MIPS_V3.1——64位流水线处理器

MIPS_V3.1——64位流水线处理器
- 一、项目概述
- 二、项目文件
- 三、系统结构
- 四、支持指令
- 五、数据通路设计
- 六、控制单元设计
- 七、冒险处理单元设计
- 八、冒险解决策略研究
- 九、测试与演示设计
- 十、串口演示设计
- 十一、时钟与资源
- 十二、参考文献

一、项目概述

本次实验我实现了64位MIPS流水线CPU，支持三种冒险的完整解决方法，支持包括5种64位指令在内的24条指令。但目前的指令都是可以在FDEMW五个周期内完成的指令，在下一版本的CPU种，我会实现支持不同时钟周期结束的指令。

因为我在5月28日第一个给老师检查了流水线，老师让助教给我配置串口调试的代码，但忙了两节课没有成功。我回去后花了很多时间研究并实现了硬件上串口调试的功能（助教给的代码不知道为什么运行不了），终于在第二周实现了通过串口在电脑上显示数据情况的功能。

我还花很多时间调研了不同解决冒险方法对CPI的影响，分析出重定向是能节约最多时间的方法。这一部分的分析，详见<七、冒险解决策略研究>

二、项目文件

根目录（/）

/bit/	存放可以加载到Nexys4实验板上的各种二进制文件
/test/	存放各种版本的汇编文件(.s)、十六进制文件(.dat)
/images/	存放实验报告所需的图片(.png)
/source/	源代码(.sv)
/Reference/	MIPS64官方文档等参考资料
.gitignore	git配置文件
memfile.dat	当前使用的十六进制文件（每行两条指令）
guide.txt	Nexys4实验板演示说明
report.md	实验报告
Nexys4DDR_Master.xdc	Nexys4实验板引脚锁定文件
simulation_behav.wcfg	仿真波形图配置文件

源代码（/source/）

因为整个程序变得更加复杂了，我重构了源代码目录的结构。现在我在source文件夹的根目录下保留了较顶层的几个模块代码，而把其他文件放到了不同功能模块的文件夹中

```
onboard.sv              在Nexys4实验板上测试的顶层模块（含串口调试）
monboard.sv             在Nexys4实验板上测试的顶层模块（不含串口调试）
simulation.sv           仿真时使用的顶层模块
top.sv                  包含mips和内存的顶层模块
mips.sv                 mips处理器的顶层模块
mem.sv                  指令和数据的混合存储器（模拟真实内存情况）
```

可复用模块（/source/utils/）

```
adder.sv                加法器单元
clkdiv.sv               时钟分频模块模块，用于演示
flopenr.sv              时钟控制的可复位的触发寄存器
flopr.sv                可复位的触发寄存器
flopencr.sv             时钟控制的可复位、可清零的触发寄存器
flopcr.sv               可复位、可清零的触发寄存器
mux2.sv                 2:1复用器
mux3.sv                 3:1复用器
mux4.sv                 4:1复用器
mux5.sv                 5:1复用器
sl2.sv                  左移2位
signext.sv              符号拓展模块
zeroext.sv              零拓展模块
```

MIPS控制单元（/source/controller/）

```
controller.sv           mips的控制单元，包含maindec和aludec两部分
aludec.sv               ALU控制单元，用于输出alucontrol信号
maindec.sv              主控单元
```

