5段流水CPU设计实验报告

1. 实验目的

1. 理解计算机指令流水线的协调工作原理，初步掌握流水线的设计和实现原理。
2. 深刻理解流水线寄存器在流水线实现中所起的重要作用。
3. 理解和掌握流水段的划分、设计原理及其实现方法原理。
4. 掌握运算器、寄存器堆、存储器、控制器在流水工作方式下，有别于实验一的设计和实现方法。
5. 掌握流水方式下，通过I/O端口与外部设备进行信息交互的方法。

2. 实验设计

2.1 取指令IF级

在第一个周期结束时(时钟上升沿处), 把指令从存储器去粗的指令写入指令寄存器。

```
module pipeif(
    pcsource,pc,bpc,da,jpc,
    npc,pc4,ins,mem_clock );
    // 输出npc(4路选择器的输出),pc4,ins
    input [31:0] pc,bpc,da,jpc;
    input [1:0] pcsource;
    input mem_clock;

    output [31:0] npc,pc4,ins;

    // bpc: branch addr
    mux4x32 selectnewpc (pc4,bpc,da,jpc,pcsource,npc);
    cla32 pc_plus4 (pc,32'h4,1'b0,pc4);
    lpm_rom_irom irom (pc[7:2],mem_clock,ins);// instruction memory.

endmodule
```

IF级需要传递信息给ID级，通过IF/ID流水线寄存器模块，起承接IF阶段和ID阶段的流水任务。将IF阶段的`pc4`和`ins`传递给ID阶段，分别存储在寄存器`dpc4`和`inst`中。

```
pipeir inst_reg ( pc4,ins,wpcir,clock,resetn,dpc4,inst ); // IF/ID 流水线寄存器
```

2.2 指令译码ID级

第二个周期第一条指令进入ID级，在第二个周期由两项工作同时在做：对第一条指令译码和从指令存储器取第二条指令。

处在ID级的指令存储在寄存器`inst`中，根据指令（为了解决hazard还会根据其他信号），`sc_cu`控制模块产生控制信号：`dwreg`,`dm2reg`,`dwmem`,`djal`,`daluc`,`daluim`,`dshift`。

`regfile`也会根据`inst`指令中的`rs`,`rt`，以及`wdi`,`wrn`,`wwreg`取出相应寄存器的值`qa`,`qb`。`regfile`会在系统clock的下沿进行寄存器写入，也就是给信号从WB阶段传输过来留有半个clock的延迟时间，亦即确保信号稳定。通过相应的`hazard`处理（见2.6）,得到最终值`da`,`db`。

ID/EXE流水线寄存器通过模块`pipedereg`处理：

```
pipedereg de_reg ( dwreg,dm2reg,dwmem,daluc,daluimm,da,db,dimm,
    drn,dshift,djal,dpc4,clock,resetn,ewreg,em2reg,ewmem,ealuc,ealuimm,
    ea,eb,eimm,ern0,eshift,ejal,epc4 );
```

将ID阶段产生的信号`dwreg`,`dm2reg`,`dwmem`,`djal`,`daluc`,`daluim`,`dshift`,`da`,`db`,`dpc4`,`dimm`,`drn`存储在EXE阶段的`ewreg`,`em2reg`,`ewmem`,`ejal`,`ealuc`,`ealuim`,`eshift`,`ea`,`eb`,`epc4`,`eimm`,`ern0`中

2.3 指令执行EXE级的电路

在EXE级，ALU模块进行计算。

```
module pipeexe (ealuc,ealuimm,ea,eb,eimm,eshift,ern0,epc4,ejal,ern,ealu );

     input ealuimm,eshift,ejal;
     input [31:0] ea,eb,eimm,epc4;
    input [4:0] ern0;
    input [3:0] ealuc;

    output [31:0] ealu;
    output [4:0] ern;

    wire [31:0] alua,alub,sa,ealu0,epc8;
    wire zero;

     assign sa = { 27'b0, eimm[10:6] };

    mux2x32 e_alu_a(ea,sa,eshift,alua);

    mux2x32 e_alu_b(eb,eimm,ealuimm,alub);

     assign epc8 = epc4 + 32'h4;

    mux2x32 e_choose_epc(ealu0,epc8,ejal,ealu);

    assign ern = ern0 | {5{ejal}};
    alu al_unit(alua,alub,ealuc,ealu0,zero);
endmodule
```

EXE/MEM流水线寄存器模块负责流水线寄存器的锁存。将`ewreg`,`em2reg`,`ewmem`,`ealu`,`ern`存储到`mwreg`,`mm2reg`,`mwmem`,`malu`,`mrn`中

