[3]==1)?\_out8:(
15. (shamt[4]==1)?\_out16:in);
16. assign \_in2=(shamt[2]==1)?\_out4:(
17. (shamt[3]==1)?\_out8:(
18. shamt[4]==1)?\_out16:in));
19. assign \_in1=(shamt[1]==1)?\_out2:(
20. (shamt[2]==1)?\_out4:(
21. (shamt[3]==1)?\_out8:(
22. (shamt[4]==1)?\_out16:in)));
23. assign out=(shamt[0]==1)?\_out1:(
24. (shamt[1]==1)?\_out2:(
25. (shamt[2]==1)?\_out4:(
26. (shamt[3]==1)?\_out8:(
27. (shamt[4]==1)?\_out16:in))));
28. endmodule

将三种类型的子移位器结合起来，就得到了移位运算器。

1. module shift(out,in1,in2,ALUFun);
2. output [31:0] out;
3. input [31:0] in1,in2;
4. input [1:0] ALUFun;
5. wire [31:0] \_arout,\_rout,\_lout;
6. arshift ar(.out(\_arout),.in(in2),.shamt(in1[4:0]));
7. rshift r(.out(\_rout),.in(in2),.shamt(in1[4:0]));
8. lshift l(.out(\_lout),.in(in2),.shamt(in1[4:0]));
9. assign out=(ALUFun[1]==1)?\_arout:(
10. (ALUFun[0]==1)?\_rout:\_lout);
11. endmodule

# 二、32位单周期MIPS设计（吴昆，黄鹏志）

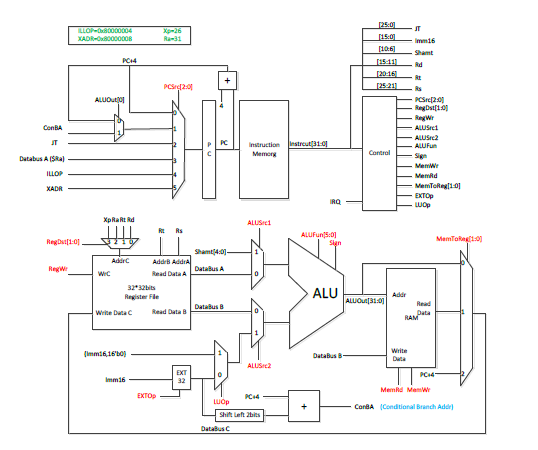
## 2.1 设计目标

1. 设计32位单周期MIPS处理器，能够执行核心的30余条指令

2. 能够正确处理中断和异常

3. 能够正确在FPGA烧入，并执行最大公约数程序

## 2.1 设计方案



单周期的实现较为熟练，实验指导书已经给出了一种可行的datapath，我们就按照这样的方式来实现。其中主要元件除了ALU和控制单元均已给出，ALU如前述。控制单元使用组合逻辑写成。途中的所有多路选择器和元件信号相接全部使用assign完成。

主要分成两个部分，PC计算以及主通路（IM->Control->RF->ALU->MEM->RF）部分。

## 2.2 PC

PC计算部分，按照图中，分别得到五个输入ConBA(B指令), JT(J指令), $Ra(jalr, jr),以及两个常量(在verilog中以parameter实现)，并街上相应的控制信号。可以参见下面代码。

对于PC监督位，只有RESET、异常、中断会使得PC[31]置为1，JR、JALR会将其置零。我们采用诸如assign PCadd4={PC[31],\_PCadd4[30:0]}; 其中assign \_PCadd4=PC+4;确保监督位在未经允许的情况下保持原样，不会被修改。

1. assign JT={PC[31:28],Instruct[25:0],2'b0};
2. assign Imm16=Instruct[15:0];
3. assign Shamt=Instruct[10:6];
4. assign Rd=Instruct[15:11];
5. assign Rt=Instruct[20:16];
6. assign Rs=Instruct[25:21];
7. assign \_PCadd4=PC+4;
8. assign PCadd4={PC[31],\_PCadd4[30:0]};
9. assign PC1=ALUOut[0]?ConBA:PCadd4;
10. assign PCIn=(PCSrc==3'b000)?PCadd4:
11. (PCSrc==3'b001)?PC1:
12. (PCSrc==3'b010)?JT:
13. (PCSrc==3'b011)?DataBusA:
14. (PCSrc==3'b100)?ILLOP:XADR;

在时钟上升沿，更新PC。

1. always @(negedge reset or posedge clk)
2. begin
3. if(~reset)
4. PC<=32'h80000000;
5. else
6. PC <= PCIn;
7. end

## 2.3 主通路关键代码

仿照流水线的概念，分成五个阶段，以使得写代码时思路比较清晰。

### 2.3.1 IF阶段

先写一个CPU时钟发生器，不在此赘述，见cpuclk.v。

取指令时先从ROM中读指令，然后由控制单元产生相应信号。实际上，在流水线设计中，我们把Control放在了ID阶段。

Control需要接受中断信号IRQ，以在中断发生时跳转到相应地址，并存储返回地址。在Control的实现（Control.v）中，将先判断是否出现中断，如果不出现再发射新的指令的控制信号。

对于未定义指令，Control将读入32位指令，并自行判断，进行相应的控制信号输出。

1. cpuclk U\_cpuclk(clk,sysclk,reset);
2. //IF
3. ROM U\_rom(PC[30:0],Instruct);
4. Control U\_control(Instruct, IRQ, PC[31],PCSrc, RegDst, RegWr,ALUSrc1, ALUSrc2, ALUFun, Sign, MemWr, MemRd, MemToReg,EXTOp, LUOp);

取几个变量，将指令分成几段，这样等下用时比较清晰。

1. assign AddrC=(RegDst==0)?Rd:
2. (RegDst==1)?Rt:
3. (RegDst==2)?Ra:Xp;
4. assign EXTout=EXTOp?(Imm16[15]?{16'hFFFF,Imm16}:{16'h0000,Imm16}):{16'h0000,Imm16};
5. assign LUout=LUOp?{Imm16,16'b0}:EXTout;
6. assign ALUIn1=ALUSrc1?Shamt:DataBusA;
7. assign ALUIn2=ALUSrc2?LUout:DataBusB;
8. assign \_EXTout\_shift2={EXTout[29:0],2'b00};
9. assign ConBA=PCadd4+\_EXTout\_shift2;//TODO会改变监督位？

### 2.3.2 ID阶段

ID阶段是比较简单的阶段，主要是读取寄存器，以及LU,EXT,ConBA的计算。

1. //ID
2. RegFile U\_regfile(reset,clk,Rs,DataBusA,Rt,DataBusB,RegWr,AddrC,DataBusC);
3. assign AddrC=(RegDst==0)?Rd:
4. (RegDst==1)?Rt:
5. (RegDst==2)?Ra:Xp;
6. assign EXTout=EXTOp?(Imm16[15]?{16'hFFFF,Imm16}:{16'h0000,Imm16}):{16'h0000,Imm16};
7. assign LUout=LUOp?{Imm16,16'b0}:EXTout;
8. assign ALUIn1=ALUSrc1?Shamt:DataBusA;
9. assign ALUIn2=ALUSrc2?LUout:DataBusB;
10. assign \_EXTout\_shift2={EXTout[29:0],2'b00};
11. assign ConBA=PCadd4+\_EXTout\_shift2;

### 2.3.3 EX阶段

执行阶段在单周期中同样非常简单，就是ALU运算。由于涉及到转发，我们在流水线设计中将转发多路选择放在了EX阶段。

1. //EX
2. ALU U\_alu(ALUOut,ALUIn1,ALUIn2,ALUFun,Sign);

### 2.3.4 MEM阶段

MEM阶段涉及到了RAM、外设，后者是唯一会产生中断的设备，因此这一部分较为复杂。

我们将UART单独写成模块，代码详见UART.v。因此有三个会产生读数据的模块Datamem, Peripheral, Uart。我们实现时在这三个模块里对读地址进行判断，如果不是自己的地址范围，就输出0；将三个得到的数据取或，就得到读取的数据。

同理，外设Peripheral和Uart会产生三种中断，将其取或就得到了中断信号IRQ，将其输入控制单元Control，即可实现中断。

1. //MEM
2. wire [31:0] mem\_rddata,peri\_rddata,uart\_data;
3. wire uart\_tx,uart\_rx,irqout,rx\_irq,tx\_irq;
4. wire [7:0] led,switch;//led输出，switch输入
5. wire [11:0] digi;//digi输出
6. DataMem U\_mem(reset,clk,MemRd,MemWr,ALUOut,DataBusB,mem\_rddata);
7. Peripheral U\_peripheral(reset,clk,MemRd,MemWr,ALUOut,DataBusB,peri\_rddata,led,switch,digi,irqout);
8. Uart U\_uart(sysclk, clk, reset, uart\_tx, uart\_rx, MemRd, MemWr, ALUOut, DataBusB, uart\_data, rx\_irq, tx\_irq);
9. assign IRQ=tx\_irq|rx\_irq|irqout;
10. //WB
11. assign DataBusC=(MemToReg==0)?ALUOut:
12. (MemToReg==1)?(mem\_rddata|peri\_rddata|uart\_data):PCadd4;

## 2.4 控制单元（黄鹏志）

控制单元首先打表（见Control.xlsx），然后按照表来写，用assign和?:操作符组合逻辑实现。（代码见control.v）某个控制信号的无关项通常被归入数量最多的状态。

输入32位指令，输出各个控制信号，以ALUFun为例：

1. assign ALUFun =
2. ((OpCode == 6'h25)||((OpCode == 0)&&(Funct == 6'h25)))?6'b011110:
3. ((OpCode == 0)&&((Funct == 6'h22)||(Funct == 6'h23)))?6'b000001:
4. ((OpCode == 6'h0c)||((OpCode == 0)&&(Funct == 6'h24)))?6'b011000:
5. ((OpCode == 0)&&(Funct == 6'h26))?6'b010110:
6. ((OpCode == 0)&&(Funct == 6'h27))?6'b010001:
7. ((OpCode == 0)&&(Funct == 0))?6'b100000:
8. ((OpCode == 0)&&(Funct == 6'h02))?6'b100001:
9. ((OpCode == 0)&&(Funct == 6'h03))?6'b100011:
10. ((OpCode == 6'h0a)||(OpCode == 6'h0b)||((OpCode == 0)&&(Funct == 6'h2a)))?6'b110101:
11. (OpCode == 6'h04)?6'b110011:
12. (OpCode == 6'h05)?6'b110001:
13. (OpCode == 6'h06)?6'b111101:
14. (OpCode == 6'h07)?6'b111111:
15. (OpCode == 6'h01)?6'b111011:
16. 0;