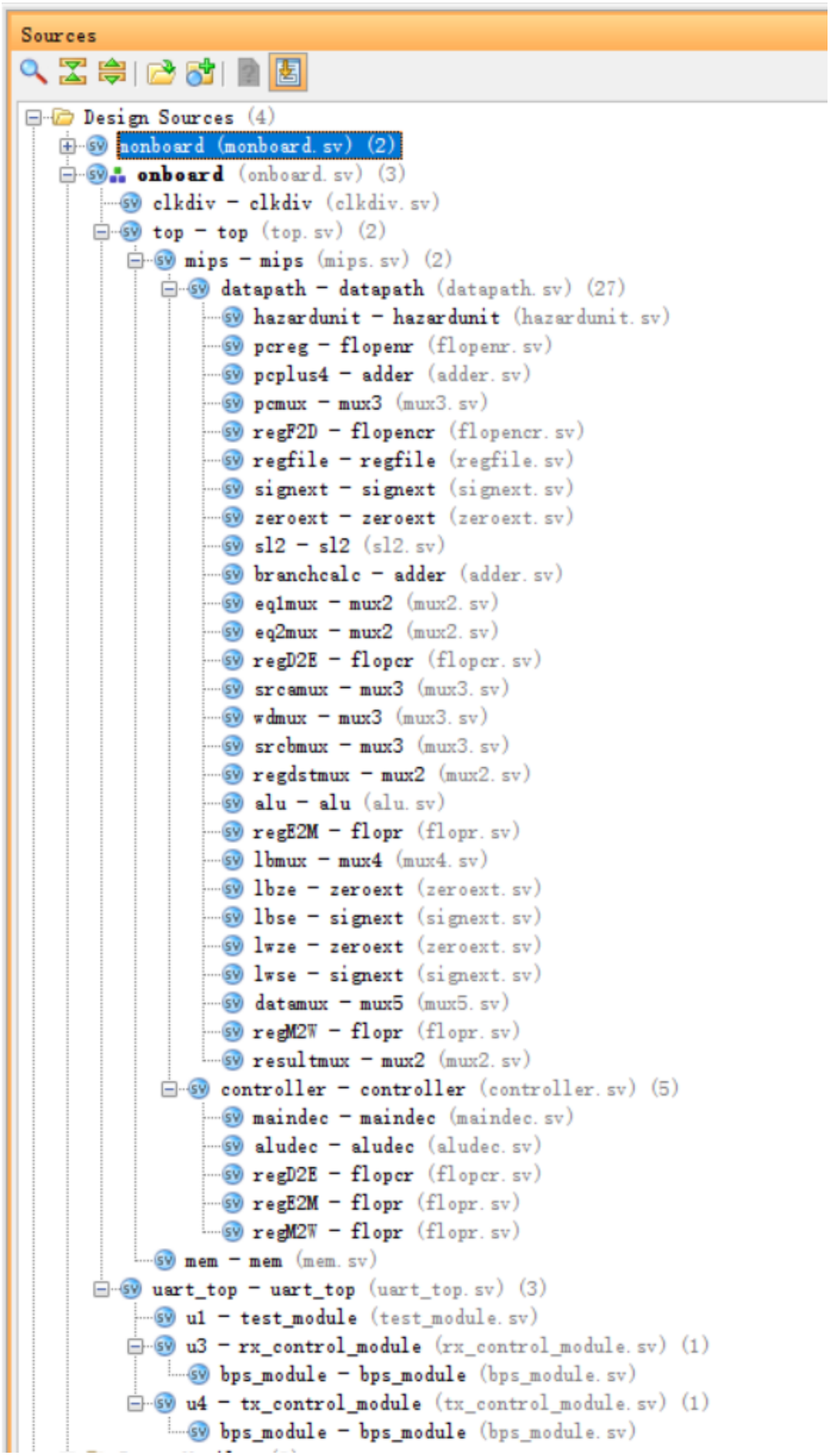
MIPS数据通路（/source/datapath/）

```
datapath.sv             数据通路，mips的核心结构
alu.sv                  ALU计算单元
hazardunit.sv           冒险处理单元
regfile.sv              寄存器文件
```

串口调试模块（/source/uart/）

```
simu_uart.sv            用于仿真测试串口调试功能
uart_top.sv             串口调试顶层模块
bps_module.sv           控制发射速率的模块
rx_control_module.sv    rx控制模块
tx_control_module.sv    tx控制模块
test_module.sv          下降沿检测模块
```

三、系统结构



整个系统结构如上图所示。由于文件过多，下方的simulation.sv和simu_uart.sv 没办法显示出来。从图上，我们可以看到最顶层是monboard和onboard，其中monboard只比onboard少了串口模块。

onboard下有三个模块：clkdiv用于进行时钟分频，给数字串口收发、七段数字灯显示和MIPS提供不同的时钟；top是核心代码的顶层模块，用于模拟一台电脑；uart_top用于处理串口收发的功能。

top模块下是处理器mips和内存mem。mips中又分为数据通路datapath和控制单元controller，这两个部分的具体介绍在<五、数据通路设计>和<六、控制单元设计>中。

