Lab06实验报告

简单的类 MIPS 多周期流水线处理器设计与实现

518021910489 陈沛宇

一、实验目的

1. 理解CPU Pipeline，了解流水线冒险(hazard)及相关性，设计基础流水线CPU；
2. 设计支持Stall的流水线CPU。通过检测竞争并插入停顿（Stall）机制解决数据冒险、控制竞争和结构冒险；
3. 在 2.的基础上，增加 Forwarding 机制解决数据竞争，减少因数据竞争带来的流水线停顿延时，提高流水线处理器性能；（PS：也允许考虑将 Stall 与 Forwarding 结合起来实现）
4. 在 3. 的基础上，通过predict-not-taken或延时转移策略解决控制冒险/竞争，减少控制竞争带来的流水线停顿延时，进一步提高处理器性能。（PS：也允许考虑将 2.、3.和 4.结合起来设计）

二、实现步骤

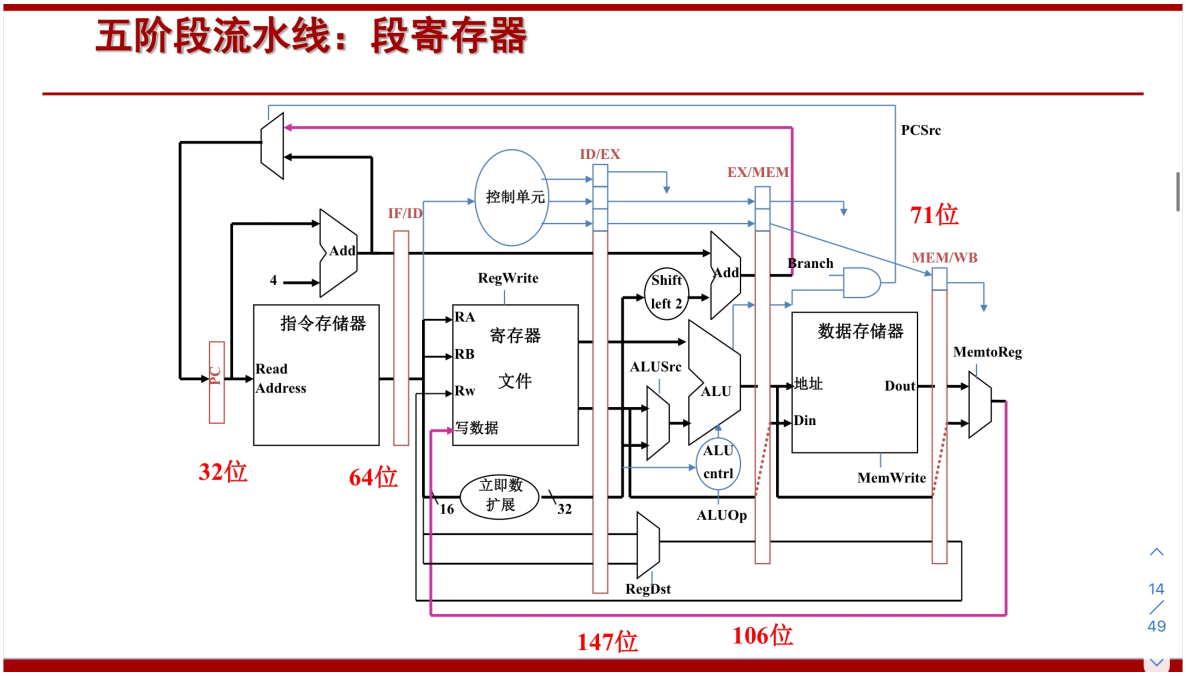
概念理解

1. CPU Pipeline：将一条指令分解为多个步骤，让不同的指令在同一时刻使用不同的原件，使各部操作重叠，是一种提高CPU吞吐量的指令执行方式，提高了指令并行的效率。

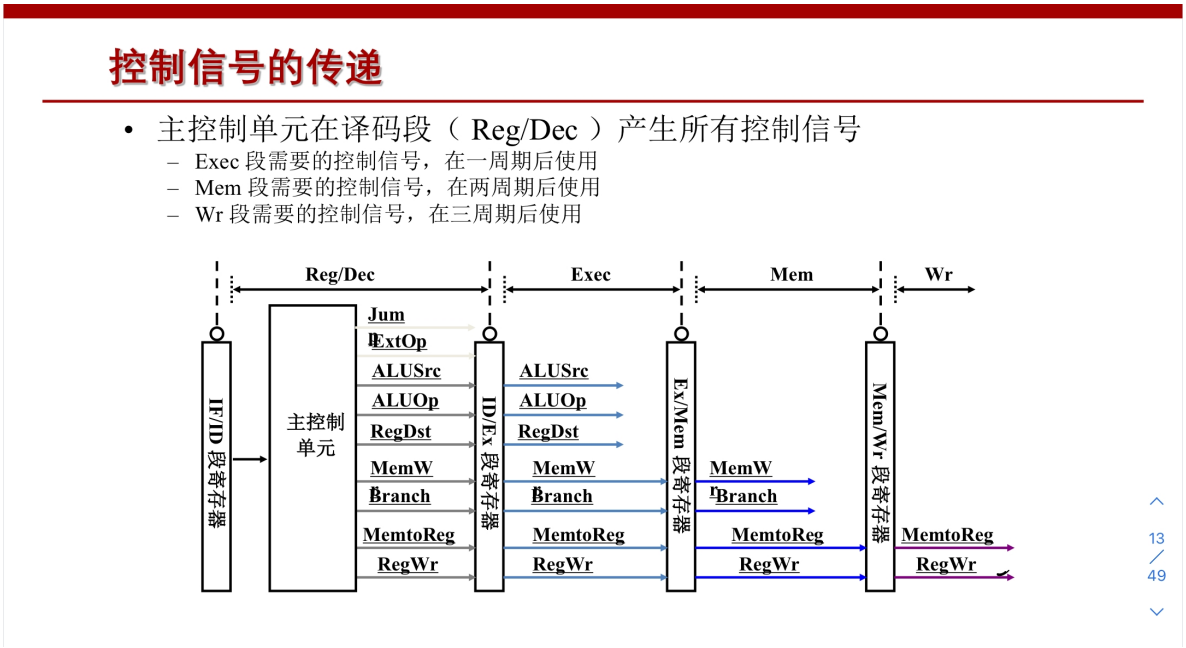
2. 流水线相关性与冒险：相邻或相近的指令之间因存在某种相关性，使得指令的执行可能受到影响；这些相关性，可能会影响指令的执行，也可能不影响，因此又称为冒险。

处理器模块分析