2.4 存储器访问MEM级电路

在MEM级通过`lpm_ram_dq_dram`模块进行存储器访问，同时`io_mem`负责外设的IO。

```
module pipemem (we,addr,datain,clock,dmem_clk,dataout,
```

```
    io_in_sw, io_out_led,io_out_hex);

    // 类似于 sc_datamem
    // 这次input没有mem_clk，直接输入dmem_clk
    input  [31:0]  addr;
   input  [31:0]  datain;

    input          we, clock,dmem_clk;

    output [31:0] dataout;
    input  [9:0]  io_in_sw;
    output [41:0] io_out_hex;
    output [9:0]  io_out_led;



    wire           dmem_clk, dram_write_enable, io_write_enable;
    wire   [31:0]  mem_out, io_out;

    assign         dram_write_enable = we & ~clock & ~addr[7];
   assign         io_write_enable = we & ~clock & addr[7];
   assign         dataout = addr[7] ? io_out : mem_out;


    lpm_ram_dq_dram  dram(addr[6:2],dmem_clk,datain, dram_write_enable, mem_out
);
    io_mem io(addr[6:2], dmem_clk, datain, io_write_enable, io_out, io_in_sw,
io_out_led,io_out_hex);


endmodule
```

MEM阶段将信号`mwreg`,`mm2reg`,`mmo`,`mrn`,`malu`存储到流水线寄存器`wwreg`,`wrn2reg`,`wrn`,`walu`中

2.5 结果写回WB级

该阶段的逻辑功能部件只包含一个多路器，所以可以仅用一个多路器的实例即可实现该部分。

```
mux2x32 wb_stage( walu,wmo,wm2reg,wdi );
```

2.6 hazard处理

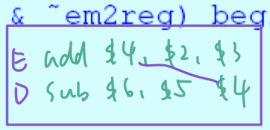
2.6.1 数据相关

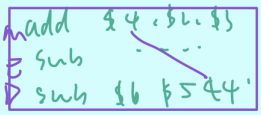
数据相关包括两种处理方式，一种是`forward data`将EXE,MEM阶段的数据提前到ID阶段使用，另一种是`stall`，`lw`指令从数据存储器读出的数据只能从MEM级前推到ID级。这意味着`lw`的后续指令如果与`lw`数据相关，需要把流水线暂停一个周期。

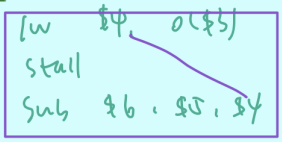
forward data

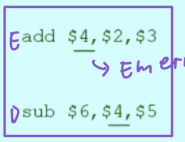
根据`ewreg`,`ern`,`em2reg`寄存器的值决定选择什么值做为`da`,`db`。选择信号`fwda`,`fwdb`作用于四路选择器，选择出ID阶段最后传递给EXE阶段的`da`,`db`的值

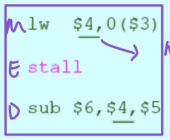
```
fwdb = 2'b00; // default forward b: no hazards
if (ewreg & (ern != 0) & (ern == rt) & ~em2reg) begin
    fwdb = 2'b01; // select exe_alu
end else begin
    if (mwreg & (mrn != 0) & (mrn == rt) & ~mm2reg) begin
        fwdb = 2'b10; // select mem_alu
    end else begin
        if (mwreg & (mrn != 0) & (mrn == rt) & mm2reg) begin
            fwdb = 2'b11; // select mem_lw
        end
    end
end

fwda = 2'b00; // default forward a: no hazards
if (ewreg & (ern != 0) & (ern == rs) & ~em2reg) begin
    fwda = 2'b01; // select exe_alu
end else begin
    if (mwreg & (mrn != 0) & (mrn == rs) & ~mm2reg) begin
        fwda = 2'b10; // select mem_alu
    end else begin
        if (mwreg & (mrn != 0) & (mrn == rs) & mm2reg) begin
            fwda = 2'b11; // select mem_lw
        end
    end
end
```











stall

流水线由于lw数据相关而需要暂停的条件为：

```
assign wpcir = ~ (ewreg & em2reg & (ern != 0 ) & (i_rs & (ern == rs) | i_rt & (ern == rt)));
```

通过`wpcir`信号控制暂停流水线

```
// 如果resetn为0，则pc为0
// 如果wpcir为1，则pc为npc  npc(4路选择器的输出)
// 如果要stall 则wpcir为0，也就是pc值不更新
dffe32 next_pc(npc, clock, resetn, wpcir, pc);
```

2.6.2 控制相关

流水线CPU在执行转移和跳转指令时会出现控制相关的问题，即在实际转向目标地址之前，转移或跳转指令的后续指令已经取到流水线中了。在我们的流水线CPU中，引起转移的指令有：`jr`,`beq`,`bne`,`j`,`jal`。