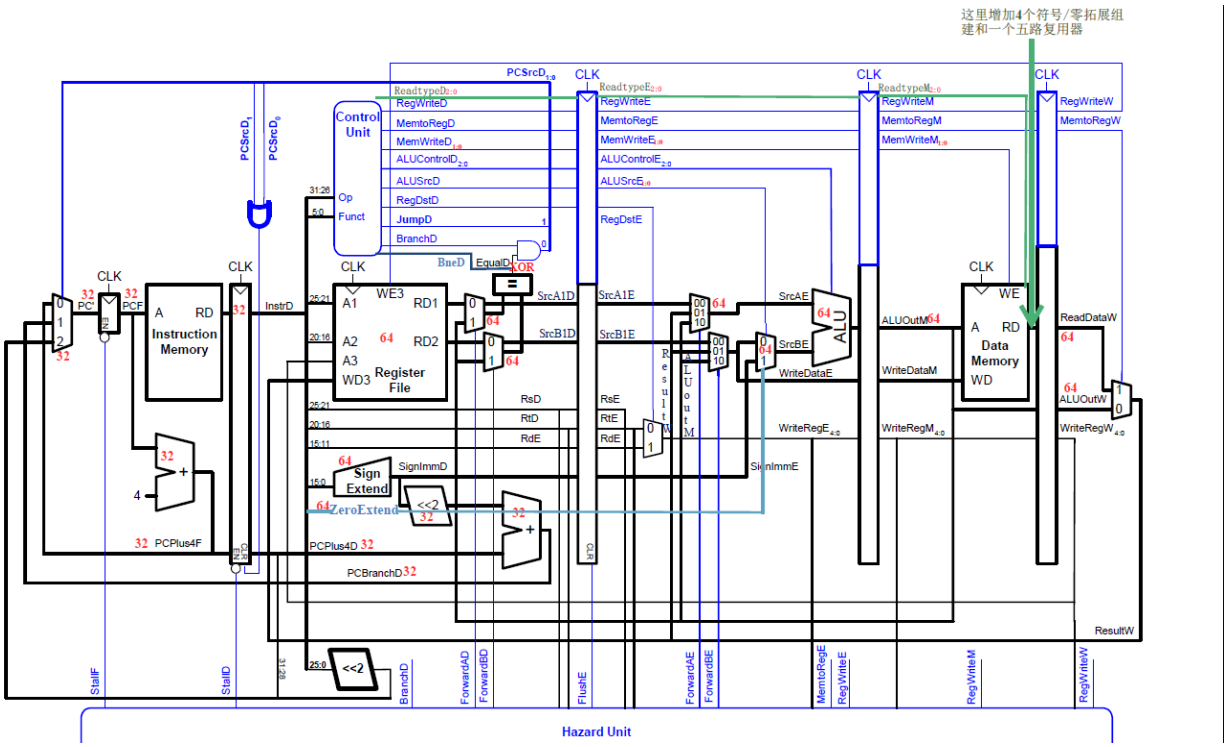
四、支持指令

本次实验和多周期64位MIPS相比没有增加新的指令，实现的全部指令如下：

1. LD，LWU，LW，LBU，LB，SD，SW，SB 共8种读写指令
2. ADD，SUB，OR，AND，SLT，DADD，DSUB，NOP* 共8种R类指令
3. ADDI，ANDI，ORI，SLTI，DADDI 共5种I类计算指令
4. BEQ，BNE，J 共3种分支、跳转指令

*：根据官方文档<MIPS64-Vol2>，NOP 指令实际上是SLL r0, r0, 0，所以也属于R类指令

五、数据通路设计



我的数据通路设计如上图所示，更清晰的图可以见/Reference/流水线图.pdf。

流水线MIPS的核心就是把每条指令的执行过程切分成Fetch、Decode、Execute、Memory和WriteBack五个阶段。当五个阶段的硬件独立时，我们就可以同时执行五条指令的不同阶段，从而提高整个处理器的运行效率。

一条指令运行过程中的数据，随着一个个阶段的执行，从（上图的）左到右进行传递，他们的控制信号也需要跟随着传递。所以，在每两个阶段的中间我们使用时钟触发寄存器来传递各个数据和各个控制信号。我自己拟定的命名方式如下：

