实验3 流水线CPU模块设计

518030910408 周韧

一. 实验环境

本次实验在Quartus II 13.0(64-bit)开发软件中进行。

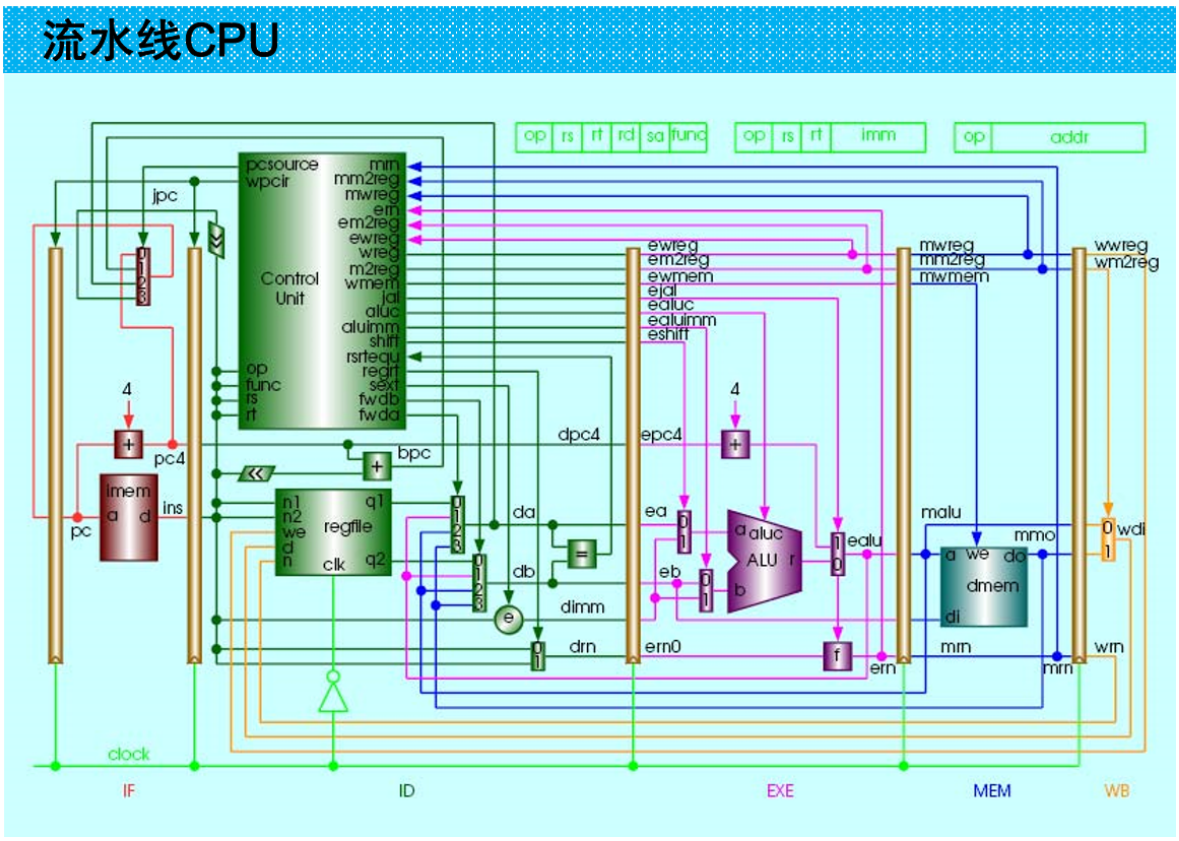
二. 实验目的

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。

三. 设计思路分析

1.分析顶层结构

本次实验指导中已经给我们提供了完整的顶层设计代码，我们需要做的是分析顶层代码结构并依据此来完成各个模块的设计。我们的设计主要依据下面这张原理图。



首先我们建立pipelined_computer.v 的顶层文件，列举出流水线五个阶段的各个核心模块的组成，而像多路选择器这样的比较简单非核心模块我们这里就不再重点描述和展示：

> IF阶段，这一阶段主要包含如下核心模块
>
> - pipepc.v， 实现的是依据时钟周期clock来控制pc的更新，将pc更新为npc
> - pipeif.v，实现的的是从imem中取指和选择不同的npc来源
> - lpm_rom_irom, 指令存储器
> - 边界：pipeir.v，实现的是将各种信号存入IF/ID寄存器中，如果有控制冒险在这里暂停住
>
> ID阶段，在pipeid.v中实现，这一部分主要包含三个核心模块
>
> - regfile.v，寄存器堆，以时序逻辑控制32个寄存器读取操作
> - pipe_cu.v，用于产生各种控制信号，数据冒险和控制冒险的wpcir，bubble信号也在这里产生
> - 边界：pipedereg.v，将各种信号存入ID/EXE寄存器中
>
> EXE阶段，在pipeexe.v中实现，这一部分主要两个核心模块
>
> - alu.v，用于各种运算的操作（包含了汉明码运算）
> - 边界：pipeemreg.v，将各种信号存入EXE/MEM寄存器中
>
> MEM阶段，在pipemem.v中实现，这一部分主要有四个核心模块
>
> - dram单元，即数据存储器，向其中存储数据
> - io_input_reg.v，控制IO设备向CPU中输入数据
> - io_out_reg.v，用于CPU向IO中输入数据
> - 边界：pipemwreg，将各种信号存入MEM/WB寄存器中
>
> WB阶段，这一部分使用一个多路选择器实现写回阶段

