计算机组成实验3

流水线CPU模块设计仿真实验

518030910283 王航宇

一、 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起到的重要作用。
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。

二、 实验要求

1. 完成五级流水线CPU核心模块的设计。
2. 完成对五级流水线CPU的仿真，仿真测试程序应该具有与实验一提供的标准测试程序代码相同的功能。对两种CPU实现核心处理功能的过程和设计处理上的区别作对比分析。
3. 完成流水线CPU的IO模块仿真，对两种CPU实现相同IO功能的过程和设计处理上的区别作对比分析。

三、 实验过程及结果

本实验解决冒险的方法：

数据冒险：lw的数据冒险通过使用wpcir插入气泡解决，此时保持pc_cnt和和IF/ID流水线寄存器的信号不变，同时清空之后的控制信号；其他指令的数据冒险通过直通解决

控制冒险：通过信号ebubble和dbubble解决，此时将流水线flush，把跳转指令下一条指令的控制信号清零。

下面是具体代码解释：

pipepc.v

作用是更新pc，在时钟上升沿，如果复位，pc=-4，如果不复位而且不插入气泡，pc就等于下一个pc（npc），如果插入气泡，pc不变。

wpcir用来解决lw的数据冒险，如果wpcir等于0，说明要控制pc_cnt（程序计数器）和IF/ID流水线寄存器的信号不变，同时清空之后的控制信号

```
module pipepc(npc,wpcir,clock,resetn,pc);//提供下一条指令的pc
    input [31:0] npc;//下一条指令pc
    input        wpcir,clock,resetn;

    output [31:0] pc;
```

```
    reg    [31:0] pc;

    always@(posedge clock or negedge resetn)//时钟上升沿
    begin
        if (resetn == 0)//复位
            begin
                pc <= -4;
            end
        else
            if (wpcir == 1)//不复位且没有气泡
                begin
                    pc <= npc;
                end
    end
endmodule
```

pipeif.v IF流水段，取出指令

根据pcsource的值，更新npc（0:pc4,1:bpc,2:da,3:jpc）。注意这里传给指令ROM的信号是mem_clock（clock的反相），留给信号半个节拍的传输时间。

```
module pipeif(pcsource,pc,bpc,da,jpc,npc,pc4,ins,mem_clock);
            //pcsource决定npc的值从哪里来，bpc是beq和bne的pc，da是jr的pc，jpc是j和jal
的pc
    input [1:0] pcsource;
    input [31:0] pc,bpc,da,jpc;
    input        mem_clock;

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

    assign pc4 = pc + 4;

    mux4x32 nextpc(pc4,bpc,da,jpc,pcsource,npc);//0:pc4,1:bpc,2:da,3:jpc

    sc_instmem imem(pc,ins,mem_clock);//取出指令
endmodule
```

sc_instmem.v

因为此时输入的时钟信号与单周期不同，所以需要修改sc_instmem.v，用mem_clock作为rom_clock。

```
module sc_instmem (addr,inst,mem_clk);
   input  [31:0] addr;
   input         mem_clk;
   output [31:0] inst;

   lpm_rom_irom irom (addr[7:2],mem_clk,inst);
endmodule
```

pipeir.v IF/ID流水线寄存器

在时钟上升沿时，如果resetn=0，信号清零，如果wpcir=0，信号保持不变，其他情况信号正常传递

```
module pipeir(pc4,ins,wpcir,clock,resetn,dpc4,inst);
//IF/ID寄存器，clock上升沿启动
    input [31:0] pc4,ins;
    input        wpcir,clock,resetn;
    output [31:0] dpc4,inst;//在ID阶段的pc4和指令
    reg [31:0] dpc4,inst;

    always@(posedge clock or negedge resetn)
    begin
        if (resetn == 0)//清零
            begin
                dpc4 <= 0;
                inst <= 0;
            end
        else
            if(wpcir == 1)
                begin
                    dpc4 <= pc4;
                    inst <= ins;
                end
    end
endmodule
```

pipeid.v ID流水段，进行解码和生成控制信号操作

因为beq、bne指令的判断跳转操作被提前到了ID段，所以就不需要mzero和ezero信号了

为了解决控制冒险，在ID段加入了ebubble和dbubble两个信号，用于在跳转指令之后flush，在ID/EXE流水线寄存器dbubble被传递给ebubble。