数据变量名=数据名+阶段首字母。例如：instrF、instrD、resultW 等。

触发寄存器名=”reg“ +前面的阶段+”2“+后面的阶段。例如regF2D、regD2E等。

下面我分析各个阶段的器件结构，代码中”...“表示省略了部分代码，完整代码见datapath.sv。冒险处理单元的介绍见<七、冒险处理单元设计>。

Fetch阶段

Fetch阶段主要是用触发寄存器确定读取指令的时机，同时用三路复用器选择beq(bne)、J和顺序执行三种PC可能性。需要向Decode阶段传递读取的指令和加四后的pc值。可以看到指令相关的器件都是一字（W）长的32位。

```
flopenr #(W)    pcreg(clk, reset, ~StallF, pcnextF, pcF);
adder   #(W)    pcplus4(pcF,32'b100,pc4F);
mux3    #(W)    pcmux(pc4F,pcbranchD,{pc4D[31:28],instrD[25:0],2'b00},pcsrcD,pcnextF);
flopencr#(64)   regF2D(clk,reset,~StallD,FlushD,...);
```

Decode阶段

Decode阶段主要完成译码、寄存器的读取、分支类pc值的计算和I类指令立即数的拓展。同时还需要提前进行是否分支的计算，在这里运用了寄存器的重定向。传到下一阶段的主要是R类指令操作器的值、寄存器读取的值和拓展后的立即数，其中零拓展用于ANDI和ORI指令。同时，我们可以看到寄存器的值和拓展后端立即数都是64位的。

```
//译码
assign  op    = instrD[31:26];
assign  funct = instrD[5:0];
assign  rsD   = instrD[25:21];
assign  rtD   = instrD[20:16];
assign  rdD   = instrD[15:11];
//寄存器读取
regfile#(N,32)  regfile(clk,regwriteW,rsD,rtD,writeregW,resultW,srca1D,srcb1D,checka,check);
//I类指令立即数拓展,也同时用于跳转类pc值的计算
signext#(I,N)   signext(instrD[15:0],signimmD);
zeroext#(I,N)   zeroext(instrD[15:0],zeroimmD);
//分支类pc值的计算
sl2     #(W)    sl2(signimmD[W-1:0],signimm4D);
adder   #(W)    branchcalc(pcF,signimm4D,pcbranchD);
//是否分支的计算
mux2    #(N)    eq1mux(srca1D,aluoutM,ForwardAD,euqalAD);
mux2    #(N)    eq2mux(srcb1D,aluoutM,ForwardBD,euqalBD);
assign equalD = (euqalAD==euqalBD)^bneD;
assign pcsrcD = {jumpD,branchD & equalD};
assign FlushD = pcsrcD[0] | pcsrcD[1];
//
flopcr#(271)    regD2E(clk,reset,FlushE,...);
```

Execute阶段

Execute阶段主要完成alu计算单元的计算，这里使用了重定向的技术来获得两个正确的操作数值。向下阶段传递的是alu计算的结果和重定向后可能写入内存的值。

```
mux3    #(N)    srcamux(srca1E,resultW,aluoutM,ForwardAE,srcaE);
mux3    #(N)    wdmux(srcb1E,resultW,aluoutM,ForwardBE,writedataE);
mux3    #(N)    srcbmux(writedataE,signimmE,zeroimmE,alusrcE,srcbE);
mux2    #(5)    regdstmux(rtE, rdE, regdstE, writeregE);
alu     #(N)    alu(srcaE,srcbE,alucontrolE,aluoutE,zero);
flopr #(133)    regE2M(clk,reset,...});
```