列出顶层文件的代码如下：

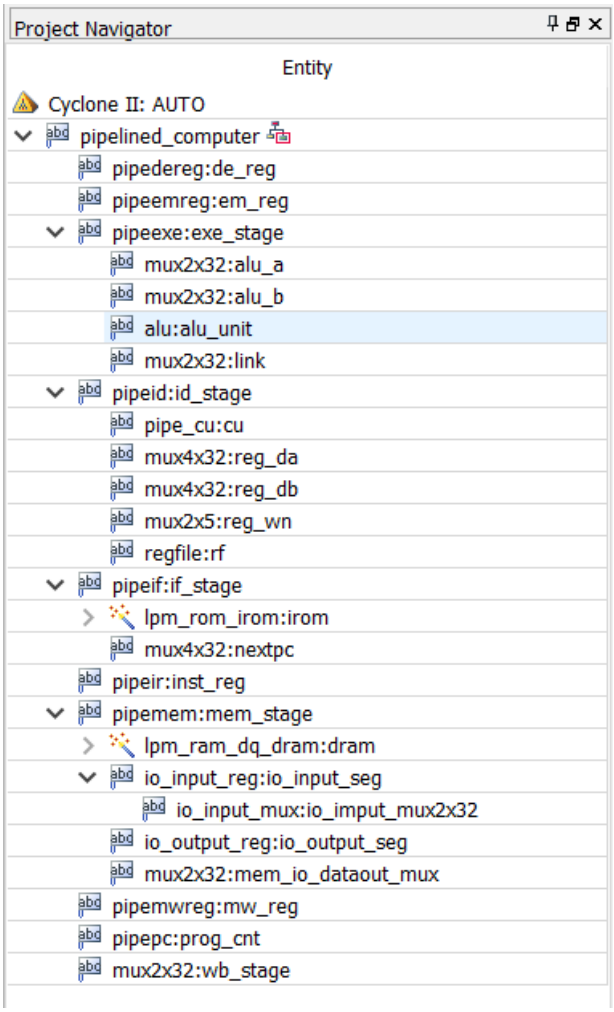
```
// pipelined_computer.v
module pipelined_computer (resetn, clock, mem_clock, opc, oinst, oins, oealu,
omalu, owalu,onpc, in_port0, in_port1, out_port0, out_port1, out_port2,
out_port3, wpcir, obubble);
//定义顶层模块 pipelined_computer，作为工程文件的顶层入口，如图 1-1 建立工程时指定。
        input resetn, clock;
        output mem_clock;
        output obubble;
        assign mem_clock = ~clock;
        //定义整个计算机 module 和外界交互的输入信号，包括复位信号 resetn、时钟信号
clock、
        //以及一个和 clock 同频率但反相的 mem_clock 信号。mem_clock 用于指令同步 ROM
        //和数据同步 RAM 使用，其波形需要有别于实验一。
        //这些信号可以用作仿真验证时的输出观察信号。
        input [5:0] in_port0, in_port1;
        output  [31:0]  out_port0,  out_port1,  out_port2,  out_port3;  /*
output  [6:0]
        out_port0,out_port1,out_port2,out_port3;*/
        wire [31:0] real_out_port0,real_out_port1,real_out_port2,real_out_port3;
        wire [31:0] real_in_port0 = {26'b00000000000000000000000000,in_port0};
        wire [31:0] real_in_port1 = {26'b00000000000000000000000000,in_port1};
        assign out_port0 = real_out_port0[31:0];//assign out_port0 =
real_out_port0[6:0];
```

```
        assign out_port1 = real_out_port1[31:0];//assign out_port0 =
real_out_port1[6:0];
        assign out_port2 = real_out_port2[31:0];//assign out_port0 =
real_out_port2[6:0];
        assign out_port3 = real_out_port3[31:0];//assign out_port0 =
real_out_port3[6:0];
        //IO 口的定义，宽度可根据自己设计选择。
        wire [31:0] pc,ealu,malu,walu;
        output [31:0] opc,oealu,omalu,owalu; // for watch
        wire owmem, write_datamem_enablem;
        assign obubble = bubble;
        assign owmem = dwmem;
        assign opc = pc;
        assign oealu = ealu;
        assign omalu = malu;
        assign owalu = walu;
        output [31:0] onpc,oins,oinst; // for watch
        assign onpc=npc;
        assign oins=ins;
        assign oinst=inst;
        //模块用于仿真输出的观察信号。缺省为 wire 型。为了便于观察内部关键信号，将其接到
        //输出管脚。不输出也一样，只是仿真时候要从内部信号里去寻找。
        wire [31:0] bpc,jpc,pc4,npc,ins,inst;
        //模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。IF 取指令阶段。
        wire [31:0] dpc4,da,db,dimm,dsa;
        //模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。ID 指令译码阶段。
        wire [31:0] epc4,ea,eb,eimm,esa;
        //模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。EXE 指令运算阶段。
        wire [31:0] mb,mmo;
        //模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。MEM 访问数据阶段。
        wire [31:0] wmo,wdi;
        //模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。WB 回写寄存器阶段。
        wire [4:0] ern0,ern,drn,mrn,wrn;
        //模块间互联，通过流水线寄存器传递结果寄存器号的信号线，寄存器号（32 个）为 5bit。
        wire [4:0] drs,drt,ers,ert;
        // 模块间互联，通过流水线寄存器传递 rs、rt 寄存器号的信号线，寄存器号（32 个）为
5bit。
        wire [3:0] daluc,ealuc;
        //ID 阶段向 EXE 阶段通过流水线寄存器传递的 aluc 控制信号，4bit。
        wire [1:0] pcsource;
        //CU 模块向 IF 阶段模块传递的 PC 选择信号，2bit。
        //wire wpcir;
        output wpcir;
        // CU 模块发出的控制流水线停顿的控制信号，使 PC 和 IF/ID 流水线寄存器保持不变。
        wire dwreg,dm2reg,dwmem,daluimm,dshift,djal; //id stage
        // ID 阶段产生，需往后续流水级传播的信号。
        wire ewreg,em2reg,ewmem,ealuimm,eshift,ejal; //exe stage
        //来自于 ID/EXE 流水线寄存器，EXE 阶段使用，或需要往后续流水级传播的信号。
        wire mwreg,mm2reg,mwmem; //mem stage
        //来自于 EXE/MEM 流水线寄存器，MEM 阶段使用，或需要往后续流水级传播的信号。
        wire wwreg,wm2reg; //wb stage
        //来自于 MEM/WB 流水线寄存器，WB 阶段使用的信号。
        wire ezero,mzero;
        //模块间互联，通过流水线寄存器传递的 zero 信号线
        wire ebubble,dbubble,bubble;
        //模块间互联，通过流水线寄存器传递的流水线冒险处理 bubble 控制信号线
        pipepc prog_cnt(npc,wpcir,clock,resetn,pc);
        //程序计数器模块，是最前面一级 IF 流水段的输入。
```

```
        pipeif if_stage(pcsource,pc,bpc,da,jpc,npc,pc4,ins,mem_clock); // IF
stage
        //IF 取指令模块，注意其中包含的指令同步 ROM 存储器的同步信号，
        //即输入给该模块的 mem_clock 信号，模块内定义为 rom_clk。// 注意 mem_clock。
        //实验中可采用系统 clock 的反相信号作为 mem_clock（亦即 rom_clock），
        //即留给信号半个节拍的传输时间。
        pipeir inst_reg(pc4,ins,wpcir,clock,resetn,dpc4,inst,bubble); // IF/ID
流水线寄存器
        //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。
        //在 clock 上升沿时，将 IF 阶段需传递给 ID 阶段的信息，锁存在 IF/ID 流水线寄存器
        //中，并呈现在 ID 阶段。
        pipeid id_stage(mwreg,mrn,ern,ewreg,em2reg,mm2reg,dpc4,inst,/*ins,*/
        wrn,wdi,ealu,malu,mmo,wwreg,mem_clock,resetn,
        bpc,jpc,pcsource,wpcir,dwreg,dm2reg,dwmem,daluc,
        daluimm,da,db,dimm,dsa,drn,dshift,djal,bubble);
        //,mzero,drs,drt/*,npc*/,ebubble,dbubble); // ID stage
        //ID 指令译码模块。注意其中包含控制器 CU、寄存器堆、及多个多路器等。
        //其中的寄存器堆，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。
        //该阶段 CU 产生的、要传播到流水线后级的信号较多。
        pipedereg de_reg(/*dbubble,drs, drt,*/ dwreg, dm2reg, dwmem,
        daluc, daluimm, da, db, dimm, dsa, drn, dshift, djal,
        dpc4, clock, resetn, /*ebubble, ers, ert,*/ ewreg, em2reg,
        ewmem, ealuc, ealuimm, ea, eb, eimm, esa, ern0, eshift, ejal, epc4);
        // ID/EXE 流水线寄存器
        //ID/EXE 流水线寄存器模块，起承接 ID 阶段和 EXE 阶段的流水任务。
        //在 clock 上升沿时，将 ID 阶段需传递给 EXE 阶段的信息，锁存在 ID/EXE 流水线
        //寄存器中，并呈现在 EXE 阶段。
        pipeexe exe_stage (ealuc, ealuimm, ea, eb, eimm, esa, eshift, ern0,
        epc4, ejal, ern, ealu/*, ezero, ert, wrn, wdi, malu, wwreg*/); // EXE
stage
        //EXE 运算模块。其中包含 ALU 及多个多路器等。
        pipeemreg
em_reg(ewreg,em2reg,ewmem,ealu,eb,ern,/*ezero,*/clock,resetn,mwreg,
        mm2reg,mwmem,malu,mb,mrn/*,mzero*/); // EXE/MEM 流水线寄存器
        //EXE/MEM 流水线寄存器模块，起承接 EXE 阶段和 MEM 阶段的流水任务。
        //在 clock 上升沿时，将 EXE 阶段需传递给 MEM 阶段的信息，锁存在 EXE/MEM
        //流水线寄存器中，并呈现在 MEM 阶段。
        pipemem mem_stage (mwmem,malu,mb,clock,mem_clock,mmo,resetn,
        real_in_port0,real_in_port1,real_out_port0,real_out_port1,
        real_out_port2,real_out_port3, write_datamem_enable); //  MEM stage
        //MEM 数据存取模块。其中包含对数据同步 RAM 的读写访问。// 注意 mem_clock。
        //输入给该同步 RAM 的 mem_clock 信号，模块内定义为 ram_clk。
        //实验中可采用系统 clock 的反相信号作为 mem_clock 信号（亦即 ram_clk），
        //即留给信号半个节拍的传输时间，然后在 mem_clock 上沿时，读输出、或写输入。
        pipemwreg mw_reg(mwreg,mm2reg,mmo,malu,mrn,clock,resetn,
        wwreg,wm2reg,wmo,walu,wrn); // MEM/WB 流水线寄存器
        //MEM/WB 流水线寄存器模块，起承接 MEM 阶段和 WB 阶段的流水任务。
        //在 clock 上升沿时，将 MEM 阶段需传递给 WB 阶段的信息，锁存在 MEM/WB
        //流水线寄存器中，并呈现在 WB 阶段。
        mux2x32 wb_stage(walu,wmo,wm2reg,wdi); //  WB stage
        //WB 写回阶段模块。事实上，从设计原理图上可以看出，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。
        //当然，如果专门写一个完整的模块也是很好的。
endmodule
```