当pcsource！=00时，dbubble=1，即说明需要进行flush操作，在cu里将控制信号全部清零

为了解决数据冒险，需要分两步，lw的数据冒险需要通过wpcir解决，其他指令的数据冒险需要通过直通解决

```
module pipeid(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/*,mzero,
                    drs,drt*/,ebubble,dbubble);
    input       mwreg,ewreg,em2reg,mm2reg,wwreg,mem_clock,resetn,ebubble;
    input [4:0] mrn,ern,wrn;
    input [31:0] dpc4,inst,wdi,ealu,malu,mmo;

    output [31:0] jpc,bpc,da,db,dimm,dsa;
    output [1:0] pcsource;
    output       wpcir,dwreg,dm2reg,dwmem,daluimm,dshift,djal,dbubble;
    output [3:0] daluc;
    output [4:0] drn;

    wire [31:0] q1,q2,da,db;
```

```
    wire [1:0]  fwda,fwdb;
    wire          z = (da == db);
    wire        regrt,sext;
    wire        e = sext & inst[15];
    wire [15:0] imm = {16{e}};
    wire [31:0] dimm = {imm,inst[15:0]};
    wire [31:0] dsa = {27'b0,inst[10:6]};     //sa部分
    wire [31:0] offset = {imm[13:0],inst[15:0],2'b0};
    wire [31:0] jpc = {dpc4[31:28],inst[25:0],2'b0};
    wire [31:0] bpc = dpc4 + offset;

    wire dbubble = (pcsource[1:0] != 2'b0);

    regfile rf(inst[25:21],inst[20:16],wdi,wrn,wwreg,mem_clock,resetn,q1,q2);
    mux4x32 da_mux(q1,ealu,malu,mmo,fwda,da);
    mux4x32 db_mux(q2,ealu,malu,mmo,fwdb,db);
    mux2x5  rn_mux(inst[15:11],inst[20:16],regrt,drn);
    sc_cu   cu(inst[31:26],inst[5:0],z,dwmem,dwreg,regrt,dm2reg,daluc,dshift,
                    daluimm,pcsource,djal,sext,fwda,fwdb,wpcir,inst[25:21],
                    inst[20:16],mrn,mm2reg,mwreg,ern,em2reg,ewreg,ebubble);
endmodule
```

sc_cu.v　CU模块

在cu模块里，在原本的基础上，还需要实现wpcir的清零控制信号、ebubble的flush和直通三个功能

根据逻辑，可以写出wpcir的逻辑表达式 assign wpcir = ~(em2reg & (ern == rs | ern == rt));

发现ebubble在cu中需要做的功能与wpcir类似，都是清空控制信号，所以将wpcir和ebubble合并成control，control=0时，清空信号

直通fwda和fwdb的逻辑表达式课上已经讲过，根据逻辑关系写出代码即可

```
module sc_cu (op, func, z, wmem, wreg, regrt, m2reg, aluc, shift,
              aluimm, pcsource, jal,sext, fwda, fwdb, wpcir,
    rs, rt, mrn, mm2reg, mwreg, ern, em2reg, ewreg, ebubble);

    input        z,mm2reg,mwreg, em2reg,ewreg, ebubble;
    input  [5:0] op,func;
    input  [4:0] rs, rt, mrn, ern;

   output       wreg,regrt,jal,m2reg,shift,aluimm,sext,wmem,wpcir;
   output [3:0] aluc;
   output [1:0] pcsource,fwda,fwdb;
   reg [1:0] fwda, fwdb;

   wire r_type = ~|op;
   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_hads = r_type & func[5] & func[4] & func[3] &
                func[2] &  func[1] & func[0];           //111111
```

```
   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

   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
    wire control = wpcir & ~ebubble;//跳转指令的bubble和lw的数据冒险

   assign pcsource[1] = control&(i_jr | i_j | i_jal);
   assign pcsource[0] = control&(( i_beq & z ) | (i_bne & ~z) | i_j | i_jal) ;

   assign wreg =control&(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 | i_hads);

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

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

    always @(*)
    begin
```

```
        if (ewreg & ~ em2reg & (ern != 0) & (ern == rs) ) //将上一条指令的alu结果直
通
            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;
    end

    always@(*)
    begin
        if (ewreg & ~ em2reg &(ern != 0) & (ern == rt) )
            fwdb <= 2'b01;
      else
            if (mwreg & ~ mm2reg & (mrn != 0) & (mrn == rt) )
                fwdb <= 2'b10;
            else
                if (mwreg & mm2reg & (mrn != 0) & (mrn == rt) )
                    fwdb <= 2'b11;
                else
                    fwdb <= 2'b00;
    end

endmodule
```

pipedereg.v ID/EXE流水线寄存器

在这里就只需要实现控制信号传递功能，其中添加了bubble信号的传递

```
module pipedereg(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);

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

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

    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;
            ers <= 0;
            ert <= 0;
            esa <= 0;
            ebubble <= 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;
            ers <= drs;
            ert <= drt;
            esa <= dsa;
            ebubble <= dbubble;
        end
    end

endmodule
```