Memory阶段

Memory阶段包括内存的读写，写内存时只需要根据传递过来的信号和数据操作即可。而读内存时，为了支持LD，LWU，LW，LBU，LB这五个不同的指令，我设计了若干选择器和符号拓展/零拓展单元。

```
//写内存
assign dataadr   =  aluoutM;
assign writedata =  writedataM;
//读内存
mux4 #(B)       lbmux(readdata[31:24], readdata[23:16], readdata[15:8],
                    readdata[7:0], dataadr[1:0], mbyte);
zeroext #(B,N)  lbze(mbyte, mbytezext);
signext #(B,N)  lbse(mbyte, mbytesext);
zeroext #(W,N)  lwze(readdata[31:0], mwordzext);
signext #(W,N)  lwse(readdata[31:0], mwordsext);
mux5    #(N)    datamux(mwordsext,mwordzext,mbytesext,mbytezext,readdata,readtypeM,readdataM);
//
flopr #(133)    regM2W(clk,reset,...);
```

Write Back阶段

Write Back阶段只需要用复用器选择一下要写回的数据即可。

```
mux2    #(N)    resultmux(aluoutW,readdataW,memtoregW,resultW);
```

六、控制单元设计

控制单元和数据路径一样需要进行不同阶段上的传递，maindec和aludec都在Decode阶段生成对应的控制信号，由三个触发寄存器将各个信号进行传递。

```
maindec maindec(clk, reset, op,
                    regwriteD,memtoregD, regdstD, memwriteD,
                    alusrcD, bneD, branchD, jumpD,
                    aluopD, readtypeD);
aludec aludec(funct, aluopD, alucontrolD);
flopcr#(14)     regD2E(clk,reset,FlushE,...);
flopr #(7)      regE2M(clk,reset,...);
flopr #(2)      regM2W(clk,reset,...);
```

最终我们需要输出的只是其中的一部分信号

```
output  logic       regdstE, regwriteE,regwriteM,regwriteW,
output  logic       memtoregE,memtoregM,memtoregW,
output  logic [1:0] memwriteM,
output  logic [1:0] alusrcE,
output  logic [3:0] alucontrolE,
output  logic       bneD,branchD,jumpD,
output  logic [2:0] readtypeM
```

具体控制信号方面，aludec的设计和上次实验相同，maindec中controls只需要16位即可。主要包含内存和寄存器读写控制、alu操作控制和分支跳转控制。

```
assign {regwrite,memtoreg,regdst, memwrite, alusrc,
            bne,branch,jump, aluop, readtype} = controls;
    always_comb
        case (op)
            RTYPE:  controls <= 16'b101_00_00_000_111_000;
            SD:     controls <= 16'b000_11_01_000_000_000;
            SW:     controls <= 16'b000_01_01_000_000_000;
            SB:     controls <= 16'b000_10_01_000_000_000;
            LD:     controls <= 16'b110_00_01_000_000_100;
            LWU:    controls <= 16'b110_00_01_000_000_001;
            LW:     controls <= 16'b110_00_01_000_000_000;
            LBU:    controls <= 16'b110_00_01_000_000_011;
            LB:     controls <= 16'b110_00_01_000_000_010;
            ADDI:   controls <= 16'b100_00_01_000_000_000;
            ANDI:   controls <= 16'b100_00_10_000_001_000;
            ORI:    controls <= 16'b100_00_10_000_010_000;
            SLTI:   controls <= 16'b100_00_01_000_011_000;
            DADDI:  controls <= 16'b100_00_01_000_100_000;
            BEQ:    controls <= 16'b000_00_00_010_000_000;
            BNE:    controls <= 16'b000_00_00_110_000_000;
            J:      controls <= 16'b000_00_00_001_000_000;
         endcase
```

七、冒险处理单元设计

冒险处理单元的核心代码在书上基本上都给出了。其中重定向包含了两个部分：一个是ALU运算数选择上的重定向、一个是分支预测读取寄存器值的重定向。而刷新流水线主要用于解决lw后读冒险和分支预测冒险。处理单元生成的StallF,StallD,FlushE信号控制了流水线是否"流动"，而ForwardAD,ForwardBD,ForwardAE,ForwardBE信号则为是否重定向寄存器以及重定向到哪个寄存器提供了控制。