与实验2中不同，这次实验中mem_clock不需要降频，只需要对clock取反即可。

顶层模块设计图如下所示



2. 核心模块设计

2.1 IF阶段

在pipepc.v中，若resetn信号置为0，则将pc置为-4；若wpcir信号为0，则认为存在load指令导致的数据冒险，暂停流水线一个周期；其他正常情况，则将pc更新为npc；

```
// pipepc.v
module pipepc(npc, wpcir, clock, resetn, pc);
// npc是new_pc或者next_pc, wpcir是lw数据冒险标记
    input [31:0] npc;
   input clock,resetn,wpcir;
   output [31:0] pc;
   reg [31:0] pc;
   always @(posedge clock)
    begin
      if (resetn == 0) // 重置
        begin
          pc <= -4;
      end
        else
        if (wpcir != 0) // wpcir若为0则流水线停顿
        begin
          pc <= npc;  // 更新pc值为new pc
      end
    end
endmodule
```

在pipeif.v中，要使用两路选择器根据输入的pcsource信号选择npc的来源，根据pc的[7:2]为在imem中选择指令数据。

```
// pipeif.v
module pipeif(pcsource, pc, bpc, da, jpc, npc, pc4, ins, mem_clock);
// pcsource选择pc的来源,da是从寄存器中取出的pc值
    input [1:0] pcsource;
    input mem_clock;
    input [31:0] pc, bpc, jpc, da;
    output [31:0] npc, pc4, ins;

    wire [31:0] npc, pc4, ins;
    assign pc4 = pc + 4;

    mux4x32 nextpc( pc4, bpc, da, jpc, pcsource, npc); // 下一个pc值
    lpm_rom_irom irom(pc[7:2], mem_clock, ins);

endmodule
```

在pipeir中，如果resetn信号为高位，则将pc、inst全部置为0；如果resetn为低位，wpcir和bubble均为高位，则正常将pc和inst信号传到下一级，如果wpcir信号为高而bubble信号为低，则认为产生控制冒险，将inst信号冲刷掉。

```
// pipeir.v
module pipeir (pc4, ins, wpcir, clock, resetn, dpc4, inst, bubble);
//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。
//在 clock 上升沿时，将 IF 阶段需传递给 ID 阶段的信息，锁存在 IF/ID 流水线寄存器
//中，并呈现在 ID 阶段。
    input [31:0] pc4, ins;
    input wpcir, clock, resetn, bubble;
    output [31:0] dpc4, inst;
    reg    [31:0]  dpc4, inst;

    always @(posedge clock or negedge resetn)
    begin
        if (resetn == 0)
        begin
            dpc4 <= 0;
            inst <= 0;
        end
        else if(wpcir & bubble) //流水线正常运行
        begin
            dpc4 <= pc4;
            inst <= ins;
        end
        else if(wpcir & ~bubble)    // 出现控制冒险
        begin
            inst <= 0;
        end
    end
endmodule
```

2.2 ID阶段

ID阶段内容较多，首先给出pipeid.v模块，里面包含对pipe_cu.v，regfile.v和几个多路选择器的例化，其中还包含例如rsrtequ，dimm等相应的很多控制信号的产生，需要的控制信号可以参见报告中给的流水线CPU的整体图。

```
// pipeid.v
module pipeid(mwreg, mrn, ern, ewreg, em2reg, mm2reg, dpc4, inst, wrn, wdi,
ealu, malu, mmo, wwreg, clock, resetn, bpc, jpc, pcsource, wpcir, dwreg, dm2reg,
dwmem, daluc, daluimm, da, db, dimm, dsa, drn, dshift, djal, bubble);
//,npc,ebubble,dbubble);    // ID stage

//ID 指令译码模块。注意其中包含控制器 CU、寄存器堆、及多个多路器等。
//其中的寄存器堆，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。
//该阶段 CU 产生的、要传播到流水线后级的信号较多。

    input [4:0] mrn, ern, wrn;
    input   mm2reg, em2reg, mwreg, ewreg, wwreg, clock, resetn;
    input [31:0] inst, wdi, ealu, malu, mmo, dpc4;
    output [31:0] bpc, dimm, jpc, da, db, dsa;
    output [1:0] pcsource;
    output wpcir, dwreg, dm2reg, dwmem, daluimm, dshift, djal, bubble;
    output [3:0] daluc;
    output [4:0] drn;

    wire [31:0] q1, q2, da, db;
    wire [1:0] fwda, fwdb;
    wire rsrtequ = (da == db);
    wire regrt, sext;
    wire e = sext & inst[15];
    wire [31:0] dimm = {{16{e}}, inst[15:0]};
    wire [31:0] jpc = {dpc4[31:28],inst[25:0],1'b0,1'b0};
    wire [31:0] dsa = {27'b0, inst[10:6]};
    wire [31:0] offset = {{14{e}},inst[15:0],1'b0,1'b0};
    wire [31:0] bpc = dpc4 + offset;

    pipe_cu cu(inst[31:26], inst[5:0], rsrtequ, dwmem, dwreg, regrt, dm2reg,
daluc, dshift,
    //              opcode,     funct,  rs==rt启动直通
    daluimm, pcsource, djal, sext, wpcir, bubble, inst[25:21], inst[20:16], mrn,
mm2reg, mwreg, ern, em2reg, ewreg, fwda, fwdb);//控制单元
    //                                                              rs,
rt
    regfile rf(inst[25:21], inst[20:16], wdi, wrn, wwreg, clock, resetn, q1,
q2);  //寄存器堆
    //                  n1:rs,      n2:rt,    d,   n,   we,   clock,  clrn,  两输
出
    mux4x32 reg_da(q1, ealu, malu, mmo, fwda, da); // 四选一
    //                      四个选项,             依据，结果
    mux4x32 reg_db(q2, ealu, malu, mmo, fwdb, db);
    mux2x5 reg_wn(inst[15:11], inst[20:16], regrt, drn);
    //                      rd,         rt,       依据，  结果

endmodule
```