流水线实现的核心是段与段之间数据和信号的传递，故需要设置段寄存器帮助传递，五段流水需要插入四个段寄存器，如下图所示：



其中需要通过段寄存器传递的信号如下图所示，已经被使用过的信号无需传递，传递的只有后续会用于实例化模块的信号：



可以看出五阶段流水的每一个段内的元件无需做太大修改，其余工作只需要加入段寄存器即可实现CPU Pipeline，故本次实验报告着重分析顶层模块的设计，与lab05重合的部分不再赘述。

顶层模块

模块描述

与lab05中顶层模块的功能相同，用于连接各个各段的元件并在各段之间传输数据，从而实现指令级别的并行执行。CPU Pipeline分为五段，分别为**取指（IF）-译码(ID)-执行(EX)-访存(MEM)-写回(WB)**，顶层模块除了连接段与段还需要实现段寄存器，在Pipeline中插入四级寄存器，可以将前一阶段的相应数据保存给下一阶段使用。

IF/ID段寄存器

该段寄存器的作用是传递PC+4的值、当前指令和后续会使用到寄存器的编号等，实现代码如下：

```
//IF|ID
    reg [31:0] IFID_PC_PLUS_4,IFID_INST;
    wire [4:0] IFID_RS=IFID_INST[25:21];
    wire [4:0] IFID_RT=IFID_INST[20:16];
    wire [4:0] IFID_RD=IFID_INST[15:11];
    wire BRANCH,ZERO,JUMP;
```