pipeexe.v EXE流水段

进行算术运算，因为没有延迟槽，所以不需要用到pc8

```
module pipeexe(ealuc,ealuimm,ea,eb,eimm,esa,eshift,ern0,
                epc4,ejal,ern,ealu,/*ezero,ert,wrn,wdi,malu,wwreg*/);
    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] alua, alub, alur;
    wire   [31:0] epc8 = epc4 + 4;
    wire   [4:0]  ern = ern0 | {5{ejal}};
    wire          ezero;

    mux2x32 a_mux( ea, esa, eshift, alua );//alu两个计算数的来源
    mux2x32 b_mux( eb, eimm, ealuimm, alub );

    alu     alu_unit( alua, alub, ealuc, alur, ezero);  // alu模块

    mux2x32 ealu_mux( alur, epc4, ejal, ealu );//判断ealu是pc+4(jal)还是alu的计算结果，没有延迟槽

endmodule
```

pipeemreg.v EXE/MEM流水线寄存器

同样的这里只需要执行控制信号传递的功能，由clock和resetn控制

```
module pipeemreg(ewreg,em2reg,ewmem,ealu,eb,ern,/*ezero,*/clock,
            resetn,mwreg,mm2reg,mwmem,malu,mb,mrn/*,mzero*/);

    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
```

pipemem.v　MEM流水段，在RAM中读写操作数

我们用mem_clock作为同步RAM的时钟（即作为ram_clock），留给信号半个节拍的传输时间，在mem_clock上升沿读写

沿用了之前I/O实验的两个I/O输入端口和三个输出端口，用于和I/O交互

```
module pipemem(mwmem,malu,mb/*,clock*/,mem_clock,mmo,resetn,
        real_in_port0,real_in_port1,real_out_port0,
        real_out_port1,real_out_port2);
    input        mwmem/*, clock*/, mem_clock,resetn;
    input [31:0] malu, mb;
    input [31:0] real_in_port0, real_in_port1;
    output [31:0] mmo, real_out_port0, real_out_port1, real_out_port2;
    wire   [31:0] mem_dataout, io_read_data;

    sc_datamem dmem(malu, mb, mmo, mwmem, mem_clock, resetn,
        real_out_port0, real_out_port1, real_out_port2, real_in_port0,
real_in_port1,
        mem_dataout, io_read_data);

endmodule
```

sc_datamem.v

因为时钟信号变成了mem_clock，所以需要稍微修改一下sc_datamem的内容，原来clock的作用由mem_clock代替

```
module sc_datamem (addr,datain,dataout,we,mem_clock,resetn,
        out_port0,out_port1,out_port2,in_port0,in_port1,
        mem_dataout,io_read_data
);

   input  [31:0]  addr;
   input  [31:0]  datain;
    input  [31:0]  in_port0,in_port1;
   input          we, mem_clock,resetn;

   output [31:0]  dataout;
    output [31:0]  out_port0,out_port1,out_port2;
    output [31:0]  mem_dataout,io_read_data;

 /*  wire           dmem_clk;  */
   wire           write_enable;
    wire [31:0]    dataout;
    wire [31:0]    mem_dataout;
    wire           write_data_enable;
    wire           write_io_enable;

   assign         write_enable = we; //可能会有问题
 /*  assign         dmem_clk = mem_clk & ( ~ clock) ; */
```

```
    assign          write_data_enable = write_enable & (~addr[7]);
    assign          write_io_enable = write_enable & addr[7];

   lpm_ram_dq_dram
dram(addr[6:2],mem_clock,datain,write_data_enable,mem_dataout );//从mem中取出
mem_dataout

    io_output io_output_reg(addr,datain, write_io_enable, mem_clock,
out_port0,out_port1,out_port2,resetn);//把数据送到外部设备

    io_input io_input_reg(addr,mem_clock,io_read_data,in_port0,in_port1);//从io中
取出io_read_data

    mux2x32 io_data_mux(mem_dataout,io_read_data,addr[7],dataout);//选择
io_read_data、mem_dataout哪个作为dataout

endmodule
```

io_input和iooutput模块在I/O实验中已经展示过，此次实验没有修改，便不再赘述

pipemwreg.v EXE/MEM流水线寄存器

同样执行传递信号的功能

```
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 ) //清零
        begin
            wwreg <= 0;
            wm2reg <= 0;
            walu <= 0;
            wmo <= 0;
            wrn <= 0;
        end
        else
        begin
            wwreg <= mwreg;
            wm2reg <= mm2reg;
            walu <= malu;
            wmo <= mmo;
```

```
            wrn <= mrn;
        end
    end
endmodule
```