regfile.v的实现方式和实验二中基本一致，这里不再赘述。

```
// regfile.v
module regfile(rna, rnb, d, wn, we, clk, clrn, qa, qb);
   input [4:0] rna, rnb, wn;
   input [31:0] d;
   input we, clk, clrn;

   output [31:0] qa,qb;

   reg [31:0] register [1:31]; // r1 - r31

   assign qa = (rna == 0)? 0 : register[rna]; // read
   assign qb = (rnb == 0)? 0 : register[rnb]; // read

   always @(posedge clk or negedge clrn)
    begin
      if (clrn == 0)  // reset
        begin
         integer i;
         for (i = 1; i < 32; i = i + 1)
            register[i] <= 0;
      end
        else
        begin
         if((wn != 0) && (we == 1))          // write
            register[wn] <= d;
      end
   end
endmodule
```

pipe_cu.v是一个比较关键的模块，与实验二中最主要的区别就在于wpcir、bubble、pcsource等信号的产生。当wpcir为0时，说明出现了load导致的数据冒险，要把一些信号置0；当bubble信号为0时，说明出现了控制冒险；在这个文件中最后还加入了对fwda、fwdb直通信号的生成，这决定着是否要启动直通以及启动什么样的直通。

```
// pipe_cu.v
module pipe_cu(op, func, rsrtequ, wmem, wreg, regrt, m2reg, aluc, shift,aluimm,
pcsource, jal, sext, wpcir, bubble, rs, rt, mrn, mm2reg, mwreg, ern, em2reg,
ewreg, fwda, fwdb);
   input [5:0] op,func;
   input rsrtequ, mwreg, ewreg, mm2reg, em2reg;
   input [4:0] rs, rt, mrn, ern;
   output wreg, regrt, jal, m2reg, shift, aluimm, sext, wmem, wpcir, bubble;
   output [3:0] aluc;
   output [1:0] pcsource, fwda, fwdb;
   reg [1:0] fwda, fwdb;
   wire r_type = ~|op;  // 是否是R型指令



    // R型指令
   wire i_add = r_type & func[5] & ~func[4] & ~func[3] &  ~func[2] & ~func[1] &
~func[0];       //100000
   wire i_sub = r_type & func[5] & ~func[4] & ~func[3] &~func[2] &  func[1] &
~func[0];      //100010
   wire i_and = r_type & func[5] & ~func[4] & ~func[3] &func[2] & ~func[1] &
~func[0];       //100100
```

```
   wire i_or  = r_type & func[5] & ~func[4] & ~func[3] &func[2] & ~func[1] &
 func[0];       //100101
   wire i_xor = r_type & func[5] & ~func[4] & ~func[3] &func[2] &  func[1] &
~func[0];       //100110
   wire i_sll = r_type & ~func[5] & ~func[4] & ~func[3] &~func[2] & ~func[1] &
~func[0];     //000000
   wire i_srl = r_type & ~func[5] & ~func[4] & ~func[3] &~func[2] &  func[1] &
~func[0];     //000010
   wire i_sra = r_type & ~func[5] & ~func[4] & ~func[3] &~func[2] &  func[1] &
 func[0];     //000011
   wire i_jr  = r_type & ~func[5] & ~func[4] &  func[3] &~func[2] & ~func[1] &
~func[0];     //001000

    // I型指令
   wire i_addi = ~op[5] & ~op[4] &  op[3] & ~op[2] & ~op[1] & ~op[0]; //001000
   wire i_andi = ~op[5] & ~op[4] &  op[3] &  op[2] & ~op[1] & ~op[0]; //001100
   wire i_ori  = ~op[5] & ~op[4] &  op[3] &  op[2] & ~op[1] &  op[0]; //001101
   wire i_xori = ~op[5] & ~op[4] &  op[3] &  op[2] &  op[1] & ~op[0]; //001110
   wire i_lw   =  op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] &  op[0]; //100011
   wire i_sw   =  op[5] & ~op[4] &  op[3] & ~op[2] &  op[1] &  op[0]; //101011
   wire i_beq  = ~op[5] & ~op[4] & ~op[3] &  op[2] & ~op[1] & ~op[0]; //000100
   wire i_bne  = ~op[5] & ~op[4] & ~op[3] &  op[2] & ~op[1] &  op[0]; //000101
   wire i_lui  = ~op[5] & ~op[4] &  op[3] &  op[2] &  op[1] &  op[0]; //001111
   wire i_j    = ~op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] & ~op[0]; //000010
   wire i_jal  = ~op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] &  op[0]; //000011

    assign wpcir = ~(em2reg & ( ern == rs | ern == rt )) ;  //lw的数据冒险  可能需
要停顿
    // 如果wpcir为1 那么插入气泡停顿 将所有的控制信号置0

   assign pcsource[1] = i_jr | i_j | i_jal;
   assign pcsource[0] = ( i_beq & rsrtequ ) | (i_bne & ~rsrtequ) | i_j | i_jal ;
    assign bubble = ~(pcsource[0] | pcsource[1]);
   assign wreg = wpcir & (i_add | i_sub | i_and | i_or   | i_xor  |
                i_sll | i_srl | i_sra | i_addi | i_andi |
                i_ori | i_xori | i_lw | i_lui  | i_jal);

   assign aluc[3] = wpcir & i_sra;
   assign aluc[2] = wpcir & (i_sub | i_or | i_lui | i_srl | i_sra | i_ori);
   assign aluc[1] = wpcir & (i_xor | i_lui | i_sll | i_srl | i_sra | i_xori);
   assign aluc[0] = wpcir & (i_and | i_or | i_sll | i_srl | i_sra | i_andi |
i_ori);
   assign shift   = wpcir & (i_sll | i_srl | i_sra);

   assign aluimm  = wpcir & (i_addi | i_andi | i_ori | i_xori | i_lw | i_sw);
   assign sext    = wpcir & (i_addi | i_lw | i_sw | i_beq | i_bne);
   assign wmem    = wpcir & i_sw;
   assign m2reg   = wpcir & i_lw;
   assign regrt   = wpcir & (i_addi | i_andi | i_ori | i_xori | i_lw | i_lui);
   assign jal     = wpcir & i_jal;


// fwda和fwda
   always @(*)
   begin
        if(ewreg & ~ em2reg & (ern != 0) & (ern == rs))  // 将上一条指令的alu结果直
通 如果上一条指令是lw的话 那么会停顿一个时钟周期 所以直通了也无所谓
                fwda<=2'b01;
```

```
        else if(mwreg & ~ mm2reg & (mrn != 0) & (mrn == rs)) //将前两条指令的alu结果直通
                fwda<=2'b10;
        else if(mwreg & mm2reg & (mrn != 0) & (mrn == rs))  // 将前两条指令的数据RAM的输出直通
                fwda<=2'b11;
        else
                fwda<=2'b00;  // 无需直通
    end


    always @(*)
    begin
       if(ewreg & ~ em2reg &(ern != 0) & (ern == rt)) //将上一条指令的alu结果直通
                fwdb<=2'b01;
       else if(mwreg & ~ mm2reg & (mrn != 0) & (mrn == rt))  //将前两条指令的alu结果直通
             fwdb<=2'b10;
       else if(mwreg & mm2reg & (mrn != 0) & (mrn == rt))   // 将前两条指令的数据RAM的输出直通
             fwdb<=2'b11;
         else
             fwdb<=2'b00; // 无需直通
    end

endmodule
```