ID/EX段寄存器

该段寄存器的作用是传递在该阶段通过主控制单元产生的控制信号、从寄存器文件中读取到的数据、立即数扩展得到的数据和上一阶段传递下来还未被使用的数据，实现代码如下：

```
//ID|EX
    reg [31:0] IDEX_READ_DATA_1,IDEX_READ_DATA_2,IDEX_IMM;
    reg [4:0] IDEX_RS,IDEX_RT, IDEX_RD;
    reg [10:0] IDEX_CTRL;
    wire
IDEX_ZERO_EXT=IDEX_CTRL[10],IDEX_JUMP=IDEX_CTRL[9],IDEX_REGDST=IDEX_CTRL[8],
         IDEX_ALUSRC=IDEX_CTRL[5],IDEX_BRANCH = IDEX_CTRL[4],IDEX_MEMREAD =
IDEX_CTRL[3],
         IDEX_MEMWRITE = IDEX_CTRL[2], IDEX_REGWRITE =
IDEX_CTRL[1],IDEX_MEMTOREG = IDEX_CTRL[0];
    wire [1:0] IDEX_ALUOP=IDEX_CTRL[7:6];
```

EX/MEM段寄存器

该段寄存器的作用是传递ALU运算出的结果、目标寄存器的编号和未被使用的控制信号，实现代码如下：

```
//EX|MEM
    reg [31:0] EXMEM_ALURES, EXMEM_WRITE_DATA;
    reg [4:0] EXMEM_DSTREG;
    reg [4:0] EXMEM_CTRL;
    reg EXMEM_ZERO;
    wire EXMEM_BRANCH = EXMEM_CTRL[4], EXMEM_MEMREAD = EXMEM_CTRL[3],
          EXMEM_MEMWRITE = EXMEM_CTRL[2], EXMEM_REGWRITE = EXMEM_CTRL[1],
          EXMEM_MEMTOREG = EXMEM_CTRL[0];
```

MEM/WB段寄存器

该段寄存器的作用是传递从内存文件中读取到的数据、上一阶段传递来的ALU运算结果、目标寄存器编号和未被使用的控制信号，实现代码如下：

```
reg [31:0] MEMWB_READ_DATA, MEMWB_ALURES;
    reg [4:0] MEMWB_DSTREG;
    reg [1:0] MEMWB_CTRL;
    wire MEMWB_REGWRITE = MEMWB_CTRL[1], MEMWB_MEMTOREG = MEMWB_CTRL[0];
```

上述4个段寄存器完成了数据的传输，保证了CPU流水运作时各元器件相互独立，从而可以用来实现指令级别的并行。

竞争检测

在无法使用前向通路转发机制解决数据冒险的地方需要暂停流水线（前向通路转发机制会在后面分析），如使用`lw`指令后紧跟着使用`lw`得到的数据。设置一个`STALL`信号，在接收到相应指令产生的信号后将`STALL`信号置1，实现代码如下：

```
// Hazard detection
    wire STALL = IDEX_MEMREAD & (IDEX_RT == IFID_RS | IDEX_RT == IFID_RT);
```

CPU Pipeline实现

取指IF

CPU Pipeline的第一个阶段是取指，需要访问指令寄存器，故初始化`InstMemory`模块得到IF阶段的指令，为了提升处理器性能，获得更好的CPI，在这一阶段也顺带计算下一条PC的地址，并且综合设计了了实验目的4中提到的使用predict-not-taken的方式解决控制冒险/竞争，减少控制竞争带来的流水线停顿延时。因为`BRANCH_RESULT`的选择需要用到在译码阶段才产生的`BRANCH`信号，故第一次一定是选择PC+4作为下一条指令的PC，这样可以将CPI降低到2，同时将本应该在译码阶段产生的一个信号移动到本阶段可以使无条件转移更快地发生，这些设计都在一定程度上提升了处理器性能。

实例化和相关运算结束后需要传递数据给段寄存器，在此解释流水线寄存器的写入策略，后续每一个阶段都采用这样的策略：指令级并行时要严格避免读写冲突，为了避免冲突，需要对存储器和寄存器的读写进行同步。故采用在时钟信号下降沿进行寄存器文件和内存文件的写入，在时钟信号上升沿进行流水线段寄存器的写入，保证对于寄存器文件和内存文件，可以在前半个时钟周期进行写入，后半个时钟周期进行读取，从而避免一部分结构冒险和数据冒险。这个策略也和上课介绍的策略有相似之处。