冒险处理单元的完整代码如下：

```
module hazardunit(
    input   logic       clk,reset,
    input   logic       branchD,
    input   logic [4:0] rsD,rtD,rsE,rtE,
    input   logic [4:0] writeregE,writeregM,writeregW,
    input   logic       memtoregE,memtoregM,regwriteE,
                        regwriteM,regwriteW,
    output  logic       StallF,StallD,FlushE,
    output  logic       ForwardAD,ForwardBD,
    output  logic [1:0] ForwardAE,ForwardBE
);
    logic   lwStallD, branchStallD;
    // 重定向
    assign ForwardAD = rsD !=0 & (rsD == writeregM) & regwriteM;
    assign ForwardBD = rtD !=0 & (rtD == writeregM) & regwriteM;
    always_comb begin
```

```
        ForwardAE = 2'b00; ForwardBE = 2'b00;
        if (rsE != 0)
            if (rsE == writeregM & regwriteM)
                ForwardAE = 2'b10;
            else if (rsE == writeregW & regwriteW)
                ForwardAE = 2'b01;
        if (rtE != 0)
            if (rtE == writeregM & regwriteM)
                ForwardBE = 2'b10;
            else if (rtE == writeregW & regwriteW)
                ForwardBE = 2'b01;
    end
    // 刷新流水线
    assign #1 lwStallD = memtoregE & (rtE == rsD | rtE == rtD);
    assign #1 branchStallD = branchD & (regwriteE &
        (writeregE == rsD | writeregE == rtD) |
        memtoregM & (writeregM == rsD | writeregM == rtD));
    assign #1 StallD = lwStallD | branchStallD;
    assign #1 StallF = StallD;
    assign #1 FlushE = StallD;
endmodule
```

八、冒险解决策略研究

如果不使用冒险处理单元，我们必须在代码中插入足够多的NOP来使程序可以在流水线上运行（插入NOP的功能实际上应该由编译器实现）。我通过分析使用不同冒险处理策略时所需要插入的NOP数和CPI，来考察不同冒险处理策略的效果。

测试方法：通过修改冒险处理单元中如下两行的代码，达到开关冒险处理策略的效果。

```
assign #1 lwStallD = memtoregE & (rtE == rsD | rtE == rtD);
//assign #1 lwStallD = 0;
assign #1 StallD = lwStallD | branchStallD;
//assign #1 StallD = lwStallD;
```

测试结果：

0.程序简介

程序	介绍
standard2.s	标准测试程序，测试所有设计中的32位指令
testls.s	64位数字的读写程序
power2.s	计算2的次幂，并存到内存中，直到2的56次幂
testmore.s	充满各种分支判断、真数据依赖的测试程序

1.只插入nop来解决冒险

程序	指令数	数据NOP*	跳转NOP	时钟周期	CPI
standard2.s	17	28	4	53	3.12
testls.s	12	10	0	26	2.16
power2.s	341	336	112	793	2.33
testmore.s	1092	1188	396	2680	2.45

*：数据NOP包括lw后NOP和数据依赖

2.使用重定向后的情况

程序	指令数	lw后NOP	跳转NOP	时钟周期	CPI
standard2.s	17	0	2	23	1.35
testls.s	12	1	0	17	1.42
power2.s	341	0	56	411	1.21
testmore.s	1092	99	198	1393	1.28

3.使用三种解决冒险的方法

程序	指令数	时钟周期	CPI
standard2.s	17	23	1.35
testls.s	12	17	1.42
power2.s	341	411	1.21
testmore.s	1092	1393	1.28

结论:

寄存器的重定向可以化解大部分的冒险，使CPI明显的下降，但无法解决lw后读冒险和分支冒险；而使用刷新流水线的方法来解决分支冒险和lw后读冒险，可以节省下编译器插入NOP的时间，但无法加快处理器运算速度、降低CPI。

九、测试与演示设计

仿真测试

仿真测试与上次实验基本一致，但原来的测试文件代码为了适应<八、冒险解决策略研究>有所改动。如下图所示，根据我设计好的逻辑，特定程序正确执行后会停在特定的位置。波形图在项目的前期用于Debug，在这份报告中就不再添加了。

```
always @(negedge clk) begin
    if (memwrite) begin
        $display("Write %d in %d",writedata,dataadr);
        if (dataadr == 100 & writedata == 7)begin
            $display("Test-standard2 pass!");
            #100 $stop;
        end
        if (dataadr == 508 & writedata == 7)begin
             $display("Test-power2 pass!");
             #100 $stop;
        end
        if (dataadr == 80 & writedata == 1)begin
             $display("Test-loadstore pass!");
             #100 $stop;
        end
        if (dataadr == 320 & writedata == 4950)begin
            $display("Test-more ends!");
            $display(cnt);
            #100 $stop;
        end
    end
    cnt = cnt + 1;
    if(cnt == 1580)begin
        $display("Some error occurs!");
        $stop;
    end
```