未定义指令的判定，就用OpCode来判断，当OpCode为0时就用Funct判断，在规定的指令集以外的全部当作未定义指令：

1. assign ill\_operation = ~(((OpCode == 6'h23) || (OpCode == 6'h2b) || (OpCode == 6'h0f) ||
2. (OpCode == 6'h08) || (OpCode == 6'h09) || (OpCode == 6'h0c) || (OpCode == 6'h20) ||
3. (OpCode == 6'h0a) || (OpCode == 6'h0b) || (OpCode == 6'h04) || (OpCode == 6'h05) ||
4. (OpCode == 6'h06) || (OpCode == 6'h07) || (OpCode == 6'h01) || (OpCode == 6'h02) ||
5. (OpCode == 6'h03) || ((OpCode == 6'h00) && ((Funct == 6'h00) || (Funct == 6'h02) ||
6. (Funct == 6'h03) || (Funct == 6'h22) || (Funct == 6'h23) || (Funct == 6'h08) ||
7. (Funct == 6'h09) || (Funct == 6'h20) || (Funct == 6'h21) || (Funct == 6'h24) ||
8. (Funct == 6'h25) || (Funct == 6'h26) || (Funct == 6'h27) || (Funct == 6'h2a) )))||PC\_31);

当然，ill\_operation变量考虑到了PC监督位的影响，当后者为1时，禁止异常、中断。中断和异常在control中最优先进行判断，也就是说有中断/异常，先处理中断/异常，流水线中也是如此，以PCSrc为例：

1. assign PCSrc[2:0]=
2. (interuption)?5:
3. (ill\_operation)?4:
4. ((OpCode == 6'h04)||(OpCode == 6'h05)||(OpCode == 6'h06)||(OpCode == 6'h07))?1:
5. ((OpCode == 6'h02)||(OpCode == 6'h03))?2:
6. ((OpCode == 6'h00)&&((Funct==6'h08)||(Funct==6'h09)))?3:
7. 0;

## 2.5 仿真测试

我们根据所给的ROM.v，进行了适当的修改，形成如下的测试（汇编语言给出）。文件详见rom\_test.v

1. #<unknown instruction>
2. lui $17 0x8000 #correct
3. addiu $17 $17 0x00004 #correct
4. addiu $16 $0 0x000AA #correct
5. sw $17 0x0003($16) #未检查 这条指令的翻译对吗 信号是对的
6. jal 0x00000005 #测 #MemToReg有問題 去掉两条指令后变成0x0000007
7. #sll $0 $0 0x00000 #nop 后面开始指令全是32'h0的问题在于rom写错了
8. sll $4 $17 0x00004 #h00112100
9. sra $5 $17 0x00003 #h001128C3
10. or $17 $4 $17 #h02748825
11. add $17 $4 $17 #$19>$20 $17=0 h02748820
12. sw $17 0x0003($16) #hAE110003
13. slt $17 $19 $20 #h0274882A
14. beq $0 $17 0x00001 #应该跳过下一条 h10110001 确实跳过了
15. sub $17 $20 $19
16. sub $17 $19 $20
17. lw $20 0x0003($16)
18. j 0x00000001

为了测试sw和lw正常，采用先存再读的方式，如果一致说明正确。

slt和beq连用，同时测试。

对or、add，以及立即数addi这些ALU逻辑指令进行了测试，对逻辑、算数移位进行了测试，还对跳转jal测试（正确跳转和设置$ra）.

对仿真波形，我们会查看PC寄存器和下一个PC值是否正确，寄存器值和控制信号是否正确。

通过仿真测试，解决了不少bug，按照顺序如下：

1.变量名错误、大小写错误引发的未连线。

2.未初始化导致的z/x.并且拒绝使用initial begin；reset的定义是下降沿触发复位，我们一开始有些部分按照高电平来写，导致了错误的信号。

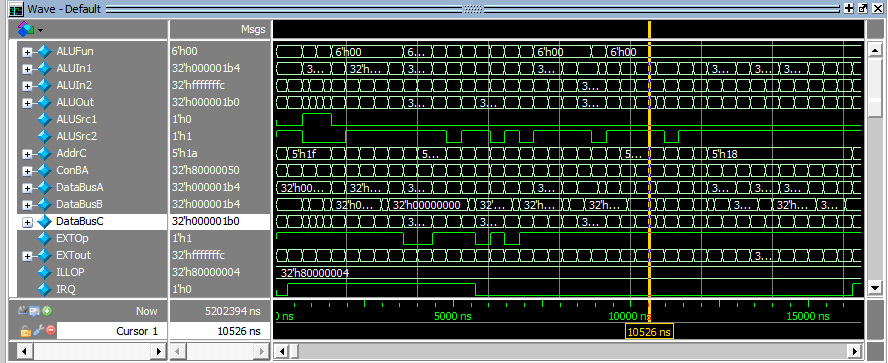
3.CPU时钟单元cpuclk初始化时忘记给s赋值，导致无正确输出。

4.当ROM\_SIZE==32时，指令执行到第八条以后恒为0，经过观察解决了ROM.v里地址计算的bug.

5.JT忘记2位的偏移。

6.计算or时发生错误，经过检查是ALU中的or中间计算结果\_or没有定义32位宽度。

最终正确的结果如图所示：

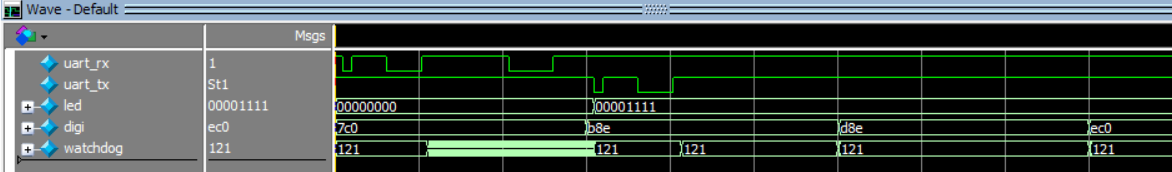


## 2.6 硬件调试（李登昊）

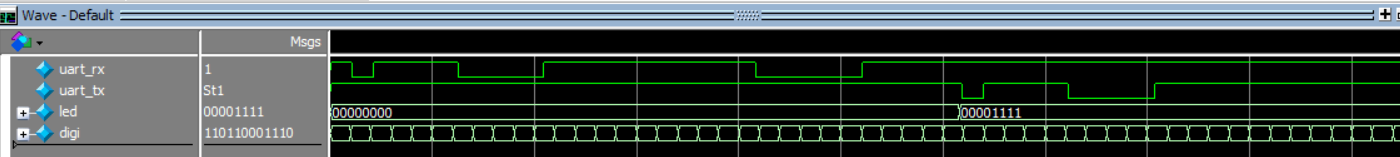
Testbench中，初始化时钟，并仿真串口输入。输入的两个数分别为0x0f和0xf0。

Watchdog为用于直观监控PC的变量。

单周期CPU：



流水线CPU：



仿真结果输出正确，符合预期。

在仿真波形正确后，第一次进行硬件调试，发现七段数码管无法被点亮。将七段数码管控制信号输出至LED，则发现控制信号始终处于高电平，这与仿真结果不符。

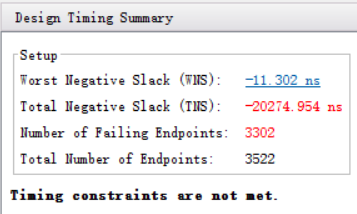
于是考虑有几种可能：1.管脚约束有误；2.定时器频率有误；3.verilog代码有缺陷。

第一种可能很快被排除，约束文件被反复检查过多次；第二种可能从仿真波形来看也是不存在的，而且在250Hz到1kHz之间尝试多个定时器频率，结果没有区别；第三种可能也是不容易让人接受的，虽然仿真没有问题的代码在综合出问题时有发生，但对vivado输出的信息不敢妄下结论。在答疑后，老师指出，vivado输出的warning中一组寄存器被vivado认为没有被使用在优化时去掉，这反映了问题所在。于是对代码的语法做了一些更改，将always(\*)语句改为assign语句，顺利地解决了这一问题。

要在仿真中找到bug的位置，首先要知道正确的波形是什么，但这并非充分条件，当在仿真中观察到bug时，要逐层深入观察，理解各个模块之间的联系，才能找到波形与预期不符的根本原因，顶层信号出现偏差可能意味着底层逻辑的问题，看到顶层的偏差就将bug定位在顶层模块是鲁莽而无效的。

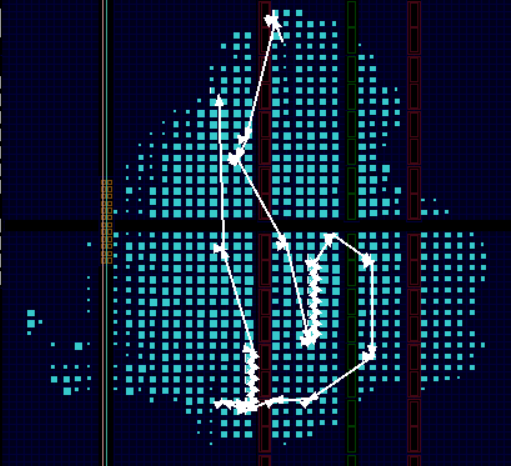
## 2.7 综合情况时序分析

在测试时序时，晶振时钟为默认的100MHz时钟，WNS为-11.302ns，如下：

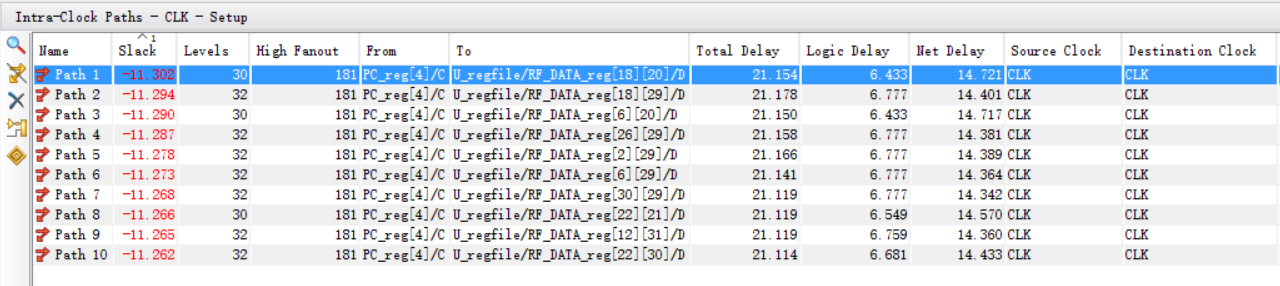


故时钟最短周期为21.302ns，最高工作频率为46.944MHz。

最长路径如下：

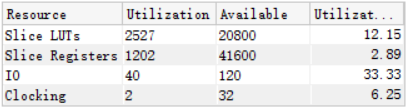


由上图不难看出，路径中有大量远距离连线，Net Delay会非常大。考察延时最大的10条路径：

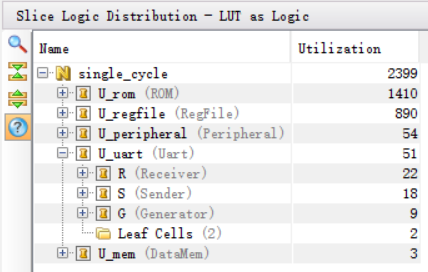


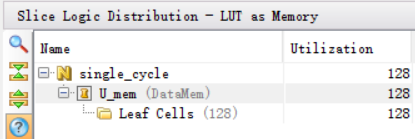
可见Net Delay确实非常大。注意到93.75%的Endpoints都不能满足时序限制，这种远距离连线是普遍存在的，时序性能仍有很大的优化空间。