pipedereg.v模块就是在resetn不为0的情况下，将指令传到下一阶段。

```
// pipedereg.v
module pipedereg (dwreg, dm2reg, dwmem, daluc, daluimm, da, db, dimm, dsa, drn, dshift, djal, dpc4, clock, resetn, ewreg, em2reg, ewmem, ealuc, ealuimm, ea, eb, eimm, esa, ern0, eshift, ejal, epc4); // ID/EXE 流水线寄存器
//ID/EXE 流水线寄存器模块，起承接 ID 阶段和 EXE 阶段的流水任务。
//在 clock 上升沿时，将 ID 阶段需传递给 EXE 阶段的信息，锁存在 ID/EXE 流水线
//寄存器中，并呈现在 EXE 阶段。

    input dwreg, dm2reg, dwmem, daluimm, dshift, djal, clock, resetn;
    input [3:0] daluc;
    input [31:0] dimm, da, db, dpc4, dsa;
    input [4:0] drn;

    output ewreg, em2reg, ewmem, ealuimm, eshift, ejal;
    output [3:0]  ealuc;
    output [31:0] eimm, ea, eb, epc4, esa;
    output [4:0]  ern0;
    reg ewreg, em2reg, ewmem, ealuimm, eshift, ejal;
    reg [3:0] ealuc;
    reg [31:0] eimm, ea, eb, epc4, esa;
    reg [4:0] ern0;

    always @( posedge clock or negedge resetn)
    begin
        if (resetn == 0)  //清零
        begin
            ewreg <= 0;
            em2reg <= 0;
```

```
            ewmem <= 0;
            ealuimm <= 0;
            eshift <= 0;
            ejal <= 0;
            ealuc <= 0;
            eimm <= 0;
            ea <= 0;
            eb <= 0;
            epc4 <= 0;
            ern0 <= 0;
            esa <= 0;
        end
        else
        begin
            ewreg <= dwreg;
            em2reg <= dm2reg;
            ewmem <= dwmem;
            ealuimm <= daluimm;
            eshift <= dshift;
            ejal <= djal;
            ealuc <= daluc;
            eimm <= dimm;
            ea <= da;
            eb <= db;
            epc4 <= dpc4;
            ern0 <= drn;
            esa <= dsa;
        end
    end
endmodule
```

2.3 EXE阶段

pipeexe.v模块主要包含一个alu单元和几个多路选择器。这一部分需要注意的是ern = ern0 | {5{ejal}}这一句，如果产生了jal语法，则将原来的pc保存到31号寄存器中再跳转。alu模块和实验二一样，不再赘述，多路选择器的信号选择也是根据原理图来做的。

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

    input [3:0] ealuc;
    input [31:0] ea, eb, eimm, epc4, esa;
    input [4:0] ern0;
    input ealuimm, eshift, ejal;
    output [31:0] ealu;
    output [4:0] ern;
    wire [31:0] a, b, r;
    alu alu_unit(a, b, ealuc, r);
    wire [4:0] ern = ern0 | {5{ejal}};
    mux2x32 alu_a(ea, esa, eshift, a);
    mux2x32 alu_b(eb, eimm, ealuimm, b);
    mux2x32 link(r, epc4, ejal, ealu);

endmodule
```

pipeemreg.v模块就是在resetn不为0的情况下，将信号传递到下一级。

```
// pipeemreg.v
module pipeemreg(ewreg, em2reg, ewmem, ealu, eb, ern, clock, resetn, mwreg,
mm2reg, mwmem, malu, mb, mrn);
    input ewreg, em2reg, ewmem, clock, resetn;
    input [31:0] ealu, eb;
    input [4:0] ern;
    output mwreg, mm2reg, mwmem;
    output [31:0] malu, mb;
    output [4:0] mrn;
    reg mwreg, mm2reg, mwmem;
    reg [31:0] malu, mb;
    reg [4:0] mrn;

    always @( posedge clock or negedge resetn)
    begin
        if (resetn == 0)
        begin
            mwreg <= 0;
            mm2reg <= 0;
            mwmem <= 0;
            malu <= 0;
            mb <= 0;
            mrn <= 0;
        end
        else
        begin
            mwreg <= ewreg;
            mm2reg <= em2reg;
            mwmem <= ewmem;
            malu <= ealu;
            mb <= eb;
            mrn <= ern;
        end
    end
endmodule
```

2.4 MEM阶段

这一部分非常类似实验二中IO端口设计中的操作，malu[7]决定了是向data memory中写入数据还是向IO中写入数据。此外例化的io_input_reg.v和io_outputreg.v作用是建立IO与CPU之间的通路实现输入输出。

```
// pipemem.v
module pipemem(mwmem, malu, mb, clock, mem_clock, mmo, resetn,
in_port0, in_port1, out_port0, out_port1, out_port2, out_port3,
write_datamem_enable);
    input mwmem, clock, resetn;
    input [31:0] malu, mb;
    input mem_clock;
    input [31:0] in_port0, in_port1;

    output [31:0] mmo, out_port0, out_port1, out_port2, out_port3;
    //wire write_datamem_enable = mwmem;
```

```
    wire [31:0] mem_dataout, io_read_data;
    wire write_enable, write_io_enable;
    output write_datamem_enable;

    assign write_enable = mwmem;
    assign write_io_enable = malu[7] & write_enable;
    assign write_datamem_enable = ~malu[7] & write_enable;

    //sc_datamem dmem(resetn, malu, mb, mmo, mwmem, mem_clock,
    //in_port0, in_port1, out_port0, out_port1, out_port2, out_port3);
    //resetn,addr,datain,dataout,we,clock,
    //in_port0,in_port1,out_port0,out_port1,out_port2,out_port3;
    mux2x32 mem_io_dataout_mux(mem_dataout, io_read_data, malu[7], mmo);
    lpm_ram_dq_dram dram(malu[6:2], mem_clock, mb, write_datamem_enable,
mem_dataout);
    io_output_reg io_output_seg(malu, mb, write_io_enable, mem_clock, resetn,
    out_port0, out_port1, out_port2, out_port3);
    io_input_reg io_input_seg(malu, mem_clock, io_read_data, in_port0,
in_port1);

endmodule
```

```
// io_input_reg.v 和 io_input_mux.v
module io_input_reg(addr, io_clk, io_read_data, in_port0, in_port1);
//inport:  外部直接输入进入ioreg
    input [31:0] addr;
    input io_clk;
    input [31:0] in_port0,in_port1;
    output [31:0] io_read_data;

    reg [31:0] in_reg0;     // input port0
    reg [31:0] in_reg1;     // input port1


    io_input_mux io_imput_mux2x32(in_reg0,in_reg1,addr[7:2],io_read_data);

    always @(posedge io_clk)
    begin
        in_reg0 <= in_port0;   // 输入端口在 io_clk 上升沿时进行数据锁存
        in_reg1 <= in_port1;   // 输入端口在 io_clk 上升沿时进行数据锁存
            // more ports，可根据需要设计更多的输入端口。

    end
endmodule



module io_input_mux(a0, a1, sel_addr, y);
    input [31:0] a0, a1;
    input [5:0] sel_addr;

    output [31:0] y;

    reg [31:0] y;

    always @*
    case (sel_addr)
```

```
        6'b100000: y = a0;
        6'b100001: y = a1;
          default: y = 32'h0;
        // more ports，可根据需要设计更多的端口。
    endcase

endmodule
```

```
// io_output_reg.v
module io_output_reg(addr, datain, write_io_enable, io_clk, resetn, out_port0,
out_port1, out_port2, out_port3);

    input [31:0] addr, datain;
    input write_io_enable, io_clk;
    input resetn;
    //reset signal. if necessary,can use this signal to reset the output to 0.
    output [31:0] out_port0, out_port1, out_port2, out_port3;

    reg [31:0] out_port0, out_port1, out_port2, out_port3;

    always @(posedge io_clk or negedge resetn)
    begin
        if(resetn == 0)
        begin
            out_port0 <= 0;
            out_port1 <= 0;
            out_port2 <= 0;
            out_port3 <= 0;
        end
        else
        begin
            if(write_io_enable == 1)
            case(addr[7:2])
                6'b100000: out_port0 <= datain;//80h
                6'b100001: out_port1 <= datain;//84h
                6'b100010: out_port2 <= datain;//88h
                6'b100011: out_port3 <= datain;//92h
            endcase
        end
    end
endmodule
```

pipemwreg.v作用是在resetn不为0的情况下向WB阶段传送信号。