取指阶段的段寄存器写入传递的数据适于PC和指令相关的数据，并且设置如果接收到`STALL`信号则将指令flush掉。

该部分实现代码如下：

```
reg [31:0] PC;
    wire [31:0] PC_PLUS_4, BRANCH_ADDR,JUMP_ADDR,BRANCH_RESULT,NEXT_PC,
IF_INST,IMM;
    InstMemory InstMemory_init(
                                .address(PC),
                                .instruction(IF_INST)
                            );
    assign PC_PLUS_4 = PC + 4;
    assign JUMP_ADDR ={PC_PLUS_4[31:28],IF_INST[25:0]<<2};

    assign BRANCH_RESULT = (BRANCH & ZERO)?BRANCH_ADDR:PC_PLUS_4;
    wire [10:0] CTRL_OUT;
    ctr jmp_init(
                .opCode(IF_INST[31:26]),
                .jump(CTRL_OUT[9])
                );
    assign JUMP = CTRL_OUT[9];
```

```
    assign NEXT_PC = (JUMP)?JUMP_ADDR:BRANCH_RESULT;

    always @ (posedge Clk)
    begin
        if (!STALL)
        begin
            IFID_PC_PLUS_4 <= PC_PLUS_4;
            IFID_INST <= IF_INST;
            PC <= NEXT_PC;
        end
        if (BRANCH)
            IFID_INST <= 0;
```

译码ID

该阶段完成的工作主要有：

1. 实例化主控制单元生成控制信号；
2. 实例化寄存器文件并根据之前传递下来的寄存器编号进行寄存器文件的读取；
3. 进行两种立即数拓展并根据`zeroext`信号进行选择，与lab05一样；
4. 生成跳转地址；
5. 传递控制信号等数据到段寄存器，数据流入下一段。

```
ctr ctr_init(
                .opCode(IFID_INST[31:26]),
                .regDst(CTRL_OUT[8]),
                .aluOp(CTRL_OUT[7:6]),
                .aluSrc(CTRL_OUT[5]),
                .branch(CTRL_OUT[4]),
                .memRead(CTRL_OUT[3]),
                .memWrite(CTRL_OUT[2]),
                .regWrite(CTRL_OUT[1]),
                .memToReg(CTRL_OUT[0]),
                .zeroext(CTRL_OUT[10])
                );

    wire [31:0] READ_DATA_1, READ_DATA_2, REG_WRITE_DATA;
    Register register_init(
                        .Clk(Clk),
                        .readReg1(IFID_RS),
                        .readReg2(IFID_RT),
                        .writeReg(MEMWB_DSTREG),
                        .writeData(REG_WRITE_DATA),
                        .regWrite(MEMWB_REGWRITE),
                        .reset(reset),
                        .readData1(READ_DATA_1),
                        .readData2(READ_DATA_2)
                        );
    wire [31:0] IMM_SEXT,IMM_0EXT;
    signext signext_init(
                    .inst(IFID_INST[15:0]),
                    .data(IMM_SEXT)
                    );
    zeroext zeroext_init(
                    .inst(IFID_INST[15:0]),
                    .data(IMM_0EXT)
                    );
```

```
    Mux32 mux32_imm_init(
                         .SEL(CTRL_OUT[10]),
                         .INPUT1(IMM_0EXT),
                         .INPUT2(IMM_SEXT),
                         .OUT(IMM)
                         );

    assign BRANCH = (READ_DATA_1 == READ_DATA_2) & CTRL_OUT[4];
    assign BRANCH_ADDR = IFID_PC_PLUS_4+(IMM<<2);
     always @ (posedge Clk)
    begin
        IDEX_CTRL <= STALL ? 0 : CTRL_OUT;
        IDEX_READ_DATA_1 <= READ_DATA_1;
        IDEX_READ_DATA_2 <= READ_DATA_2;
        IDEX_IMM <=IMM;
        IDEX_RS <= IFID_RS;
        IDEX_RT <= IFID_RT;
        IDEX_RD <= IFID_RD;
    end
```