## 2.8 综合情况面积分析



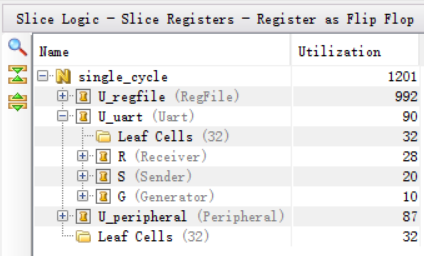
LUT使用情况如下：

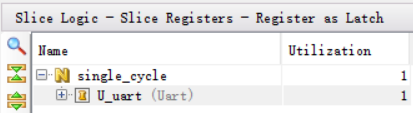


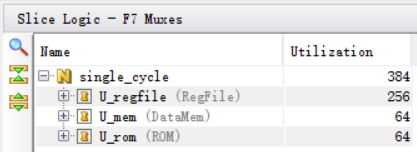


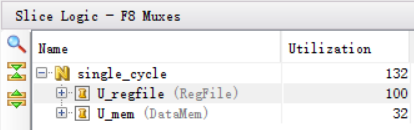
可见ROM使用的逻辑LUT占总LUT的一半以上，这一用量与汇编程序长度有关。Reg File也使用了大量逻辑LUT，超过总用量的三分之一。

寄存器使用情况如下：



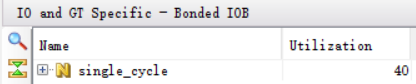


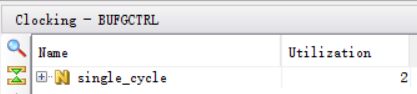




可见80%以上的寄存器被RegFile用于Flip Flop。

IO均为Bonded IOB，LED使用16个，数码管使用12个，开关使用8个，串口使用2个，复位信号使用1个，晶振时钟使用1个，共40个：





## 2.7 体会（吴昆）

由于单周期的datapath，测例，机器码主要都是我写的。感觉到单周期的实现较为简单。这可能是因为流水线的复杂程度远远大于单周期，在最后两周给我们留下了深刻的印象。另一方面，也是因为实验指导书已经画出了这一部分的datapath，按图索骥即可。而流水线由于缺乏高效率的绘图工具，往往靠想象和沟通，就显然没有一张图来的轻松。

我深深意识到一个良好的测例是非常重要的。单周期的测例我比较得意，也是思考了一段时间写出来的。甚至在之后的流水线测试时，最初使用的也是我的单周期测例，发挥了很大的作用。原本只是用来先解决掉一些简单的宽度错误的小bug，通过单周期和流水线的仿真结果一致来保证流水线基本的正确性，没想到却歪打正着有很多复杂的情形，比如分支、跳转指令以及B指令、sw指令的转发。

如果测例能够包含所有的情形，主要是所有的指令（单周期），转发、冒险（流水线），以及发生中断异常的种种可能，那么测例的通过就保证了CPU的逻辑是完全正确的。

测例没有包括异常和jr指令，而这两个在之后流水线上跑最大公约数汇编程序时发生了问题，这一方面也说明了测例的重要性；另一方面也说明流水线会测到单周期（和ALU），所以可以不必单独对单周期进行测试，节省时间。怪我咯。

而这两个错误都是control单元打表出现了偏差。jr是在之前核对时，我的想法和鹏志发生了分歧，因而标注出来准备讨论并相应修改，但后来忘了。这也说明了在多人开发中沟通是比较难而重要的。

# 三、32位流水线MIPS处理器设计（黄鹏志，吴昆）

## 3.1 设计目标

1. 设计32位流水线MIPS处理器，要求在满足2.1中提及单周期的所有要求

2. 流水线处理器的行为应与上述单周期处理器一致。

也就是说同样的汇编程序应该在单周期和流水线上得到同样的结果，要做到汇编程序开发者无需关心二者的差异做额外修改。特别地，中断和异常发生后，返回地址须与单周期的一致，详见3.5.3 精确异常。

3. 设计比较简单。

按照实验指导书上的要求进行，与标准MIPS汇编有一些差异。比如：没有branch delay slot。

## 3.2 流水线的基本设计（黄鹏志）

### 3.2.1 阶段寄存器

流水线设计与单周期设计的一个基本的不同就是每个阶段之间有一个阶段寄存器存储、分隔每个阶段的数据。为了实现流水线中每个阶段的时序顺序，一共需要在5个阶段中加入4个阶段寄存器，以及在一开始加上一个PC寄存器，其余的电路除去寄存器堆和内存之外在理论上均应该是逻辑电路，不再使用寄存器。

4个阶段寄存器分别命名为：IF\_ID\_Register，ID\_EX\_Register，EX\_MEM\_Register，MEM\_WB\_Register。所有阶段寄存器都有一个reset信号作为外接异步清零信号，低电平时寄存器清零，除PC信号改为0x80000004（即起始地址0x80000000+4）；有一个clk信号作为时钟信号，取上升沿作为寄存器存入使能；输入接口为xx\_in，输出接口为xx\_out每一个xx\_out对应一个xx\_reg，即对所有xx\_out总有：

1. assign xx\_out = xx\_reg;

所有阶段寄存器接口如下：

#### IF/ID寄存器

IF\_ID\_Register接口较少，取其完整代码作介绍：

1. module IF\_ID\_Register(clk, Flush, Instruction\_in, PC4\_in, Instruction\_out, PC4\_out,reset, IRQ, PC31, Flushed\_out);
2. input [31:0] Instruction\_in, PC4\_in;
3. input Flush, clk;
4. input reset;
5. input IRQ, PC31;
6. output Flushed\_out;
7. output [31:0] Instruction\_out, PC4\_out;
8. reg [31:0] Instruction\_reg, PC4\_reg;
9. reg Flushed\_reg;
10. always @(posedge clk or negedge reset) begin
11. if (~reset) begin
12. Instruction\_reg = 0;
13. PC4\_reg = 32'h80000004;
14. Flushed\_reg = 0;
15. end
16. else begin
17. PC4\_reg[30:0] = PC4\_in[30:0];
18. PC4\_reg[31] = PC31?PC4\_in[31]:IRQ?1:PC4\_in[31];
19. if(Flush) begin
20. Instruction\_reg = 0;
21. Flushed\_reg = 1;
22. end
23. else begin
24. Instruction\_reg = Instruction\_in;
25. Flushed\_reg = 0;
26. end
27. end
28. end
29. assign Instruction\_out = Instruction\_reg;
30. assign PC4\_out = PC4\_reg;
31. assign Flushed\_out = Flushed\_reg;
32. endmodule

instruction\_in为IF阶段输入的PC地址对应的指令，PC4\_in为PC+4后的结果，Flush为由于流水线内部操作引起的“清除”指令标志，这个清除不会将所有信号置零，而是有选择地对信号进行操作；Flushed为一个标志，若接收到Flush为1，则这个标志置为1，否则为0；PC31为当前PC\_ID的第31位作为输入，IRQ为要求中断的外部信号，当其为1，证明外部要求进入中断操作，需要和PC31共同判断调整PC+4最高位即监督位为1（见19行）。具体逻辑在之后的中断处理中讨论。

#### ID/EX寄存器

ID\_EX\_Register接口如下：

1. module ID\_EX\_Register (…);
2. input clk,Flush;
3. input reset,Flushed\_in;
4. input [2:0] PCSrc\_in;
5. input [1:0] RegDst\_in;
6. input RegWr\_in, ALUSrc1\_in, ALUSrc2\_in, Sign\_in, MemWr\_in,MemRd\_in;
7. input [5:0] ALUFun\_in;
8. input [1:0] MemToReg\_in;
9. input [31:0] PC4\_in, R1\_in, R2\_in, Imm\_in,Instruction\_in;
10. input [4:0] RdAddress\_in,Shamt\_in;
11. output [2:0] PCSrc\_out;
12. output [1:0] RegDst\_out;
13. output RegWr\_out, ALUSrc1\_out, ALUSrc2\_out, Sign\_out, MemWr\_out,MemRd\_out,Flushed\_out;
14. output [5:0] ALUFun\_out;
15. output [1:0] MemToReg\_out;
16. output [31:0] PC4\_out, R1\_out, R2\_out, Imm\_out, Instruction\_out;
17. output [4:0] RdAddress\_out,Shamt\_out;

除去IF\_ID\_Register中介绍过的的控制寄存器行为的信号，ID\_EX\_Register中主要存储的是所有在ID阶段生成的控制信号、PC+4地址、指令instruction（用于之后定位Rs、Rt）、经过了扩展的32位立即数Imm、可能需要写入的寄存器地址RdAddress、5位立即数Shamt。各\_in与\_out均为直接对应的关系。

#### EX/MEM寄存器

EX\_MEM\_Register：

1. module EX\_MEM\_Register (…);
2. input clk,reset;
3. input [2:0] PCSrc\_ID;
4. input FORWARD,Flushed\_out\_ID, Flushed\_out\_EX;
5. input [2:0] PCSrc\_in;
6. input RegWr\_in, MemWr\_in,MemRd\_in;
7. input [1:0] MemToReg\_in;
8. input [31:0] ALUResult\_in, RegToMemData\_in, PC4\_in ;
9. input [4:0] RdAdress\_in;
10. input [31:0] PCIn, PC4\_ID;
11. output [2:0] PCSrc\_out;
12. output RegWr\_out, MemWr\_out,MemRd\_out;
13. output [1:0] MemToReg\_out;
14. output [31:0] ALUResult\_out, RegToMemData\_out, PC4\_out;
15. output [4:0] RdAdress\_out;