实验板演示

这次实验中，我设计一套更符合64位系统的测试演示方案：

8位7段数码管，每位一个十六进制数，共表示一个32位数字。具体显示的内容由开关和当前处理器状态决定。

实验板大体上有数据检查模式和默认模式两种：数据检查模式中利用SW6-5来控制要检查的内容，用SW4来选择检查数据的高位和低位。而默认模式中，7段数码管的八位显示分别为—— 7-6：当前Fetch阶段的PC值；5：写寄存器的位置；4:写内存的类型；3-2：SW8-15表示的数；1-0：对应寄存器的低8位。

具体的开关功能如下表：

开关	上（1）	下（0）
SW0	RESET	
SW1	暂停	开启
SW2	四倍速	常速
SW3	默认模式	检查模式
SW4	64位的高位	64位的低位
SW6-5	00 寄存器的值	01 内存的值
SW6-5	10 写数据的地址	11 写数据的值
SW7	显示CLK	显示写寄存器
SW15-SW8	控制地址	

5月28日演示流程：

1. 先SW3上+SW5上开始查看内存
2. SW1上，开始运行
3. 运行若干步暂停，查看寄存器
4. 可以等待寄存器变化
5. 当写内存时暂停
   5.1 SW6上 查看写内存的地址
   5.2 SW5上 查看写内存的内存
   5.3 SW4上 查看高位的值
6. SW2上四倍速快速运行