执行EX

在执行阶段首先进行了源操作数的选择，使用到了转发机制，转发机制即：将之前周期的执行阶段或访存阶段得到的数据，直接作为当前执行阶段的操作数之一。

转发机制的数据来源有：执行阶段的运算单元在上一周期运算出的结果，访存阶段在上一周期读取还未写回寄存器文件的内容。

在转发机制的实现中参考了文章https://blog.csdn.net/accelerato/article/details/92847394和《计算机组成与设计》中的内容。

在选择好了源操作数后便可以进行`aluctr`模块和`ALU`模块的实例化了，实例化方法不在此赘述。实例化结束后同样在时钟信号上升沿进行段寄存器的写入。

```
//转发机制
    wire FWD_EX_A = EXMEM_REGWRITE & EXMEM_DSTREG != 0 & EXMEM_DSTREG ==
IDEX_RS;
    wire FWD_EX_B = EXMEM_REGWRITE & EXMEM_DSTREG != 0 & EXMEM_DSTREG ==
IDEX_RT;
    wire FWD_MEM_A = MEMWB_REGWRITE & MEMWB_DSTREG != 0 & !(EXMEM_REGWRITE &
EXMEM_DSTREG != 0 & EXMEM_DSTREG != IDEX_RS) & MEMWB_DSTREG == IDEX_RS;
    wire FWD_MEM_B = MEMWB_REGWRITE & MEMWB_DSTREG != 0 & !(EXMEM_REGWRITE &
EXMEM_DSTREG != 0 & EXMEM_DSTREG != IDEX_RT) & MEMWB_DSTREG == IDEX_RT;
    wire [31:0] ALU_SRC_A = FWD_EX_A ? EXMEM_ALURES : FWD_MEM_A ? REG_WRITE_DATA
: IDEX_READ_DATA_1;
    wire [31:0] ALU_SRC_B = IDEX_ALUSRC ? IDEX_IMM : FWD_EX_B ? EXMEM_ALURES
:FWD_EX_B ? EXMEM_ALURES : FWD_MEM_B ? REG_WRITE_DATA : IDEX_READ_DATA_2;
    wire [31:0] MEM_WRITE_DATA = FWD_EX_B ? EXMEM_ALURES :FWD_EX_B ?
EXMEM_ALURES : FWD_MEM_B ? REG_WRITE_DATA : IDEX_READ_DATA_2;

    wire [3:0] ALU_CTRL_OUT;
    aluctr aluctr_init(
                       .Funct(IDEX_IMM[5:0]),
                       .aluOp(IDEX_ALUOP),
                       .rs(IDEX_RS),
                       .aluCtrOut(ALU_CTRL_OUT)
                       );
    wire [31:0] ALU_RES;
```

```
    ALU alu_init(
                .input1(ALU_SRC_A),
                .input2(ALU_SRC_B),
                .input3(IDEX_IMM[10:6]),
                .aluCtr(ALU_CTRL_OUT),
                .zero(ZERO),
                .aluRes(ALU_RES)
                );

    wire [4:0] DST_REG;
    Mux5 mux5_dstreg_init(
                          .SEL(IDEX_REGDST),
                          .INPUT1(IDEX_RD),
                          .INPUT2(IDEX_RT),
                          .OUT(DST_REG)
                          );

    always @ (posedge Clk)
    begin
        EXMEM_CTRL <= IDEX_CTRL[4:0];
        EXMEM_ZERO <= ZERO;
        EXMEM_ALURES <= ALU_RES;
        EXMEM_WRITE_DATA <= MEM_WRITE_DATA;
        EXMEM_DSTREG <= DST_REG;

    end
```

访存MEM

访存阶段的实现较简单，只有单纯的实例化和段寄存器写入，实现代码如下：

```
wire [31:0] MEM_READ_DATA;
    dataMemory dataMemory_init(
                                .Clk(Clk),
                                .address(EXMEM_ALURES),
                                .writeData(EXMEM_WRITE_DATA),
                                .memRead(EXMEM_MEMREAD),
                                .memWrite(EXMEM_MEMWRITE),
                                .readData(MEM_READ_DATA)
                                 );

    always @ (posedge Clk)
    begin
        MEMWB_CTRL <= EXMEM_CTRL[1:0];
        MEMWB_READ_DATA <= MEM_READ_DATA;
        MEMWB_ALURES <= EXMEM_ALURES;
        MEMWB_DSTREG <= EXMEM_DSTREG;
    end
```