由于`j`,`jal`,`jr`是无条件转移指令，在`ID`级确定是否转移没有问题。而`beq`,`bne`是条件转移指令，正常情况下在EXE阶段才能知道是否跳转。可以在`ID`级比较出两个寄存器数据是否相等，使用异或门和或非门实现这种比较。

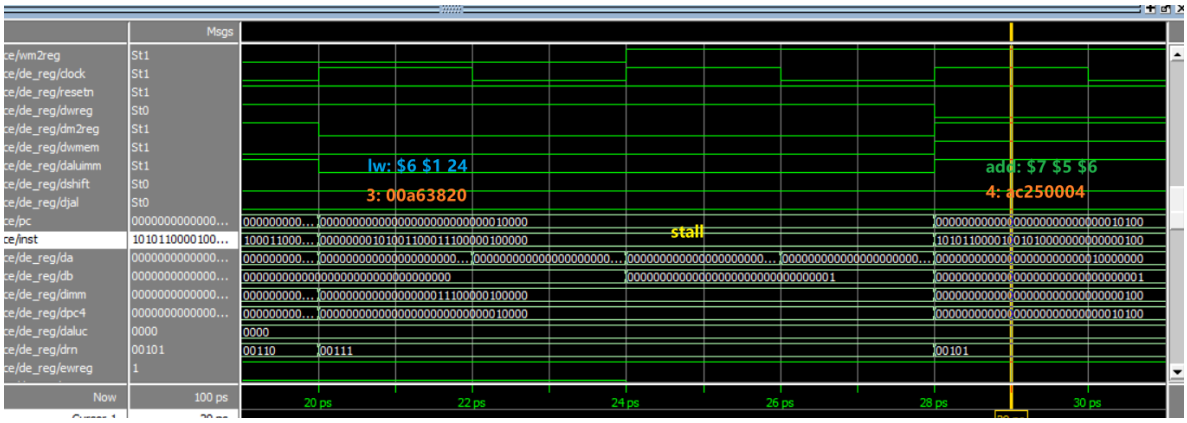
```
assign z = ~|(da^db);
```

3. 仿真测试

通过仿真，可以看到流水线CPU的行为符合预期

测试输入为sw的第一个被加数为1，第二个被加数也为1

3.1 stall

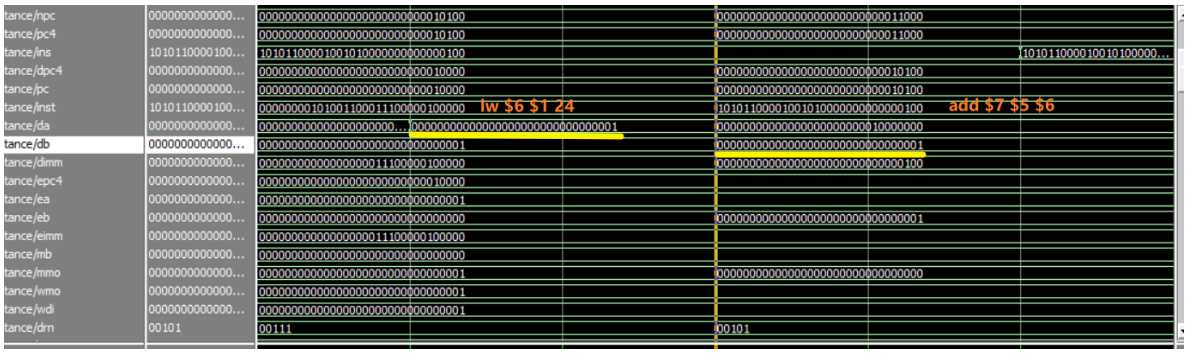


根据汇编代码可知：

```
lw $6 $1 24          # input inport1 to $5
add $7 $5 $6         # add inport0 with inport1 to $7
```

lw指令的下一句使用了lw的目标寄存器，所以我们需要进行一个stall，也就是通过控制`wpcir`控制`pc`等待一个周期，不更新。确保`add`指令使用的`$6`寄存器的值是最新的

3.2 forward data



根据汇编代码可知：

```
2: lw $5 $1 20          # input inport0 to $4
3: lw $6 $1 24          # input inport1 to $5
4: add $7 $5 $6         # add inport0 with inport1 to $6
```

```
 2: 8c260018;
 3: 00a63820;
 4: ac250004;
```