EX\_MEM\_Register在前一个阶段寄存器的基础上，去掉了在EX阶段已经使用了的关于ALU计算的Sign、ALUSrc1、ALUSrc2、ALUFun以及写入寄存器堆的MemtoReg和读取寄存器堆的RegDst等信号，加入了PCIn, PC4\_ID两个从IF、ID阶段输入的PC+4地址与控制Forward的信号FORWARD和Flushed\_out\_ID, Flushed\_out\_EX两个判断标志。新加入的信号的意图，是为了判断可能写入26号寄存器的返回地址PC4究竟是什么。其逻辑语句如下：

1. PC4\_reg = (PC4\_in[31]||Flushed\_out\_EX)?
2. ((PC4\_ID[31]||Flushed\_out\_ID)?PCadd4:PC4\_ID):
3. PC4\_in;

这里的含义将在之后Forward的讨论中说明。

#### MEM/WB寄存器

MEM\_WB\_Register接口如下：

1. module MEM\_WB\_Register (…);
2. input clk, reset;
3. input RegWr\_in;
4. input [1:0] MemToReg\_in;
5. input [31:0] PC4\_in, Data\_in, ALUResult\_in, peri\_rddata, uart\_data;
6. input [4:0] RdAddress\_in;
7. output RegWr\_out;
8. output [1:0] MemToReg\_out;
9. output [31:0] PC4\_out, Data\_out, ALUResult\_out, peri\_rddata\_WB, uart\_data\_WB;
10. output [4:0] RdAddress\_out;

其中相比前者，去除了已经使用的信号和数据，额外新增的有：peri\_rddata, uart\_data两个来自Uart和Peripheral外设的数据，Data\_in从内存中读取的可能写入寄存器的数据。

### 3.2.2 其它主要元件及其逻辑

#### IF阶段

ROM5器件，即内存（5是编程时的最终版本号），其中已有的指令为中断异常处理指令以及题目要求的求最大公约数函数，通过PC的后31位取指令

1. ROM5 U\_rom(PC[30:0],Instruct);。

PC选择器。与单周期的简单多路选择器不同，PC的选择有更为复杂的逻辑：

1. assign PCIn=
2. (PCSrc==3'b100)?ILLOP://规定4是异常，5是中断
3. (PCSrc==3'b101)?XADR:
4. (PCSrc\_EX==3'b001&&bcomp)?ConBA:
5. (PCSrc\_EX==3'b011)?DataBusA\_Forwarded:
6. (PCSrc==3'b010)?JT:PCadd4;

PCIn为即将输入PC寄存器的新地址。容易看到，此处的PCIn根据PCSrc选择，但是和单周期设计有所不同的是，PCSrc的来源不只ID阶段，还有EX阶段，且各选择有优先级。

首先要判断的是异常和中断状态，之后优先判断分支跳转，Jr跳转，这是因为在先执行的指令中有这两个跳转之后会造成后续按照PC+4所取的指令无效，因此需要优先判断。

#### ID阶段

1. RegFile U\_regfile(reset,clk,Rs,DataBusA,Rt,DataBusB,RegWr\_WB,RdAddress\_WB,DataBusC);
2. assign AddrC=(RegDst==0)?Rd:
3. (RegDst==1)?Rt:
4. (RegDst==2)?Ra:Xp;
5. assign EXTout=EXTOp?(Imm16[15]?{16'hFFFF,Imm16}:{16'h0000,Imm16}):{16'h0000,Imm16};
6. assign LUout=LUOp?{Imm16,16'b0}:EXTout;
7. assign Shamt=Instruction\_ID[10:6];

此阶段用RegFile器件进行了寄存器堆的读写操作：从Rs与Rt中得到32位数据DataBusA与DataBusB，并根据WB阶段的寄存器写入控制信号RegWr\_WB和写入地址RdAddress\_WB与写入数据DataBusC进行写入操作。

AddrC是根据RegDst控制信号得到的写入地址，将传入ID\_EX\_Register，其中Xp为常数26，即返回地址的存储目标；EXTout是根据EXTOp控制信号得到的扩展后32位立即数；LUout是根据LUOp信号选择的阶段最终输出的32位立即数；Shamt是5位立即数。

#### EX阶段

1. assign DataBusA\_Forwarded = (ForwardA == 2'b10)?ALUOut\_MEM:(ForwardA == 2'b01)?DataBusC:DataBusA\_EX;
2. assign DataBusB\_Forwarded = (ForwardB == 2'b10)?ALUOut\_MEM:(ForwardB == 2'b01)?DataBusC:DataBusB\_EX;
3. assign ALUIn1=ALUSrc1\_EX?Shamt\_EX:DataBusA\_Forwarded;
4. assign ALUIn2=ALUSrc2\_EX?Imm\_EX:DataBusB\_Forwarded;
5. branch\_compare U\_branch\_compare(bcomp,ALUIn1,ALUIn2,ALUFun\_EX[4:1]);
6. ALU U\_alu(ALUOut,ALUIn1,ALUIn2,ALUFun\_EX,Sign\_EX);
7. assign \_EXTout\_shift2={Imm\_EX[29:0],2'b00};
8. assign ConBA=PC4\_EX+\_EXTout\_shift2;

EX阶段主要是ALU计算的阶段。DataBusA\_Forwarded是DataBusA经过Forward选择之后得到的32位数据，DataBusB\_Forwarded同理，其选择是通过forward单元生成的控制信号Forward决定的，分别可能来自上一周期ALU的输出、当前周期WB的DataBusC和从寄存器读出的原始数据；而ALUIn1和ALUIn2是根据ALUSrc选择得到的ALU输入； \_EXTout\_shift2是32位立即数左移两位后得到的数，与PC4\_EX相加用于选址。

branch\_compare是独立于ALU的一个特别的比较单元，主要用于分支跳转的计算。将这个设计独立于ALU而不利用ALU的相减功能的主要目的是减小扇出从而减短关键路径，将在之后详细解释。

#### MEM阶段

1. DataMem U\_mem(reset,clk,MemRd\_MEM,MemWr\_MEM,ALUOut\_MEM,RegToMemData\_MEM,mem\_rddata);
2. Peripheral U\_peripheral(reset,clk,MemRd\_MEM,MemWr\_MEM,ALUOut\_MEM,RegToMemData\_MEM,peri\_rddata,led,switch,digi,irqout);
3. Uart U\_uart(sysclk, clk, reset, uart\_tx, uart\_rx, MemRd\_MEM, MemWr\_MEM, ALUOut\_MEM, RegToMemData\_MEM, uart\_data, rx\_irq, tx\_irq);
4. assign IRQ=rx\_irq|tx\_irq|irqout;

MEM阶段主要在DataMem中进行了读写，并且接入外设Peripheral和Uart。中断信号IRQ也在这里生成，3中中断任一满足就会触发。

#### WB阶段

1. assign DataBusC=(MemToReg\_WB==0)?ALUOut\_WB: (MemToReg\_WB==1)?(Data\_WB|peri\_rddata\_WB|uart\_data\_WB):PC4\_WB;

WB阶段主要生成了写回寄存器的32位数据DataBusC，可能是WB阶段存储的之前的ALU输出，或者之前的外设输出Data\_WB、peri\_rddata\_WB、uart\_data\_WB，或者是写入一个用来跳转的地址PC4\_WB。寄存器堆写入已经在ID阶段讨论。

#### 非单个阶段内器件

1. HazardDetection U\_HazardDetection(PCSrc\_EX, bcomp, PCSrc, MemRd, /\*ID\_EX\_Rt\*/Rt,Instruct[25:21],/\*IF\_ID\_Rt\*/Instruct[20:16], bubble,ID\_EX\_Flush,IF\_ID\_Flush, EX\_MEM\_FORWARD);
2. forward U\_forward(ForwardA,ForwardB,RegWr\_MEM,RegWr\_WB,RdAddress\_MEM,RdAddress\_WB,Rs\_EX,Rt\_EX);
3. always @(negedge reset or posedge clk)
4. begin
5. if (~reset)
6. IRQ\_sync<=0;
7. else
8. IRQ\_sync<=IRQ;
9. end

分别是冒险检测单元HazardDetection、Forward控制指令生成单元forward和IRQ寄存器。

其中，用于forward比较的控制信号和寄存器地址来自于MEM阶段和WB阶段，因为在这两个阶段的数据可能是刚刚被ALU计算出的或者即将写入寄存器堆而未写入的，这两种情况满足forward到EX阶段的ALU进行计算的需求。之所以要将IRQ加上寄存器变为时序信号，是为了避免IRQ信号在时钟上升沿前产生突变，导致流水线逻辑变换时间不够，造成错误。

## 3.3 转发、停和取消指令（吴昆）

这一节主要介绍转发forward模块(Forward.v)和冒险检测HazardDetection模块(Hazard\_Detection.v)。我们在具体的实现中，由于共通性，冒险检测单元不但负责load-use hazard的检测和停信号的产生，以及跳转指令、分支成功以及异常中断时清空寄存器的信号的产生。

### 3.3.1 冒险和异常中断

流水线中需要小心应对的是冒险和异常中断。

冒险中主要考虑的是数据冒险和控制冒险。对于数据冒险，转发和load-use hazard检测可以全部解决。流水线的数据冒险主要依靠转发解决，对于转发无法解决的load –use hazard，冒险检测单元检测并产生新号将流水线的前几个阶段停住（根据具体实现）。

对于控制冒险，由于不存在branch delay slot，在我们的实现中主要表现为分支和跳转指令改变PC后在IF（和ID）阶段的指令需要取消，我们在检测到这一情形后清空相应的寄存器。

在异常、中断发生时也需要取消指令。具体实现时，我们还将控制单元响应异常、中断的输出直接发射到EX/MEM。

### 3.3.2 转发

转发控制策略参见forward模块（Forward.v），参照了教材上的代码。

1. assign ForwardA = (EX\_MEM\_RegWr&&EX\_MEM\_Rd!=0&&EX\_MEM\_Rd==ID\_EX\_Rs)?2'b10:
2. (MEM\_WB\_RegWr&&MEM\_WB\_Rd!=0&&MEM\_WB\_Rd==ID\_EX\_Rs)?2'b01:
3. 2'b00;
4. assign ForwardB = (EX\_MEM\_RegWr&&EX\_MEM\_Rd!=0&&EX\_MEM\_Rd==ID\_EX\_Rt)?2'b10:
5. (MEM\_WB\_RegWr&&MEM\_WB\_Rd!=0&&MEM\_WB\_Rd==ID\_EX\_Rt)?2'b01:
6. 2'b00;