写回WB

写回阶段的实现较简单，只有单纯的实例化，实现代码如下：

```
Mux32 mux32_wb_init(
                        .SEL(MEMWB_MEMTOREG),
                        .INPUT1(MEMWB_READ_DATA),
                        .INPUT2(MEMWB_ALURES),
                        .OUT(REG_WRITE_DATA)
                        );
```

初始化

在`Top.v`的最后部分进行了段寄存器的初始化，代码如下：

```
always @ (reset)
    begin
        PC = 0;
        IFID_PC_PLUS_4 = 0;
        IFID_INST = 0;
        IDEX_RS = 0;
        IDEX_READ_DATA_1 = 0;
        IDEX_READ_DATA_2 = 0;
        IDEX_IMM = 0;
        IDEX_RT = 0;
        IDEX_RD = 0;
        IDEX_CTRL = 0;
        EXMEM_ALURES = 0;
        EXMEM_WRITE_DATA = 0;
        EXMEM_DSTREG = 0;
        EXMEM_CTRL = 0;
        EXMEM_ZERO = 0;
        MEMWB_READ_DATA = 0;
        MEMWB_ALURES = 0;
        MEMWB_DSTREG = 0;
        MEMWB_CTRL = 0;
    end
```

三、仿真测试

测试程序

内存文件中的数据为：

```
00000011 00000000 00000000 00000000          //3
00000100 00000000 00000000 00000000          //4
00000111 00000000 00000000 00000000          //7
00001000 00000000 00000000 00000000          //8
```

编写了如下汇编程序检测处理器是否正常工作：

```
lw $1,8($0)             //$1=7
lw $2,4($0)             //$2=4
ori $3,$2,8000h         //$3=8004h    出现数据相关
andi $4,$1,0002h        //$4=0002h
INC1:
addi $5,1($5)           //$5=$5+1
sll $6,2($2)            //$6=16
srl $7,2($6)            //$7=4
```

```
add $8,$6,$5            //$8=17,18......
sw $8,16($0)            //memFile[4]=$8
j Bran                  //jump to Bran
addi $1,1($1)           //$1=$1+1
addi $1,1($1)           //$1=$1+1
Bran:
beq $2,$7,INC1          //if($2==$7),branch to INC1
addi $1,1($1)           //$1=$1+1
addi $1,1($1)           //$1=$1+1
```

将此汇编翻译为机器指令如下，注意以小端序排列：

```
00001000 00000000 00000001 10001100
00000100 00000000 00000010 10001100
00000000 10000000 01000011 00110100
00000010 00000000 00100100 00110000
00000001 00000000 10100101 00100000
10000000 00110000 00000010 00000000
10000010 00111000 00000110 00000000
00100000 01000000 11000101 00000000
00010000 00000000 00001000 10101100
00001100 00000000 00000000 00001000
00000001 00000000 00100001 00100000
00000001 00000000 00100001 00100000
11110111 11111111 01000111 00010000
00000001 00000000 00100001 00100000
00000001 00000000 00100001 00100000
```

激励文件

激励文件与lab05没有差异，只是删减了部分信号，原因为lab5没意识到可以直接调出`regFile[]`和`memFile[]`进行检查而重新声明了他们，故在此去掉了他们，实现代码如下：