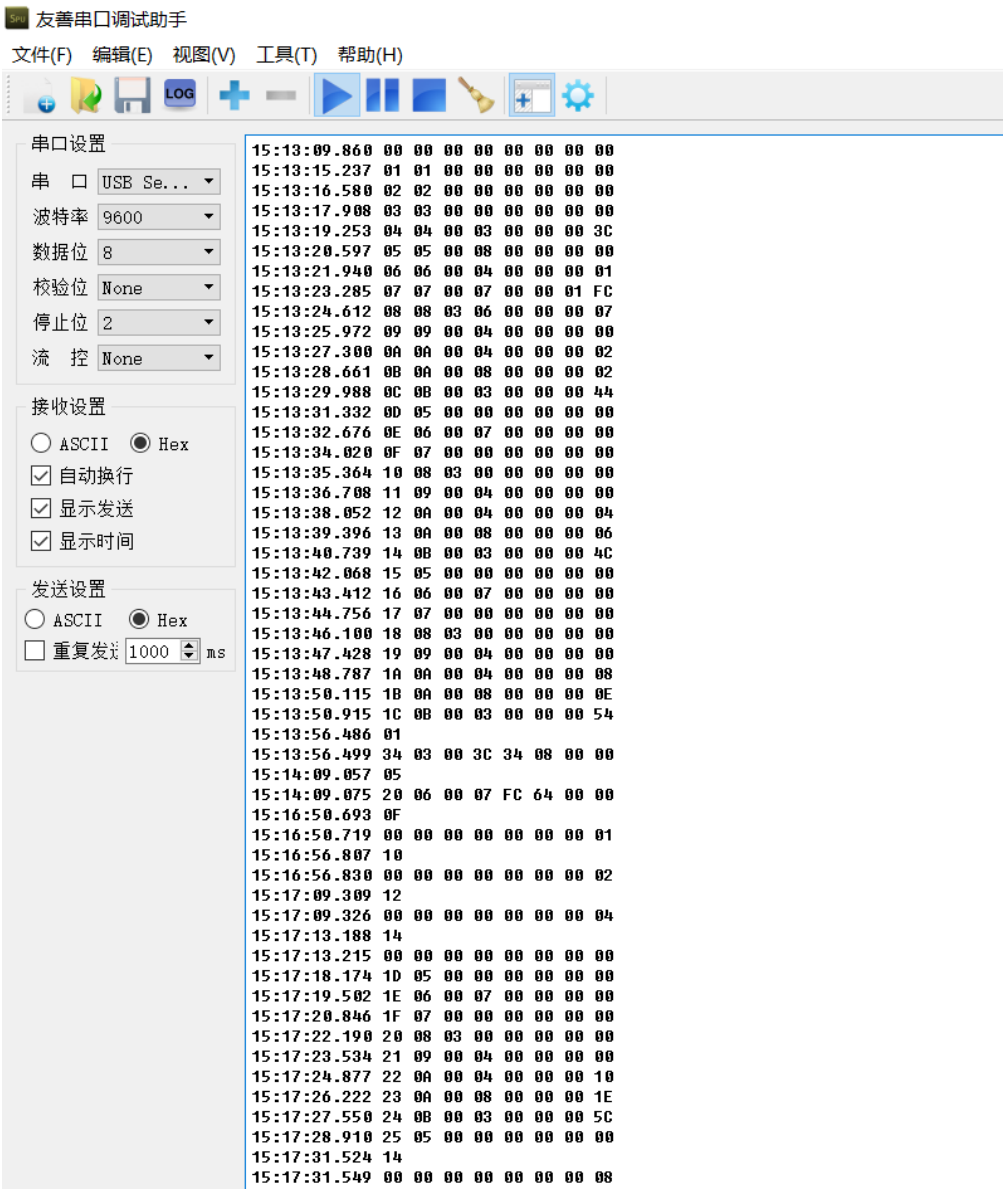
十、串口演示设计

串口即是Nexys4实验板与计算机相连的数据端口，我们生成的二进制文件通过串口传到实验板上。同样的，在实验板实时运行的时候，我们也可以通过串口实时地与计算机通讯。

我参照网上的一些资料，实现了串口接发功能的System Verilog代码(/source/uart)，最终实现如下演示功能：

1. 每个MIPS的时钟周期，实验板向计算机发送：时钟周期、Fetch阶段的PC、是否读内存及读内存的种类、被修改的寄存器、以及寄存器修改的值的低32位
2. 当计算机输入一个2位十六进制数（0-255）A时，实验板向计算机发送内存上第A个字（word）开始的两个字的值（即可以查看64位数）**[一般先用开关暂停MIPS]**

下图是power2.s运行时串口调试助手的截图，可以很清楚地看到循环的进行和寄存器的变化。后面输入1byte的地址也可以获得正确的数据。



串口可以同时进行发送1bit和接受1bit的操作，我们需要自己设计一套特定的编码方案使串口发送的0/1序列可以被解码。以发送数据为例，当我们设定好的tx_sig信号为真时，先发出一个1和一个0表示数据的开始，然后发送8bit的数据，然后再发送2个1表示结束。由于一次只能发送1byte的数据64位的数据则需要8次才能发送完毕。

码元发送的频率被称为波特率，结束后发送2个1则表示有两个停止位，在部分编码方案中还会有校验位的存在。如果没有进行正确的串口设置，同样的数据01可能会得到不同的显示。

顺便总结一下整个系统中各个时钟的频率

时钟	频率	用处/备注
CLK100MHZ	100MHZ	Nexys4自带时钟
CLK380	381HZ	用于刷新7段译码器
CLK1_6	1.49HZ	快MIPS时钟周期
CLK0_4	0.37HZ	慢MIPS时钟周期
clk_trx	50MHZ	串口模块的检测时钟
bps_clk	9600HZ	5207 个clk_trx
波特率	9600	50M = 9600 * 5207
发送1Byte数据	800	12个bps_clk
最快发送bit频率	6400	8bit/12bps_clk

十一、时钟与资源

Design Timing Summary

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.827 ns	Worst Hold Slack (WHS):	0.324 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	28	Total Number of Endpoints:	28	Total Number of Endpoints:	29

All user specified timing constraints are met.

Utilization

Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
LUT	2373	63400	3.74
LUTRAM	900	19000	4.74
FF	735	126800	0.58
IO	42	210	20.00
BUFG	6	32	18.75

十二、参考文献

1. MIPS64-Vol1.pdf
2. MIPS64-Vol2.pdf
3. 流水线图.pdf：原图是课本的作者在另一本书上的配图