但有两个地方教材有问题。首先，转发源教材上写的是Rd，但实际上早执行的指令写入的寄存器有Rs和Rd两种，为此我们在ID阶段推断得到写入寄存器的编号：

1. assign AddrC=(RegDst==0)?Rd:
2. (RegDst==1)?Rt:
3. (RegDst==2)?Ra:Xp;

并不断地向之后的阶段传，命名为RdAdress\_EX, RdAdress\_MEM等，用这个信号来替代教材上的Rd，就可以修正这个由于教材考虑情形较为简单而没有涉及到的问题。

第二，是教材对MEM冒险控制（即第343行和346行）策略，对Rs增加了and (EX/MEM.RegisterRd != ID/EX.RegisterRs)条件，对Rt为and (EX/MEM.RegisterRd != ID/EX.RegisterRt)。但我们根据推导发现是不需要的，并且增加这条在测试中导致了错误。例如：EX阶段的指令是跳转指令，而ID阶段的Rs是某寄存器，且此寄存器号码恰好等于EX指令中对应Rd指令的地址。这样尽管对于EX阶段的forward判定由于EX\_MEM\_RegWr==0而无效，但是如果MEM阶段需要对ID阶段的Rs执行forward，就会导致由于有EX/MEM.RegisterRd == ID/EX.RegisterRs（尽管EX阶段的所谓“Rd”是没有意义的），从而导致无法执行转发。事实上，不需要这一条判断语句，逻辑也是正常可执行的。

对于流水线的datapath做了相应修改，转发数据的选择在EX阶段：

1. assign DataBusA\_Forwarded = (ForwardA == 2'b10)?ALUOut\_MEM:(ForwardA == 2'b01)?DataBusC:DataBusA\_EX;
2. assign DataBusB\_Forwarded = (ForwardB == 2'b10)?ALUOut\_MEM:(ForwardB == 2'b01)?DataBusC:DataBusB\_EX;
3. assign ALUIn1=ALUSrc1\_EX?Shamt\_EX:DataBusA\_Forwarded;
4. assign ALUIn2=ALUSrc2\_EX?Imm\_EX:DataBusB\_Forwarded;

先通过转发控制信号得到DataBusA/B\_Forwarded，替代原来的DataBusA/B输入ALUSrc控制的多路选择器，DataBusB\_Forwarded在经过EX/MEM后替代原来的DataBusB输入存储器。

这里非常考验理解的是转发在流水线设计中的作用是修正，以此使完成转发的数据DataBusA/B\_Forwarded与单周期中原DataBusA/B一致，因此原先需要DataBusA/B作为输入的地方都要改成DataBusA/B\_Forwarded。一开始我们只将转发作为ALU的输入，却没有考虑到存储器的输入也要改成转发，导致了错误。

### 3.3.3 阶段寄存器清空

在跳转以及分支成功时，之前发射的指令需要取消，具体实现方法是清空对应的IF/ID以及ID/EX寄存器。由于精确异常（维护正确的返回地址）的需要，寄存器上的PC+4信号不会被清空。在优化中我们会提到，其实清空只需将该寄存器准备发射的控制信号重置为nop，而数据可以不清空。

由于B指令是在EX阶段判断是否能跳转并在下一个时钟上升沿发射跳转后地址的指令（如果成功），这就意味着IF、ID需要清空，jr也是类似的情形。而j和jal只需清空IF指令。当然jr，j和jal是任意情况都会跳转的。

1. assign bID\_EX\_Flush = (EX\_PCSrc==3'b001) && (ALUOut\_0==1);
2. assign jIF\_ID\_Flush = ID\_PCSrc==3'b010;
3. assign jrIF\_ID\_Flush = EX\_PCSrc==3'b011;
4. assign jrID\_EX\_Flush = jrIF\_ID\_Flush;
5. assign IF\_ID\_Flush = jIF\_ID\_Flush|jrIF\_ID\_Flush|bIF\_ID\_Flush|int\_expt\_Flush;
6. assign bIF\_ID\_Flush = bID\_EX\_Flush;
7. assign ID\_EX\_Flush = bID\_EX\_Flush|jrID\_EX\_Flush|int\_expt\_Flush;
8. assign EX\_MEM\_FORWARD =int\_expt\_Flush;

判断分支指令是否导致取消的逻辑如第一行所示，先判断EX阶段的PCSrc也就是判断是否是分支指令，再根据ALU输出最低位判断是否成功跳转。

跳转指令只需根据相应节段的PCSrc判断即可。

我们在判断是否有需要取消指令的指令后，再分别得到各级的清空信号。

对于中断及异常，由于中断发生时，会在下一个时钟上升沿发出控制信号，此时MEM/WB阶段的指令已经产生或部分产生作用，无法取消。为了方便实现精确异常，我们将没有发生作用的指令全部清空，也就是在下一个时钟上升沿IF/ID，ID/EX将输出nop控制信号，同时我们将控制器响应异常中断的控制信号直接发送至EX/MEM让它在下一个时钟上升沿发射。

在几个阶段寄存器中，我们把清空指令的输入信号称为Flush，而异常中断发生时直接输出控制信号并清除（覆盖）原先的输出信号的独特输入信号成为Forward。下面给出阶段寄存器响应Flush和Forward信号的逻辑作为参考，比较trivial就不解释了。

ID\_EX\_Register中的Flush信号响应逻辑如下：

1. if(Flush) begin
2. PCSrc\_reg = 0;
3. RegWr\_reg = 0;
4. MemWr\_reg = 0;
5. MemRd\_reg = 0;
6. PC4\_reg = PC4\_in;
7. Flushed\_reg = 1;
8. end

EX\_MEM\_Register中的Forward信号响应逻辑如下：

1. if (FORWARD) begin
3. RegWr\_reg = 1;
4. MemWr\_reg = 0;
5. MemRd\_reg = 0;
6. MemToReg\_reg = 2;
8. PC4\_reg = (PC4\_in[31]||Flushed\_out\_EX)?((PC4\_ID[31]||Flushed\_out\_ID)?PCadd4:PC4\_ID):PC4\_in;//wk10upd TODO: what's the meaning?
9. RdAdress\_reg = 26;//Xp
10. end

### 3.3.4 冒险检测单元

HazardDetection模块（Hazard\_Detection.v）实现了检测load-use hazard，并且由于具体的分工，我们在这里实现了B、J指令以及异常中断产生这些需要清空阶段寄存器的情形下清空信号的产生。

其中load-use hazard的检测如下：

1. assign bubble = (~ID\_EX\_Flush || ~IF\_ID\_Flush) && ID\_EX\_MEMRd && ((ID\_EX\_Rt==IF\_ID\_Rs && ID\_EX\_Rt!=0) || ((ID\_EX\_Rt==IF\_ID\_Rt)&&(ID\_EX\_Rt!=0)));

在一般情形下，如果load和use指令都有效，不会被上述情形清空，那么判断逻辑就是第2、3行。

这与书上一致，非常容易理解。当lw在ID/EX寄存器及use指令在IF/ID寄存器上时，产生bubble信号，使得IF阶段和PC暂停一个回合，ID向后传一个nop。

但是，当IF/ID及ID/EX寄存器需要清空的情形，只要清空，因为load和use指令不会发生作用，并且在下个周期就会被清空，因此不用考虑load-use hazard产生的bubble，这就是第一行的逻辑。

## 3.4 编码中遇到的bug

第一个bug是Forward在一定情况下会失效——当两条指令间已经隔了两条指令时，应该不需要Forward，但是却出现了结果的错误。通过仿真排查，我们发现这是由于regfile没有实现基本要求：即先写后读造成的。我们将regfile的写入改为了时钟下降沿写入，解决了这个问题。

第二个bug是混淆了reset信号和Flush信号的含义。因为都是“清零”，在一开始的设计中我们将reset和Flush设计为阶段寄存器中的同一接口，但是却发现两者在功能上的差距很难忽视（例如Flush再清零时没有清零PC+4，以及需要有一个标志位对寄存器是否曾经被Flush加以区分等），因此在之后设计成了两个信号。

除此之外，我们还发生了许多小的bug，例如变量忘记声明宽度或者宽度声明错误，变量名大小写错误，Control元件生成信号的笔误，reset信号是高电平还是低电平复位等信号取非的错误。在多人合作时，由于沟通问题，经常出现这样的小错误拖慢进度。这是我们应该吸取的教训。

## 3.5 流水线的中断和异常处理（黄鹏志）

我们认为，流水线是单周期的加速，因此在行为上要和单周期保持一致，也就是说汇编程序的设计人员可以不加考虑硬件地在流水线和单周期上跑相同的代码而得到正确的结果，并不用进行额外的调整。（流水线涉及目标2）

精确异常（precise exception）是流水线设计中一个非常重要的概念，且实现复杂：通过硬件的设计，实现任何情形下中断处理完后都能够返回到正确的地址，与单周期相同情形下的返回地址一致。这就要求在进入中断时硬件能够向Xp写入一个正确的返回地址，而这在分支、跳转情形时非常复杂。通常实现方法是将过去的几个PC存储在一张表上，然而即便如此也要分情形，找到正确的地址。

异常与中断处理相似，存地址并跳转到指定的处理地址，再由汇编语言实现在进入死循环（详见汇编代码），这即是对于未定义指令的处理。中断处理更为复杂，因此在以下讨论中大部分以中断为核心。

这一节我们先阐述设计的原则，然后讨论一些具体的实现细节。

### 3.5.1 原则

1. 中断异常处理优先级最高。

Control在产生新的信号时如果有需要处理的中断或异常，会优先处理而不考虑准备解析的指令。这样保证了对两者处理的及时；而且，这个原则方便我们对清空寄存器的实现。

2. CPU在响应IRQ信号前，需要将IRQ同步为时序信号。

这在之前已经有讨论，如果在任意时刻IRQ都有可能进入时序电路并使信号产生变化，将使得输出不可预料。

3. 执行过程和结果要和已经验证的单周期的表现一致。

也就是说在单周期下能够跑的程序，在流水线上应该有同样的行为和结果，不会发生错误。这就要求我们。特别是连续中断、分支指令等情况下的处理。

### 3.5.2 中断处理触发的讨论

由定义，中断处理的触发需要满足：IRQ为1且PC监督位不为1，代码如下：（在IF\_ID\_Register中）

1. PC4\_reg[31] = PC31?PC4\_in[31]:IRQ?1:PC4\_in[31];

PC31为ID阶段的PC+4的监督位，当其为1，无论IRQ为何值，都不会触发中断改变监督位；只有当PC31为0，才通过讨论IRQ判定是否改变PC监督位。这样的操作避免了连续的中断（IRQ为1的情况有多种，处理完一种后另一种仍未处理的话，会造成IRQ连续为1不变）下，PC监督位也连续为1，造成Control单元无法正常生成第二个中断的中断处理控制信号。