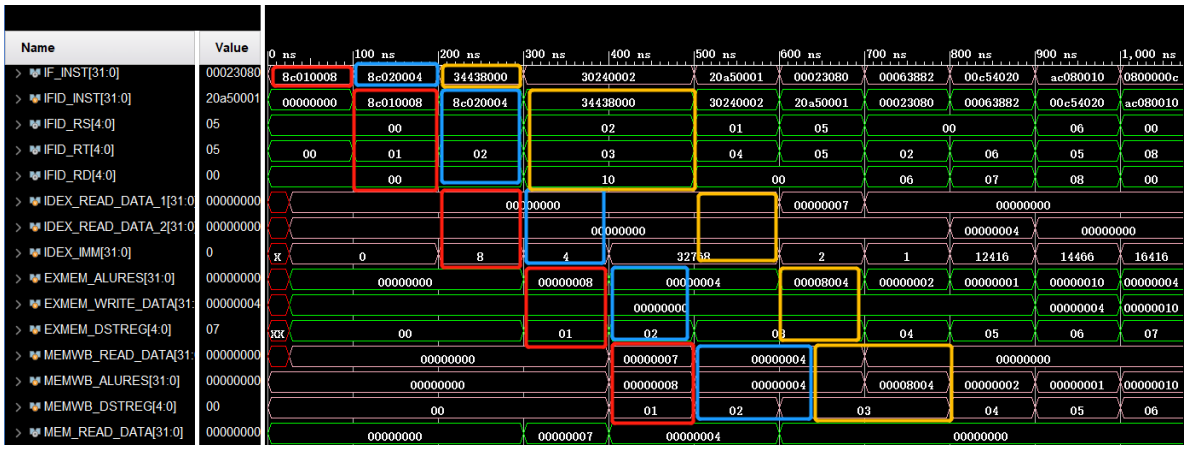
```
module Top_tb(

    );
    reg Clk,reset;
    always #50 Clk=!Clk;
    Top top_init(
                .Clk(Clk),
                .reset(reset)
                );
    reg [5:0] cnt;
    wire [31:0] regFile [0:31];



    initial begin
        $readmemb("C:/Archlabs/lab06/data.txt",
top_init.dataMemory_init.memFile);
        $readmemb("C:/Archlabs/lab06/inst.txt",
top_init.InstMemory_init.insMem);

        Clk = 1;
        reset = 1;
        #25
```

```
        reset = 0;
        #2000;
    end
endmodule
```

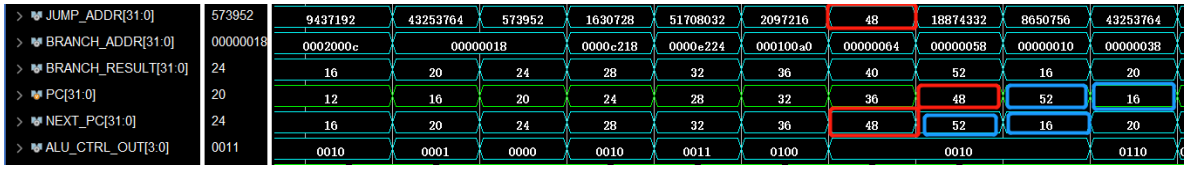
请在测试时更换绿色字体中的绝对路径，否则将找不到所需的两个文件。

仿真波形

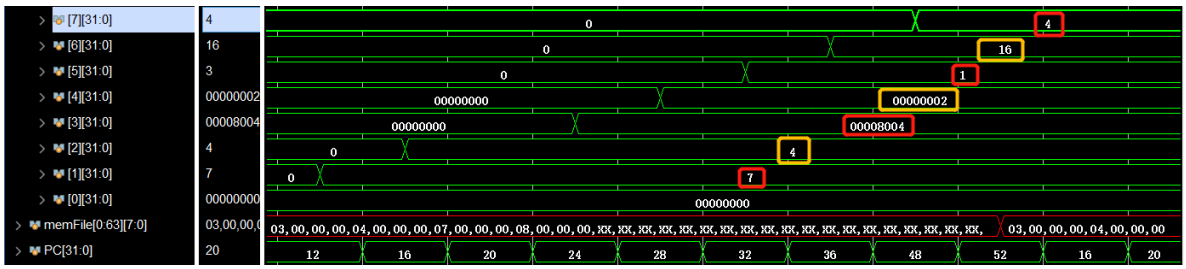
首先展示各段寄存器，图中以三种颜色标注除了流水阶段，可以看出数据是逐段传递的，并且需要`STALL`的第三条`ori`指令也正确停顿，表现形式为其执行时间为前两条的两倍。