```
//pipemwreg.v
module pipemwreg(mwreg, mm2reg, mmo, malu, mrn, clock, resetn, wwreg, wm2reg,
wmo, walu, wrn);
    input mwreg, mm2reg, clock, resetn;
    input [31:0] mmo, malu;
    input [4:0] mrn;
    output wwreg, wm2reg;
    output [31:0] wmo, walu;
    output [4:0] wrn;
    reg wwreg, wm2reg;
    reg [31:0] wmo, walu;
```

```
    reg [4:0] wrn;
    always @(posedge clock or negedge resetn)
    begin
        if (resetn == 0)
        begin
            wwreg <= 0;
            wm2reg <= 0;
            wmo <= 0;
            walu <= 0;
            wrn <= 0;
        end
        else
        begin
            wwreg <= mwreg;
            wm2reg <= mm2reg;
            wmo <= mmo;
            walu <= malu;
            wrn <= mrn;
        end
    end
endmodule
```

2.5 WB 阶段

这一部分仅仅使用一个两路32位选择器实现，比较简单，与原理图中一样，不再赘述。

```
// mux2x32.v
module mux2x32 (a0,a1,s,y);

   input [31:0] a0,a1;
   input        s;

   output [31:0] y;

   assign y = s ? a1 : a0;

endmodule
```

四. 仿真结果展示与分析

首先我们使用给定的第一个汇编指令进行信号模拟，这部分没有对IO模块进行测试。mif文件如下所示，，这部分的mif文件的汇编指令中包含了直通、lw数据冒险、jal、jr、j等控制冒险操作，可以通过这些指令验证我们是否正确实现了所有的操作：

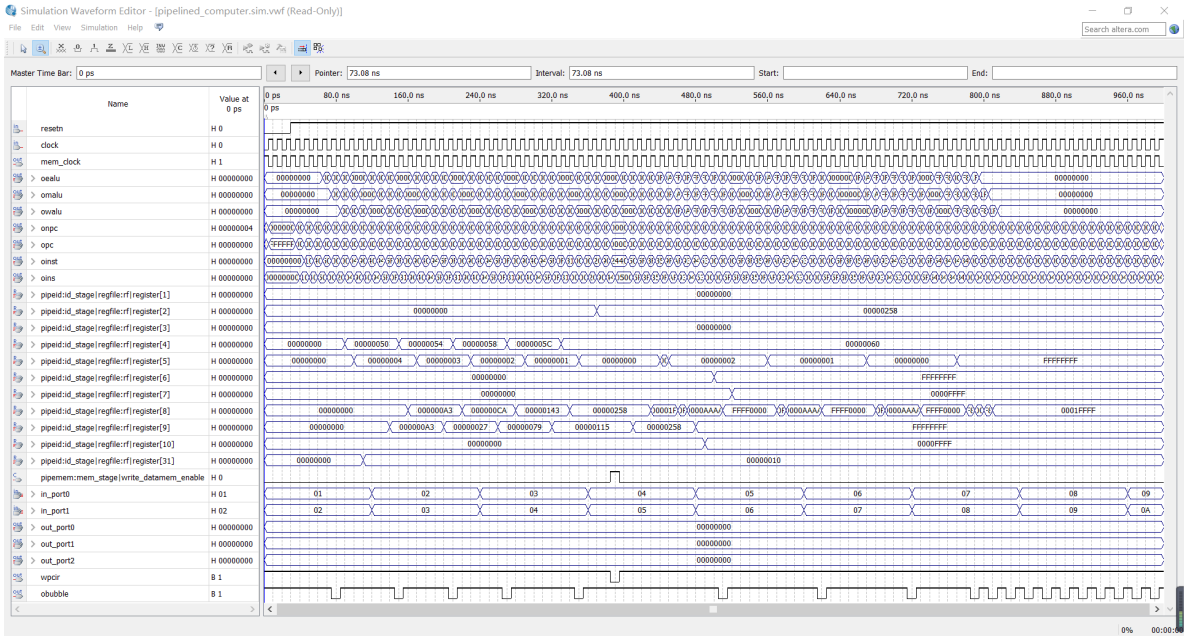
```
DEPTH = 64;           % Memory depth and width are required %
WIDTH = 32;           % Enter a decimal number %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless %
                      % otherwise specified, radixes = HEX %
CONTENT
BEGIN
[0..3F] : 00000000;   % Range--Every address from 0 to 1F = 00000000 %
```

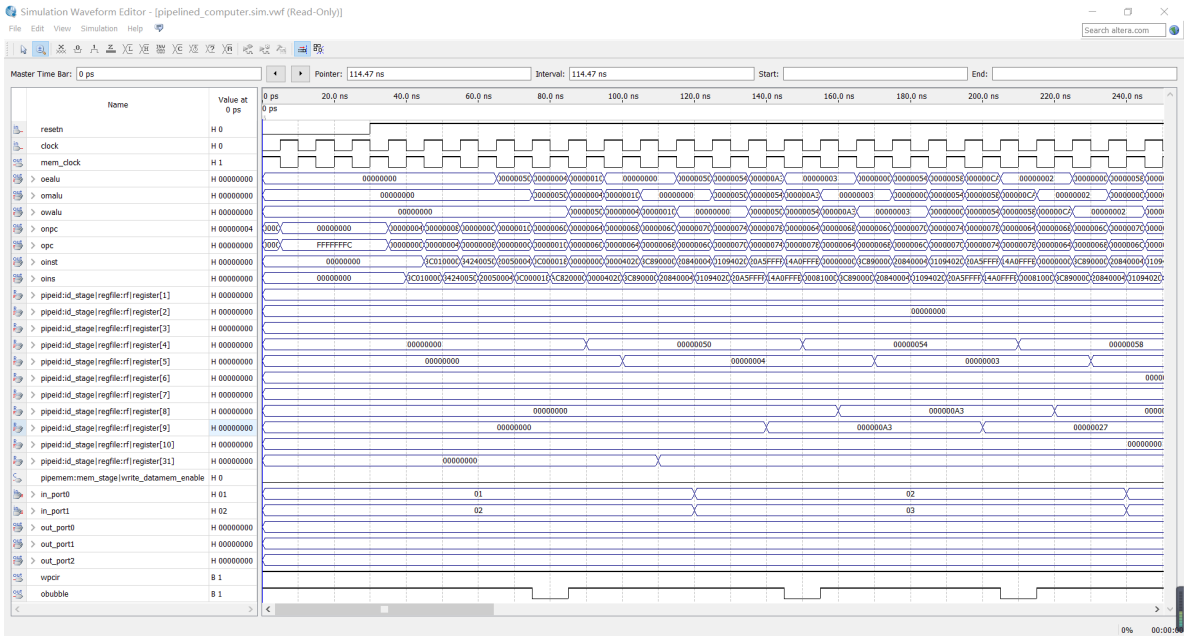
```
 0 : 3c010000;        % (00) main:   lui $1, 0           # address of data[0] %
     %
 1 : 34240050;        % (04)         ori $4, $1, 80       # address of data[0] %
     %
 2 : 20050004;        % (08)         addi $5, $0, 4       # counter %
      %
 3 : 0c000018;        % (0c) call:   jal sum              # call function %
      %
 4 : ac820000;        % (10)         sw $2, 0($4)         # store result %
     %
 5 : 8c890000;        % (14)         lw $9, 0($4)         # check sw %
     %
 6 : 01244022;        % (18)         sub $8, $9, $4       # sub: $8 <- $9 - $4 %
     %
 7 : 20050003;        % (1c)         addi $5, $0, 3       # counter %
      %
 8 : 20a5ffff;        % (20) loop2:  addi $5, $5, -1      # counter - 1 %
      %
 9 : 34a8ffff;        % (24)         ori $8, $5, 0xffff   # zero-extend:
0000ffff %   %
 A : 39085555;        % (28)         xori $8, $8, 0x5555  # zero-extend:
0000aaaa %   %
 B : 2009ffff;        % (2c)         addi $9, $0, -1      # sign-extend:
ffffffff %   %
 C : 312affff;        % (30)         andi $10, $9, 0xffff # zero-extend:
0000ffff %   %
 D : 01493025;        % (34)         or $6, $10, $9       # or: ffffffff %
     %
 E : 01494026;        % (38)         xor $8, $10, $9      # xor: ffff0000 %
      %
 F : 01463824;        % (3c)         and $7, $10, $6      # and: 0000ffff %
      %
10 : 10a00001;        % (40)         beq $5, $0, shift    # if $5 = 0, goto
shift %   %
11 : 08000008;        % (44)         j loop2              # jump loop2 %
     %
12 : 2005ffff;        % (48) shift:  addi $5, $0, -1      # $5 = ffffffff %
      %
13 : 000543c0;        % (4c)         sll $8, $5, 15       # <<15 = ffff8000 %
      %
14 : 00084400;        % (50)         sll $8, $8, 16       # <<16 = 80000000 %
      %
15 : 00084403;        % (54)         sra $8, $8, 16       # >>16 = ffff8000
(arith) % %
16 : 000843c2;        % (58)         srl $8, $8, 15       # >>15 = 0001ffff
(logic) % %
17 : 08000017;        % (5c) finish: j finish             # dead loop %
      %
18 : 00004020;        % (60) sum:    add $8, $0, $0       # sum %
      %
19 : 8c890000;        % (64) loop:   lw $9, 0($4)         # load data %
      %
1A : 20840004;        % (68)         addi $4, $4, 4       # address + 4 %
      %
1B : 01094020;        % (6c)         add $8, $8, $9       # sum %
      %
```

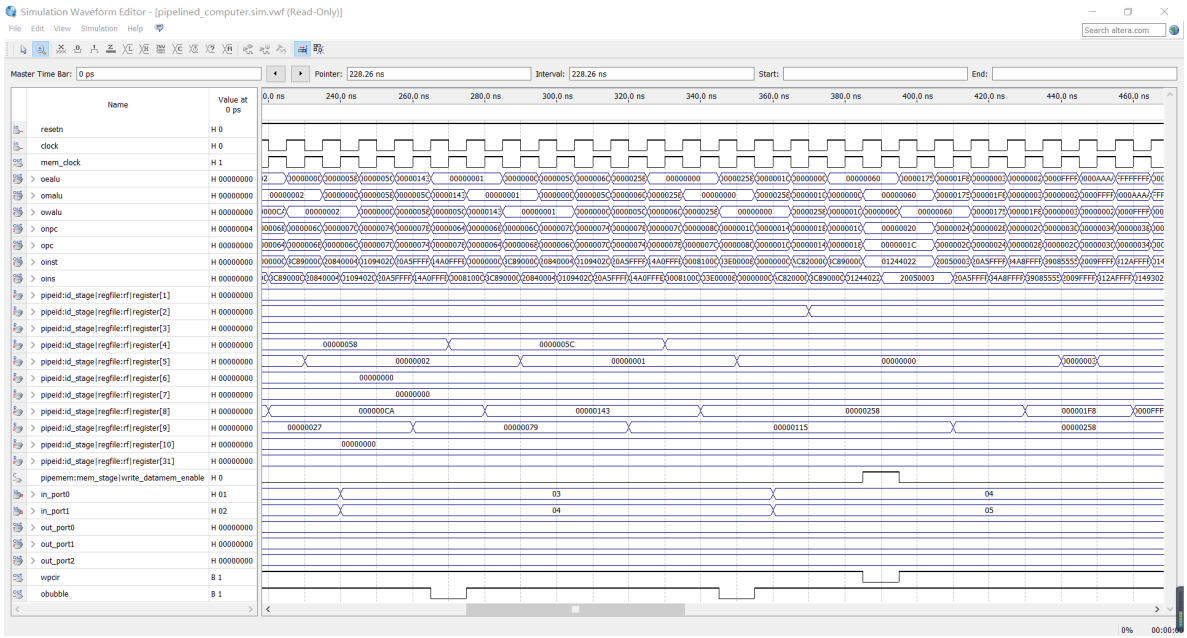
```
1C : 20a5ffff;        % (70)         addi $5, $5, -1      # counter - 1 %
     %
1D : 14a0fffb;        % (74)         bne $5, $0, loop     # finish? %
     %
1E : 00081000;        % (78)         sll $2, $8, 0        # move result to $v0 %
     %
1F : 03e00008;        % (7c)         jr $ra               # return %
     %
END ;
```

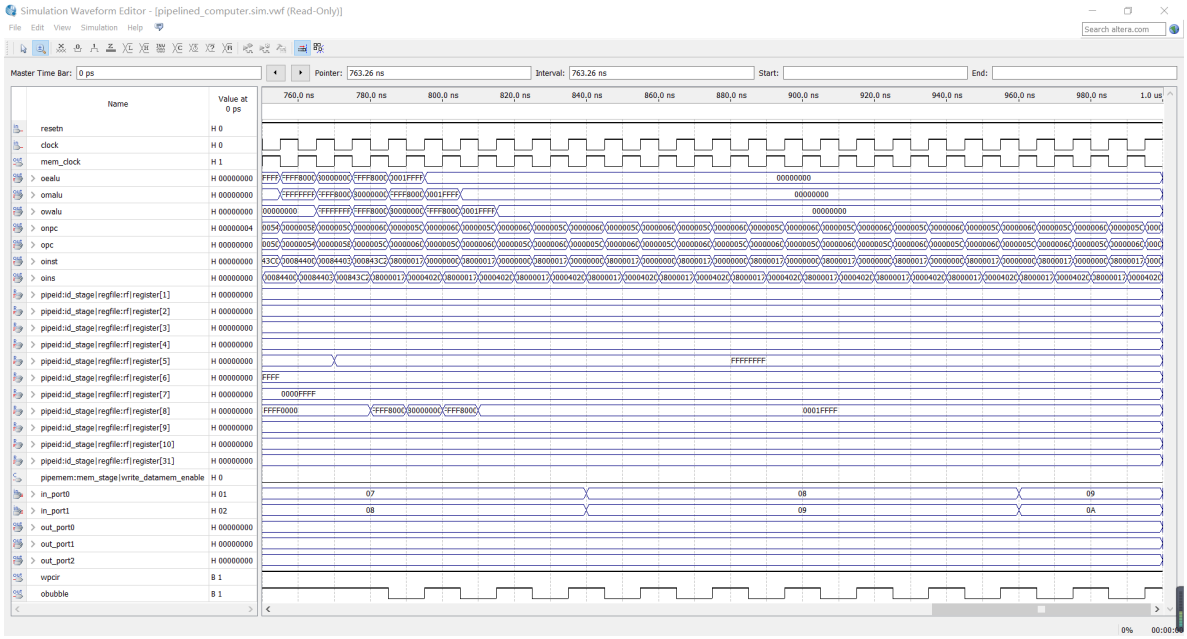
运行pipelined_computer_test_wave_01.vwf产生波形：

先展示全局波形：









可以看到我们的波形与实验指导中所给的波形，无论是寄存器的值还是信号的值都是一致的。

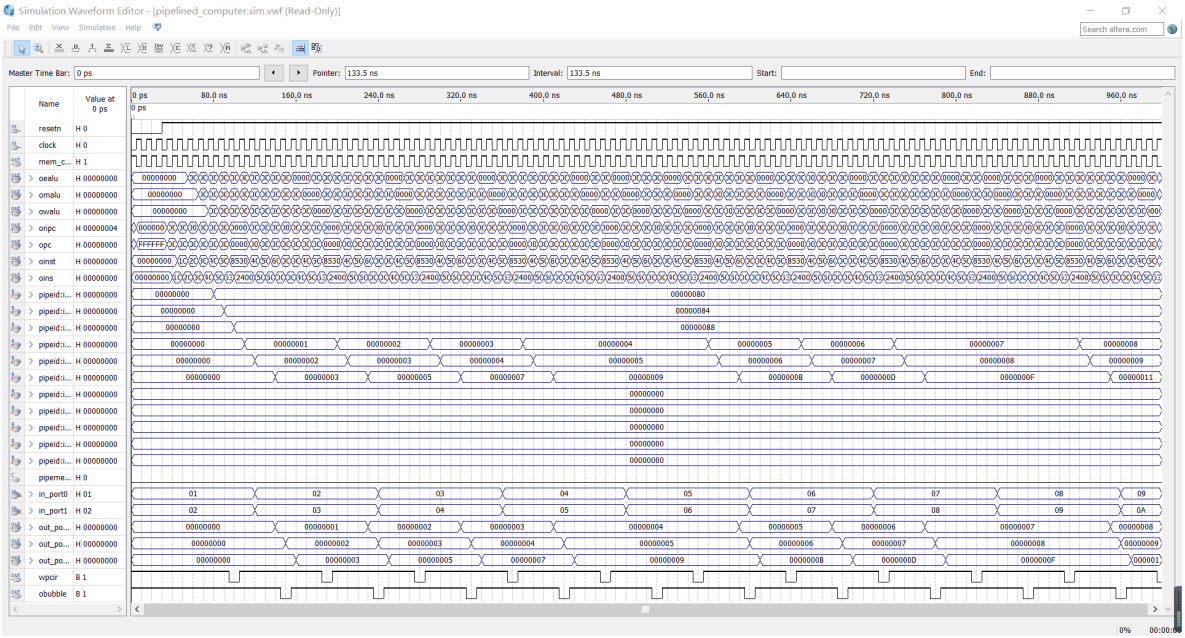
接下来验证IO端口是否正确，使用如下的mif文件，这部分的汇编指令中包含了对IO端口的存值取值循环操作：

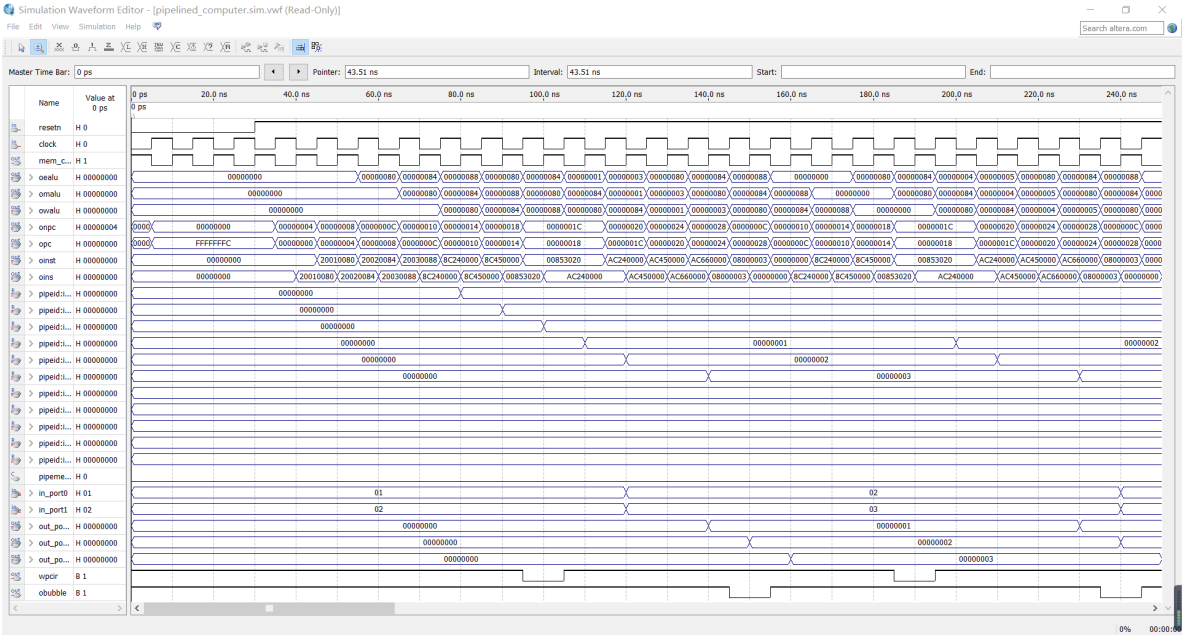
```
DEPTH = 16;           % Memory depth and width are required %
WIDTH = 32;           % Enter a decimal number %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless %
                      % otherwise specified, radixes = HEX %
CONTENT
BEGIN
[0..F] : 00000000;    % Range--Every address from 0 to 1F = 00000000 %