### 3.5.3 精确异常

这一节讨论如何做到任何情形下维护正确的返回地址。

在中断处理开始时，我们要跳转到处理指令处；结束后，我们需要跳转回中断前的地址，这即是一个jal指令的表现。流水线的处理和单周期有很大不同。

这个跳转的基本要求和普通跳转相同，也就是尚未对寄存器堆和内存造成影响。我们处理的方式就是将即将进入EX\_MEM\_Register的EX阶段指令改为了一个jal指令，而由于EX及之前的指令均未对寄存器堆和内存造成影响，也就是说EX阶段、ID阶段或者IF阶段的指令所对应的地址就应该作为返回地址。因此作出的改变之一，就是将此三个阶段的PC+4都接入了EX\_MEM\_Register。

选择PC的代码如下：（在EX\_MEM\_Register中）

1. PC4\_reg = (PC4\_in[31]||Flushed\_out\_EX)? ((PC4\_ID[31]||Flushed\_out\_ID)?PCadd4:PC4\_ID):PC4\_in;

PC4\_in来自EX阶段，PC4\_ID来自ID阶段，PCadd4来自IF阶段。

作为返回地址需要满足两个条件：所属的寄存器没有由于BEQ等指令被Flush过（例如BEQ此时正在EX\_MEM\_Register且生效，那么下个时钟上升沿进入EX\_MEM\_Register的如果简单选用EX阶段的PC+4就会造成返回地址是一个本不被执行的指令），且寄存器中不是中断处理指令（例如连续中断时，来自EX的PC+4很可能是监督位为1的中断处理指令所在地址）。之前我们在IF\_ID\_Register和ID\_EX\_Register加入了Flushed标志来检测阶段寄存器中的指令是否已经被“清零”，因此只需要检查PC监督位和Flushed标志即可。依次往前检查，三个阶段中第一个符合要求的PC+4就是我们要的返回地址+4。

这样的讨论避免了连续中断时错误返回属于中断处理的地址，以及错误地返回并不会执行的由于跳转引起的bubble的地址，令中断处理十分安全可靠，同时不可避免地增加了时钟周期。

## 3.6 针对频率的优化

这一部分单独形成了优化报告，利用Vivado时序报告分析关键路径的方法以及失败的优化尝试详见优化报告，这里仅简述成功的优化策略。

通过下文前两小节改进，我们使得worst negative slack从12ns下降为9.9ns，并且能够在FPGA上以100MHz速度跑。通过之后几项改进，我们最终降低worst negative slack至9.55ns（Vivado 2014）。

### 3.6.1 ALU组件去除级联依赖（吴昆）

在Vivado中观察时序报告，我们发现最长路径为转发->ALU输入->减法->比较器->PC。经过考虑，我们认为这是一条发生转发的b指令。

最长路径中出现了ALU输入->减法->比较器，由于我们对于相等的实现是减法结果Z输入到比较器中，导致了最长路径。因此我们对比较器单独设计逻辑，不依赖加法、减法器的输出。

### 3.6.2 B指令独立比较器（吴昆）

经过上述调整后，关键路径明显缩短，但仍然经过了减法，我们分析后觉得这是相等判断==综合器自动使用作差实现。由于输入ALU要经过控制信号的重重判断，我们认为为B指令布置独立的比较器可以改善性能，这样做了以后发现果真如此。

通过亮相调整，我们的电路可以在FPGA上以100MHz的速度运行。

### 3.6.3 控制信号扇出的改进（黄鹏志）

继续优化，仔细观察关键路径的path details，我们发现一些控制信号的扇出十分大（fo=169， net delay= 1~2ns），严重拖慢了频率。因此我们针对一些控制信号进行了优化：

#### Flush信号

在IF\_ID\_Register和ID\_EX\_Register的Flush信号使用频繁且控制了很多寄存器的值，扇出巨大。由于Flush指令的效应是取消对应指令的影响，因此指令只要不影响：（1）PC选择；（2）寄存器堆的数据；（3）内存的数据，就可以了。

因此，对于IF\_ID\_Register，只需要在Flush==1时额外执行

1. Instruction\_reg = 0;
2. Flushed\_reg = 1;

对于ID\_EX\_Register，只需要额外执行

1. PCSrc\_reg = 0;
2. RegWr\_reg = 0;
3. MemWr\_reg = 0;
4. MemRd\_reg = 0;
5. Flushed\_reg = 1;

即可，其它和Flush==0时一致。

这样就大大减少了Flush的扇出，减小了时钟周期。

#### FORWARD信号

在EX\_MEM\_Register中的FORWARD信号与Flush相同，减少了部分不必要的扇出。

## 3.7 仿真与硬件调试

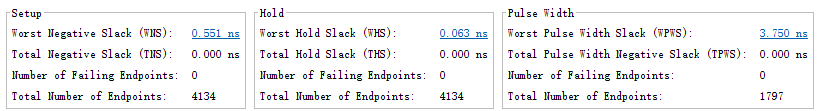
流水线的调试过程主要是通过multisim完成的。在确认单周期运行无误的情况下，在流水线上运行我们所编写的对多种复杂情况进行考量的测试程序并和单周期运行结果比对，从而由不同之处一步一步找到设计中的缺陷或者编写中的错误。

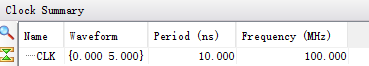
在初期的常见错误就是变量未定义（导致长度错误），变量名大小写（由于多人参与编写导致）不一致，Control打表出错等等编写错误。由于熟悉的模型为单周期模型，流水线在各种逻辑上与单周期都有很大的不同，后期即开始考虑逻辑错误。例如IRQ需要变为时序信号，在PC选择过程中PCSrc控制信号的来源与优先级（单周期只需要一个多路选择器即可实现），forward单元与冒险检测单元的控制信号来源与判断逻辑等，都是在这个过程中通过发现流水线和单周期表现不一致而讨论的得到的。

最终将题目要求程序在流水线上仿真，在VIVADO上烧录测试。我们发现在硬件测试中，代码的always @(\*)需要改为assign构成的逻辑语句，否则无法正确烧到板子上，说明Verilog可以先出来并不代表FPGA一定支持。硬件描述语言归根结底是电路的描述，要在实现时有硬件的概念和印象，必须要对此加以注意。最终成功在硬件上运行了要求的程序。

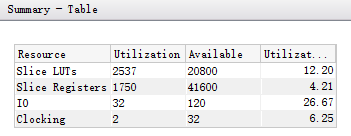
## 3.8 综合情况

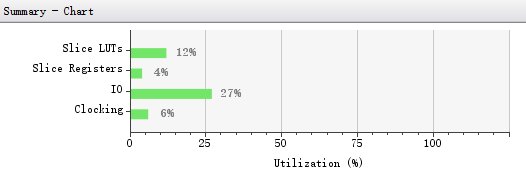
#### 时序





#### 面积





## 3.9 体会（吴昆，黄鹏志）

流水线是这个实验中耗时最长的，也是我们最有收获的一次经历。

设计流水线本身就是困难而有趣的，即便在教材告诉我们一个大概的框架乃至每个元件的可能实现细节后。比如教材转发源只写了Rd，而没有考虑Rs的情形，又比如MEM转发部分按照教材的写法确实是错误的，必须把他深思熟虑后新加的条件去掉才正确。和写代码一样，只是凭着惯性，不思考写出来的确实会有很多疏漏，无论是bug还是没有考虑到的边角条件。

而通过设计、解决bug，我们对于教材中的转发、冒险检测的逻辑设计有了更深刻地掌握。最有趣的是，鹏志长考后得出冒险检测单元和转发的输入信号一个应该直接连，另一个则需要连在下一级阶段寄存器的输出以等待一个时钟上升沿，竟然与教材上上课未及的图完全一致。

很多时候，我们都是在debug中不断修正设计，既要保证逻辑的正确性，又要保证在尽可能不改变原先的设计的原则和框架的条件下，把逻辑改对。

设计是有很大的自由度的，选择如何设计则是根据当初或者设计中逐渐明晰的原则，以及实现的复杂程度。我们一开始就明确了流水线的行为要跟单周期一样，这对我们的设计结果产生了重要的影响。比如说中断后的返回地址是由硬件维护，还是如同书上所说存储一个历史PC表，然后由中断处理程序来进行选择？因为单周期的返回地址不像流水线一样，后者在未经设计下会在分支等情形下将错误的PC+4值存入Xp，前者则始终会存入正确的PC+4。由于流水线要和单周期行为一致，也就是说中断处理程序不变，因此我们通过硬件实现了精确异常。

设计中最复杂的依然是精确异常，在分支和连续中断后情况变得非常棘手。我们在写好后，发现跑到beq会有bug，原因在于没有考虑到beq执行到EX/MEM阶段时控制单元输出中断控制信号到ID/EX的情形。在这个过程中，我们既要考虑调整时要保证什么不变（原则），又要考虑可能的方案和他们的成本，还要考虑到这样的解决方法能不能较好地解决问题并不引起新的bug。在这个过程中，锻炼了我们全面思考各种corner case的能力，这在写代码中非常重要；也锻炼了我们的设计和决断能力。

我问过某位学长他怎么做到精确异常的，他说比较简单，原因在于他们引入了branch delay slot。但我们一开始就没有这个设计，所以从始至终都要原谅自己当初的选择。其实也是从学长的回答中我猛然意识到我们没有这个引入branch delay slot，有很多元素乃至理念是在当初明确设计原则时没有明显讨论甚至觉察到的，尽管branch delay slot还是比较明确的一种，通过事后的反思却可以发现他们对设计带来的影响，发现不同的选择带来的不一样的结果。

在无尽的debug中，我们深深感到，今后在编写类似程序时要加强成员的沟通，这样能避免很多不必要的BUG；在前期工作中一定要充分研究程序的逻辑，否则通过后期debug弥补会十分耗时耗力；要充分了解编程平台所支持的语法逻辑，避免跨平台时出现的问题。

有一些人生经验是老师和助教分享过的，只是淡淡带过，当时听了很有道理就不管了，在自己动手时才发现这些是我们破费周折后才引入的。由于明白了这些结论针对的实际情形，以及经过了推理过程得到了同样的结果，因而大大加深了对它们的印象。比如regfile改为下降沿写，分支指令不使用ALU而使用独立的比较器等等。我总有一种感觉，一开始想的时候能够有一种宏观的理解，但是具体写代码的时候就会只聚焦在自己当前的任务和需要解决的问题上了，那些长者们的人生经验就没有想起；而当调试正确性和性能时，就会发现这些结论的得出理所应当。