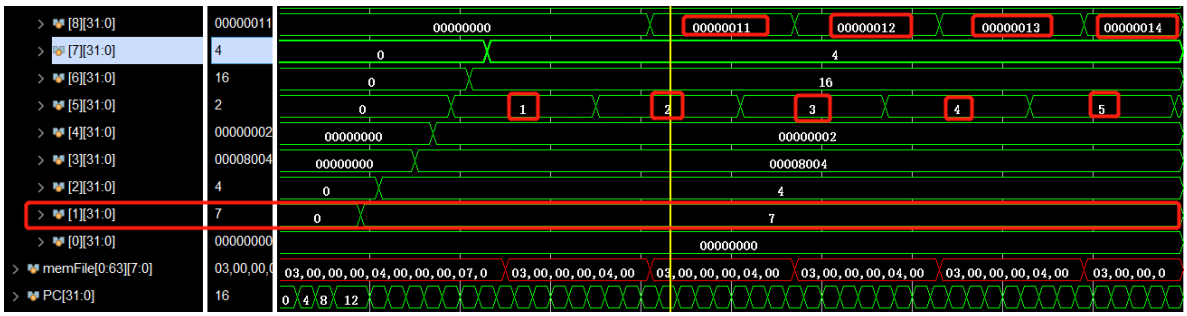
然后展示无条件跳转和条件转移的处理正确性，红色框选为无条件转移，结果正确；蓝色框选为条件转移，由于predict-not-taken故错误执行一个周期，但会在后面被flush掉，不会影响寄存器文件的数据，这个会在后续寄存器文件的截图中给出，可以看到`$1`的数据并未发生更改，条件跳转位置正确。



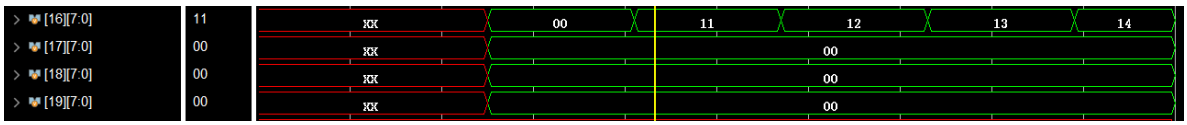
接下来是寄存器文件的数据正确性，可以看出流水线执行的写回结果正确，与汇编相符。



此时已经开始`beq`的循环，可以看到`$5`和`$8`开始了自增，`$1`的值没有发生改变，与预期相符，验证了`beq`的正确性。



最后是内存文件的数据正确性，`sw`指令应该周期性更新`memFile[4]`中的值，下图验证了正确性。（`memFile[16...19]`即前文所谓的`memFile[4]`）



综上，所有CPU Pipeline实现的功能都被验证了正确性，CPU Pipeline实现成功。

四、总结与心得体会

总结

本次实验是这门课的最后一个实验，虽然由于能力原因我只完成了基础实验部分没有做其他拓展，但是还是有不少收获。

本次实验的实现较上一次实验并未有太多难度增加，需要完成的主要就是段寄存器，这需要我们对系统结构中流水线的知识充分了解。

本次实验的难点在于仿真测试时的验证，由于信号众多，很容易头晕目眩，在此提出一个具有可行性的小建议，可以采用对仿真波形分组并更改信号颜色的方法帮助我们辨别。如我在本次调试中，先将一个段寄存器的所有信号放在一起，然后段与段之间调换颜色，这样就不会出现想查看某一个信号而找不到从而影响效率的问题了。

心得体会

为期六次的实验课到我编写到这一段的时候已经基本宣告结束了，很遗憾本学期只能在线上做仿真而无法在学校的实验室里进行硬件的上板验证。

在我高中的时候就思考过CPU到底是如何运行的，CPU是如何利用硬件结构完成0和1的运算等等问题，我想经历了一学期的专业课学习和本次实验我已经初步得到了答案，所以心情非常激动。在浏览IT行业资讯时我也了解到FPGA相关领域是现在较为火热的领域，很高兴能有这样一次机会对这个领域初窥一二。

本门课程帮助我加深了对系统结构这门课的理解，只有实现的时候才明白自己之前对流水线的理解还是非常粗浅的，还有很多问题值得我们去深入思考。本门课程同样培养了我的代码管理能力和调试能力，以前我从来没有遇到如此规模的信号和文件，在本学期实验进行调试时也走了不少弯路，但所幸有这样一次机会帮助我找到了自己的方法。

在此感谢老师这学期的帮助，非常感谢老师在群里对各种问题细致的解答！