第四句汇编需要使用的寄存器`$6`的值，要通过forward data，从mmo寄存器中取得。

根据图中信息，db确实为1，符合预期

3.3 control hazard

根据汇编代码：

```
1: beq $1 $2 1
2: addi $1 $1 1
3: addi $2 $2 1
4: j 0
```

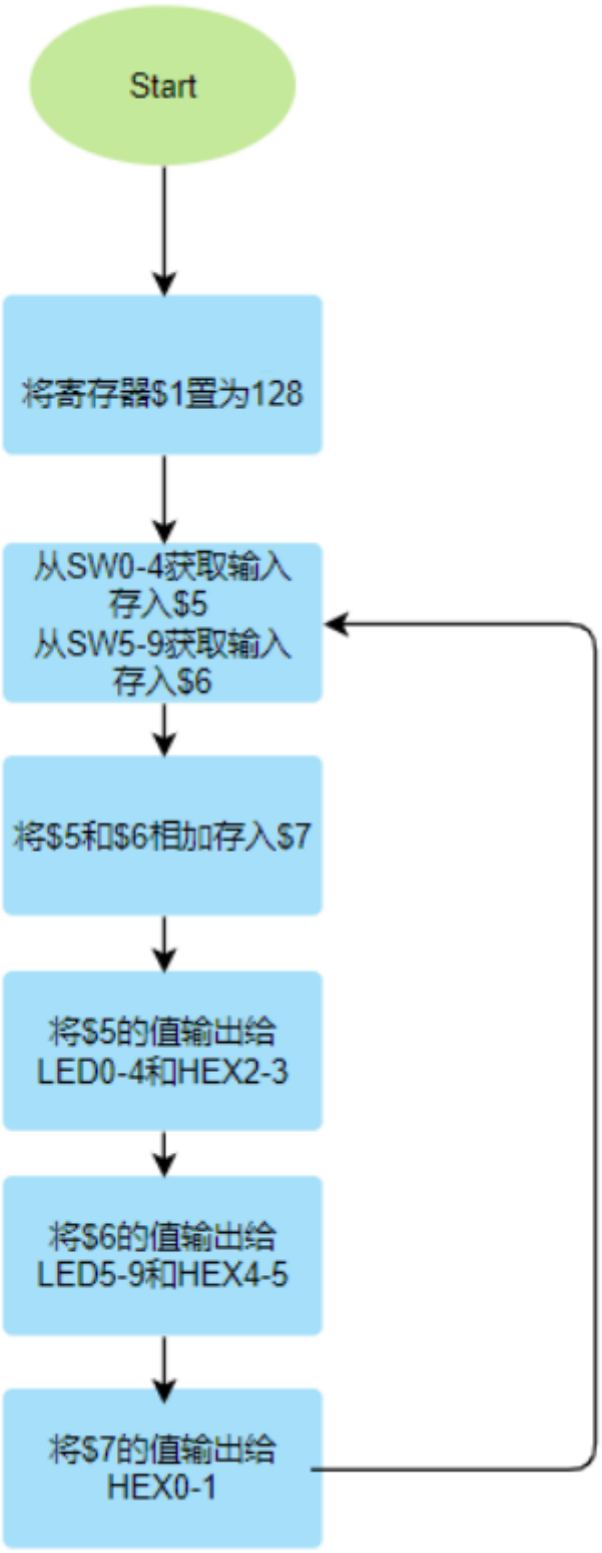
可以看到根据汇编代码的逻辑，第一句寄存器1和寄存器2判等成功，pc应该加8，跳过第二句，直接执行第三句。观察仿真结果截图，的确是如此，符合预期。

综上，所有的hazard该流水线CPU都可以正常解决。

4. 汇编代码以及流程图

汇编代码提供的功能和实验二相同，完成功能，将sw[9:5]作为第一个输入的被加数，显示在数码管HEX4,HEX5 ，将sw[4:0]作为第二个输入的被加数，显示在数码管HEX2,HEX3。最后将相加结果显示在数码管HEX0,HEX1。

```
addi $1 $1 128  # $1 = 128
lw $5 $1 20     # $5 <- addr(128+20)    SW0-4
lw $6 $1 24     # $6 <- addr(128+24)    SW5-9
add $7 $5 $6    # $7 = $5 + $6
sw $5 $1 4      # $5 -> addr(128+4)     HEX2,HEX3
sw $5 $1 12     # $5 -> addr(128+12)    LED0-4
sw $6 $1 8      # $6 -> addr(128+8)     HEX4,HEX5
sw $6 $1 16     # $6 -> addr(128+16)    LED5-9
sw $7 $1 0      # $7 -> addr(128+0)     HEX0,HEX1
j 1
```



以上的汇编代码可以测试forward data 和 stall，不能测试control hazard，所以另外编写control hazard的测试汇编代码如下：

```
beq $1 $2 1
addi $1 $1 1
addi $2 $2 1
j 0
```

因程序较为简单，不绘制流程图，具体解释见3.3