# 四、机器码转换程序（吴昆）

## **4.1 设计目标**

将比较简单的MIPS汇编程序转换为机器码。

## 4.2 设计方案

我们实现了一个编译程序，可以转换简单的MIPS汇编程序，它满足：1. 只有代码，没有.data区和.text标记；2. 指令在实验指导书中规定的30种中；3. 不存在伪指令。

编译程序的一个难点是分支、跳转指令的实现，在汇编语言中我们会在这些指令中指定一个标签，并在代码的相应位置放上标签，但是这个标签的地址在此时并没有明确。我们采用的是两步循环的办法：第一遍循环顺序翻译所有可以翻译的指令，并把分支和跳转指令的立即数部分先当作是0，同时记录标签和分支、跳转指令的地址。第二遍循环（重定位）将所有的分支、跳转指令的立即数填上相应的值，由于此时已经知道了目标地址和该指令所在地址，因此是可以完成的。

大致流程是，对于一条指令，先根据OPCODE，查表得到OPCODE（或者FUNCT），再将各个部分（比如$rs,imm等）翻译成数，然后根据各部分的偏移相加。

当然之前有一些对文件字符串预处理的部分，目的是使得之后的处理可以比较方便地找到指令字符串和各“参数”（比如$rs,imm等）字符串，第二是使得空格、换行的存在与否不会影响处理的结果。

## **4.3 主流程**

由于处于次要的地位，且代码内容比较多，仅对主流程进行阐述。

具体可以参见注释，其实可以直接看converter.py。

主流程代码如下，输入一个诸如[“lw”,”$s1”,”4($sp)”,”sra”,”$s2”,”$t2”,”0x1”],输出一个复杂dict，key为地址数，val为一个dict，其中包括了”machine\_code”项，value就为翻译后的机器码 。

ef convert\_assembly(clean\_code,reverse=False):  
 *"""  
 将clean\_code转换为机器码，组织格式为  
 {address:{"machine code":machine\_code,("relocation":relocation,etc.)}}  
 当reverse为True时，j和beq不再使用label，而是将其本应作标签的参数当作立即数进行相应跳转  
 :return: dict  
 """* i = 0  
  
 # 将指令转换为{address:{"machine\_code":machine\_code(,"relocation":relocation,"tag":tag)}}  
 converted\_instruction = {}  
 tag\_address = {} # tag\_name:address  
 text\_address\_relocate = [] # 记录了需要重定向的指令的地址  
 address = TEXT\_ADDRESS\_START\_POSITION  
 while (i < len(clean\_code) - 1):  
 if not converted\_instruction.get(address):  
 converted\_instruction[address] = {}##有标签时这条指令会使标签消失  
 # 遇到了一个标签  
 if clean\_code[i].find(":") != -1:  
 converted\_instruction[address]["tag"] = clean\_code[i][0:-1]  
 tag\_address[clean\_code[i][0:-1]] = address  
 i += 1  
 continue  
 # 遇到了一个指令  
 converted\_instruction[address]["instruction"] = clean\_code[i]  
  
 param\_number, param\_types\_offsets, (constant, constant\_offset), instruction\_type = get\_instruction\_type(  
 clean\_code[i])  
 converted\_instruction[address]["origin\_line\_list"] = [clean\_code[i]] + ["Not parsed"]\*param\_number  
 converted\_instruction[address]["instruction\_type"] = instruction\_type  
 machine\_code = constant << constant\_offset  
 for param, param\_type, param\_offset in param\_types\_offsets:  
 converted\_instruction[address]["origin\_line\_list"][param+1]=clean\_code[i+param+1]  
 if param\_type == "immediate" or param\_type == "shamt": ##写成param\_type=="immediate" or "shamt"是不行的  
 try:  
 num\_specified = int(clean\_code[i + param + 1]) #默认十进制，和其他的都不同  
 except Exception:  
 num\_specified = int(clean\_code[i + param + 1],16)  
 ##第一次写漏写了立即数取反  
  
 if num\_specified < 0:  
 assert param\_type == "immediate", "should not have negative number as shamt" # 只可能负数出现在imm，而不是shamt  
 num\_specified = get\_negint\_represent(num\_specified, IMM\_MODULUS[  
 converted\_instruction[address]["instruction\_type"]])  
 num\_specified\_alternative = ~(-num\_specified) % ( ##漏掉了负号，应该先得到绝对值，再来一波bit manipulation  
 1 << IMM\_MODULUS[converted\_instruction[address]["instruction\_type"]]) + 1  
 assert num\_specified == num\_specified\_alternative  
 machine\_code += num\_specified << param\_offset  
 elif param\_type == "register":  
 machine\_code += decode\_register(clean\_code[i + param + 1]) << param\_offset  
 elif param\_type == "SL\_register":  
 reg\_no, imm = decode\_sl\_register(clean\_code[i + param + 1])  
 machine\_code += reg\_no << param\_offset[0]  
 machine\_code += imm << param\_offset[1]  
 elif param\_type == "relocate":  
 converted\_instruction[address]["relocate"] = clean\_code[i + param + 1]  
 text\_address\_relocate.append(address)  
 converted\_instruction[address]["relocate\_type"] = clean\_code[i] # *TODO:remove this and following line* assert converted\_instruction[address]["relocate\_type"] == converted\_instruction[address]["instruction"]  
 else:  
 raise ValueError("unexpected param\_type")  
 converted\_instruction[address]["machine\_code"] = machine\_code  
 i += param\_number # 寄存器/立即数  
 # 解析下一条指令  
 i += 1  
 address += 4  
  
 # 对跳转指令进行重定向，注意不同指令的差异：  
 # B开头的是[17:2]位，jr,jal也是，*TODO:似乎都是？* for text\_address in text\_address\_relocate: ##这一步一开始直接跳过了，原因在第一次循环时没有向这个数组加入东西  
 if not reverse:  
 destination = tag\_address[converted\_instruction[text\_address]["relocate"]] ##写成了address  
 converted\_instruction[text\_address]["relocate"] = destination  
 # 对重定向的地址的机器码加上地址（立即数部分）：  
 ##没有考虑B指令应该是跳转地址和PC+4的差，并注意还是需要处理负偏移  
 if converted\_instruction[text\_address]["instruction\_type"] == "I": # 分支指令  
 machine\_code\_offset = ((destination - text\_address - 4) >> 2) % (1 << 16)  
 if machine\_code\_offset < 0:  
 machine\_code\_offset = get\_negint\_represent(machine\_code\_offset, 16)  
 assert IMM\_MODULUS[converted\_instruction[text\_address]["instruction\_type"]] == 16  
 else: # 跳转指令  
 machine\_code\_offset = (destination >> 2) % (1 << 26) ##写成了直接对IMM\_MODULUS取模，但1<<后者才是真正的模值  
 assert IMM\_MODULUS[converted\_instruction[text\_address]["instruction\_type"]] == 26  
 assert BJ\_MODULUS[converted\_instruction[text\_address]["instruction"]] == IMM\_MODULUS[  
 converted\_instruction[text\_address]["instruction\_type"]] # *TODO：去除BJ\_MODUUS,条件是，立即数只有J是26位，其它全部16位* converted\_instruction[text\_address]["machine\_code"] += machine\_code\_offset << 0  
 else:#全是立即数  
 converted\_instruction[text\_address]["machine\_code"]+=int(converted\_instruction[text\_address]["relocate"],16)  
 return converted\_instruction

## **4.4 体会**

在实现中出现了许多bug，其中比较重要的几个是1. 分支指令的立即数计算采用了和跳转指令一样的方法，而没有用(目标地址-指令地址-4)>>2；2. 机器码采取各个部分值移位偏移量求和得到，忘记对立即数取补码，导致负数直接加到机器码上出现错误；3. sw，lw的立即数指定方法并非单独一项，而是类似于4($sp)的形式，在对字符串进行操作时没有考虑到这一点。

这些bug的发现，或者是输出和正确的机器码不一致，或者是在运行中直接报错。在这里我们非常开心的发现可以利用MARS模拟器得到正确的机器码翻译结果，进行比较。我们使用了2段程序，覆盖了所须30个指令的大部分，通过调试都可以正确得到结果。

对于未能详尽的指令，分析如下：翻译的正确性主要在于：1. 每一个指令都能够正确地给出各个常量(OPCODE或FUNCT)以及常量、寄存器、立即数的移位量；2.重定位能够正确给出目标和指令地址，并且对分支、跳转两种立即数形式给出正确的结果；3.正确地将字符寄存器表示转换成5位数字，以及将负立即数表示成补码。这些在两个程序的正确输出中都得到了证明，而且没有覆盖到的指令主要是非beqB指令以及R指令，而sw/lw等比较独特的都经过了测试。我们说R指令的常量以及各组成部分移位量是差不多的，只要FUNCT表没有写错就不会翻译错，而B指令主要在于立即数的计算，也得到了确保。因此，我们的编译程序是正确的。

还写了一个提取ROM.v中机器码翻译成近似汇编语言的程序translator.py（主要问题是跳转、分支指令没有翻译出标签，直接跟着立即数）。

# 五、汇编程序（李登昊）

## 5.1 设计目标

使用汇编，实现更相减损术求最大公约数

## 5.2 设计方案

### 5.2.1 更相减损术

起始时操作数存储在$a0,$a1两个寄存器中，比较二者大小，用数值非较小的操作数减去另一个操作数，直到某次结果为零，将操作数数值存入$v0。

1. Check:
2. slt $t0, $a0, $a1
3. bne $t0, $zero, Less
4. More:
5. sub $a0, $a0, $a1
6. bne $a0, $zero, Check
7. add $a0, $zero, $a1
8. j Result
9. Less:
10. sub $a1, $a1, $a0
11. bne $a1, $zero, Check
12. Result:
13. add $v0, $a0, $zero
14. sw $v0, 12($t9)
15. sw $v0, 24($t9)
16. addi $a0, $zero, 0
17. addi $a1, $zero, 0
18. addi $v0, $zero, 0
19. j Wait

### 5.2.2 开始运行

CPU开始运行时，需要从内核态进入用户态。由于只有jr和jalr指令可以将PC[31]置零，首先通过jr指令跳转至用户态。之后初始化定时器：

考虑到单周期CPU工作频率在40MHz左右，流水线CPU工作频率在100MHz左右，TH写入0xffff9e57则CPU时钟频率为定时器频率的25000倍，数码管刷新频率为400Hz或1000Hz，符合人眼观察时对数码管刷新频率的要求。