0 : 20010080;         % (00) main: addi $1, $0, 128 # outport0, inport0
   %
1 : 20020084;         % (04)       addi $2, $0, 132 # outport1, inport1
   %
2 : 20030088;         % (08)       addi $3, $0, 136 # outport2
    %
```

```
3 : 8c240000;         % (0c) loop: lw   $4, 0($1)   # input inport0 to $4
   %
4 : 8c450000;         % (10)       lw   $5, 0($2)   # input inport1 to $5
   %
5 : 00853020;         % (14)       add  $6, $4, $5  # add inport0 with inport1 to
$6 %
6 : ac240000;         % (18)       sw   $4, 0($1)   # output inport0 to outport0
    %
7 : ac450000;         % (1c)       sw   $5, 0($2)   # output inport1 to outport1
    %
8 : ac660000;         % (20)       sw   $6, 0($3)   # output result to outport2
   %
9 : 08000003;         % (24)       j loop           #
   %
END ;
```

运行pipelined_computer_test_wave_01.vwf产生波形：

全局波形如下所示：





可以看到in_port和out_port端口的值以及相加的计算都是正确的。由此证明我们的IO端口设计成功。

五. 实验总结和感想

本次实验中，我们从自己动手实践的角度完成了流水线CPU的实验设计，这让我们对流水线CPU的整体结构和运作方式有了更进一步的理解和掌握，同时我们通过对信号的各种判断逻辑，运用了直通、停顿和冲刷技术实现了数据冒险、控制冒险的处理的操作，本次实验中对IO端口的处理让我们在实验二的基础上进一步了解了IO和CPU的交互。本次实验让我收获很多，非常感谢各位老师和助教对我的帮助和指导！