1. Next:
2. sll $ra, $ra, 1
3. srl $ra, $ra, 1
4. jr $ra
5. Start:
6. jal Next
7. lui $t9, 0x4000
8. sw $zero, 8($t9)
9. lui $t8, 0xffff
10. addi $t8, $zero, 0x9e57 #TH
11. sw $t8, 0($t9)
12. lui $t8, 0xffff
13. addi $t8, $zero, 0xffff #TL
14. sw $t8, 4($t9)
15. sw $t8, 20($t9)
16. addi $t8, $zero, 3
17. sw $t8, 8($t9)
18. 之后等待串口获取操作数，并对操作数进行一次拷贝，在原始储存操作数的寄存器被修改后仍可在数码管显示操作数。
19. Wait:
20. beq $a0, $zero, Wait
21. add $a2, $a0, $zero
22. beq $a1, $zero, Wait
23. add $a3, $a1, $zero

### 5.2.3 中断处理

在完备的中断处理程序中，中断处理需要将所有会被修改的寄存器压栈，在处理结束后再出栈。考虑到本程序在主函数中只涉及到很少的寄存器，在中断处理中未使用主函数中涉及的寄存器，省略了一些操作。

读外设地址，取出其中的标志位判断中断类型，首先检查是否为定时器中断，之后判断是否为串口读中断，最后判断是否为写中断，若均不是则判断为异常。中断处理结束后返回时写外设地址，恢复现场，最后跳转回中断发生处。

1. IRQ:
2. lui $t9, 0x4000
3. lw $s1, 8($t9)
4. andi $t8, $s1, 4
5. bne $t8, $zero, Timer
6. lw $s1, 32($t9)
7. andi $t7, $s1, 8
8. bne $t7, $zero, Read
9. andi $t6, $s1, 4
10. bne $t6, $zero, Write
11. j ill\_handler #不属于以上中断的任何一个，遇到了bug
12. Return:
13. lw $s1, 8($t9)
14. addi $t8, $zero, 2
15. or $s1, $s1, $t8
16. sw $s1, 8($t9)
17. lw $s1, 32($t9)
18. addi $t8, $zero, 3
19. or $s1, $s1, $t8
20. sw $s1, 32($t9)
21. addi $t3, $k0, -4
22. jr $t3

对于计时器中断，在中断中七段数码管被刷新。首先关闭定时器，TCON写入0。之后读取七段数码管当前信息，判断待刷新数码管编号。将待刷新数码管编号信息存入$t1寄存器，将待刷新内容存入$t6寄存器。之后跳转至软件译码部分。

1. Timer:
2. andi $s1, $s1, 0xfff9
3. sw $s1, 8($t9)
4. lw $s2, 20($t9)
5. andi $t7, $s2, 0x0f00
6. addi $t2, $zero, 0x0700
7. addi $t1, $zero, 0x0b00
8. beq $t7, $t2, Second
9. addi $t2, $zero, 0x0b00
10. addi $t1, $zero, 0x0d00
11. beq $t7, $t2, Third
12. addi $t2, $zero, 0x0d00
13. addi $t1, $zero, 0x0e00
14. beq $t7, $t2, Fourth
15. addi $t1, $zero, 0x0700
16. andi $t6, $a2, 0x00f0
17. srl $t6, $t6, 4
18. j Decode
19. Second:
20. andi $t6, $a2, 0x000f
21. j Decode
22. Third:
23. andi $t6, $a3, 0x00f0
24. srl $t6, $t6, 4
25. j Decode
26. Fourth:
27. andi $t6, $a3, 0x000f
28. j Decode

软件译码时逐一比较$t6寄存器中的值和0~15，并将译码结果写入$t4寄存器，若一致则将数码管编号信息和数码管内容写回七段数码管外设地址，之后返回。

1. Decode:
2. addi $t5, $zero, 0
3. addi $t4, $zero, 0xc0
4. beq $t5, $t6, Digi
5. addi $t5, $zero, 1
6. addi $t4, $zero, 0xf9
7. beq $t5, $t6, Digi
8. ……
9. addi $t5, $zero, 14
10. addi $t4, $zero, 0x86
11. beq $t5, $t6, Digi
12. addi $t4, $zero, 0x8e
13. Digi:
14. addi $s2, $t4, $t1
15. sw $s2, 20($t9)
16. j Return

对于串口读中断，首先将UART\_CON写入0，判断此时读到的是第几个操作数，读入操作数，之后返回。

1. Read:
2. andi $s1, $s1, 0xfff0
3. sw $s1, 32($t9)
4. beq $a0, $zero, Firstpara
5. beq $a1, $zero, Secondpara
6. j Return
7. Secondpara:
8. lw $a1, 28($t9)
9. j Return
10. Firstpara:
11. lw $a0, 28($t9)
12. j Return

对于串口写中断，将结果寄存器清零后返回。

1. Write:
2. addi $v0, $zero, 0
3. j Return

### 5.2.4 异常处理

进入死循环。

1. ill\_handler:
2. j ill\_handler

# 六、外设（李登昊）

## 6.1 设计目标

设计单周期和流水线CPU通用的外设，通过汇编程序软件控制，用中断方式访问。

## 6.2 设计方案

### 6.2.1 定时器及开发板上外设资源

以网络学堂上提供的Peripheral.v为基础，将always(\*)语句改为assign语句，以组合逻辑读外设，在时序逻辑中写外设：

1. assign rdata = (rd&(addr==32'h40000000))?TH:
2. (rd&(addr==32'h40000004))?TL:
3. (rd&(addr==32'h40000008))?{29'b0,TCON}:
4. (rd&(addr==32'h4000000C))?{24'b0,led}:
5. (rd&(addr==32'h40000010))?{24'b0,switch}:
6. (rd&(addr==32'h40000014))?{20'b0,digi}:
7. 32'b0;

同时在复位部分中添加将LED和七段数码管复位的语句。

1. led <= 0;
2. digi <= 0;

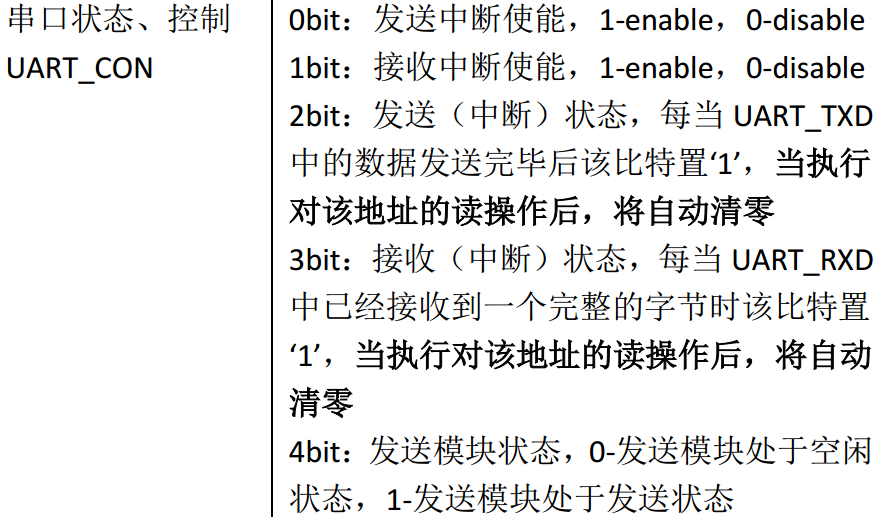
### 6.2.2 UART

UART采用在春季学期第四次实验中使用的接收模块Receiver，发送模块Sender，波特率发生模块Generator（具体代码见附件2015011093\_ex4.pdf）。而作为CPU的外设，UART控制部分不使用类似第四次实验的硬件实现，而使用受CPU的指令控制的封装：

输入包括晶振时钟sysclk，CPU时钟clk，复位信号reset，串口接收信号uart\_rx，读信号rd，写信号wr，写数据wdata，读写地址addr。

输出包括串口发送信号uart\_tx，串口收中断rx\_irq，串口发中断tx\_irq，读数据rdata。

对UART的控制重点在于正确处理uart\_con。



uart\_con[4]通过组合逻辑，对Sender模块的输出tx\_status取反。

1. uart\_con[4] <= ~tx\_status;

uart\_con[3]与uart\_con[2]置’1’的时机是捕捉rx\_status和tx\_status的上升沿，但由于Sender模块和Receiver模块的时钟是波特率时钟，其频率远低于CPU时钟，需要借助两个标志位con3和con2，以CPU时钟的频率对信号进行采样从而正确捕捉上升沿。当CPU访问过uart\_con地址后，uart\_con[3]与uart\_con[2]将自动清零。

1. if (rd & (addr == 32'h40000020)) begin
2. if (uart\_con[1])
3. uart\_con[3] <= 0;
4. if (uart\_con[0])
5. uart\_con[2] <= 0;//使能读时才清零中断
6. end
7. if (tx\_status & ~con2) begin
8. con2 <= 1;
9. uart\_con[2] <= 1;
10. end
11. if (con2 & ~tx\_status) begin
12. con2 <= 0;
13. end
14. if (rx\_status & ~con3) begin
15. con3 <= 1;
16. uart\_con[3] <= 1;
17. rx\_tmp <= rx\_data;
18. end
19. if (con3 & ~rx\_status) begin
20. con3 <= 0;
21. end
22. if(tx\_status & tx\_ready) begin
23. tx\_ready <= 0;
24. data <= tx\_tmp;
25. end

uart\_con[1]与uart\_con[0]的变化由CPU写入控制。

1. if(wr) begin
2. case(addr)
3. 32'h40000018: begin tx\_tmp <= wdata[7:0]; tx\_ready <= 1; end
4. 32'h40000020: uart\_con[1:0] <= wdata[1:0];//中断状态不给写
5. default: ;
6. endcase
7. end

串口收发中断rx\_irq，tx\_irq的由uart\_con的相应位控制，（中断）状态和中断使能均为 ’1’ 时触发中断。

1. assign tx\_irq = uart\_con[0]&uart\_con[2];
2. assign rx\_irq = uart\_con[1]&uart\_con[3];

# 七、参考文献

1. David A. Patterson, John L. Hennessy. Computer Organization and Design, The Hardware/Software Interface, Third Edition[M]. Morgan Kaufmann, 2004.

2. Sarah Harris, David Harris. Digital Design and Computer Architecture: ARM Edition[M]. Morgan Kaufmann, 2015.

3. John L. Hennessy, David A. Patterson. Computer Architecture: A Quantitative Approach[M]. Morgan Kaufmann, 2015.

4. Vivado Design Suite User Guide: Synthesis UG901 (v2017.1)[EB]. April 5, 2017

5. Onur Mutlu. Precise Exception, 15-740/18-740 Computer Architecture[EB].

<http://www.ece.cmu.edu/~ece740/f11/lib/exe/fetch.php?media=wiki:lectures:onur-740-fall11-lecture8-precise-exceptions-afterlecture.pdf>

6. James F. Frenzel. MIPS Instruction Reference[EB].

<http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>