最终的顶层文件pipelined_conputer.v

根据具体需要，删去了一些多余的信号

```
//////////////////////////////////////////////////////////////
//                                                          //
// School of Software of SJTU                               //
//                                                          //
//////////////////////////////////////////////////////////////

module pipelined_computer
(resetn,clock,mem_clock,opc,oinst,oins,oealu,omalu,owalu,onpc,/*da,db,

pcsource*/,in_port0,in_port1,out_port0,out_port1,out_port2,out_port3);
   //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 /*,mem_clock*/;
    output mem_clock;
   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
    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,flush;
   //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，需要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
  /* 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;
   //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);

   pipeif   if_stage(pcsource,pc,bpc,da,jpc,npc,pc4,ins,mem_clock);

   pipeir   inst_reg(pc4,ins,wpcir,clock,resetn,dpc4,inst);

   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/*,mzero,
                     drs,drt,npc*/,ebubble,dbubble);

   pipedereg
de_reg(dbubble,drs,drt,dwreg,dm2reg,dwmem,daluc,daluimm,da,db,dimm,dsa,drn,dshif
t,djal,dpc4,clock,resetn,

ebubble,ers,ert,ewreg,em2reg,ewmem,ealuc,ealuimm,ea,eb,eimm,esa,ern0,eshift,ejal
,epc4);
```

```
   pipeexe
exe_stage(ealuc,ealuimm,ea,eb,eimm,esa,eshift,ern0,epc4,ejal,ern,ealu/*,ezero,er
t,wrn,wdi,malu,wwreg*/);

   pipeemreg
em_reg(ewreg,em2reg,ewmem,ealu,eb,ern/*,ezero*/,clock,resetn,mwreg,mm2reg,mwmem,
malu,mb,mrn/*,mzero*/);

   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*/);

   pipemwreg
mw_reg(mwreg,mm2reg,mmo,malu,mrn,clock,resetn,wwreg,wm2reg,wmo,walu,wrn);

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

endmodule
```

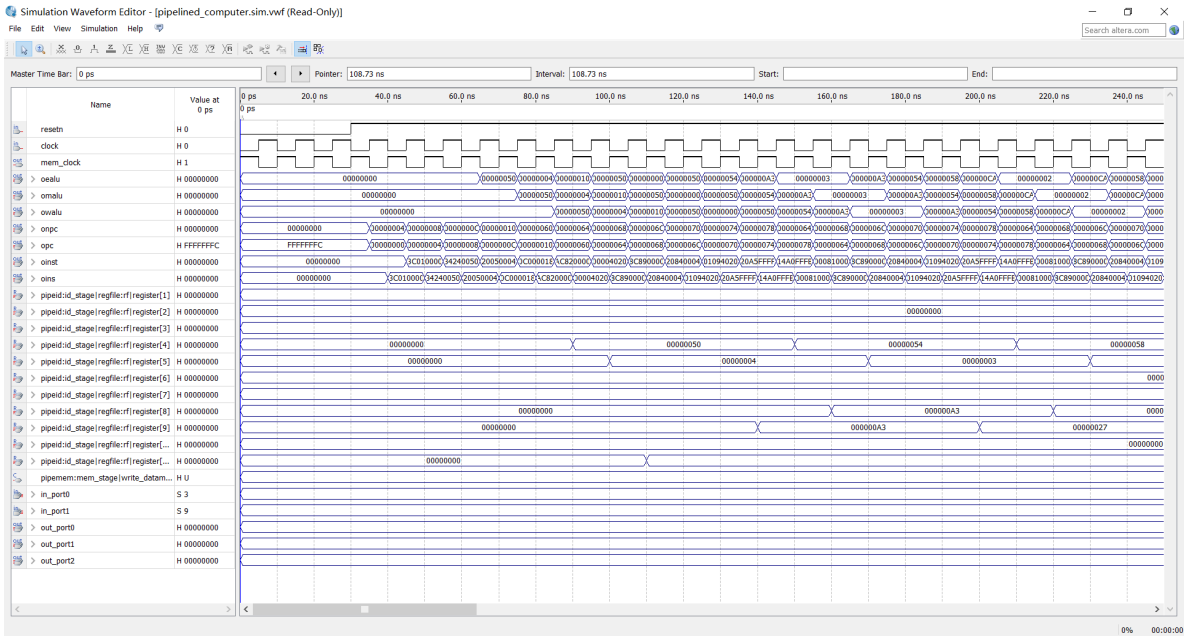
以下为实验结果，此次仿真的mif文件如下

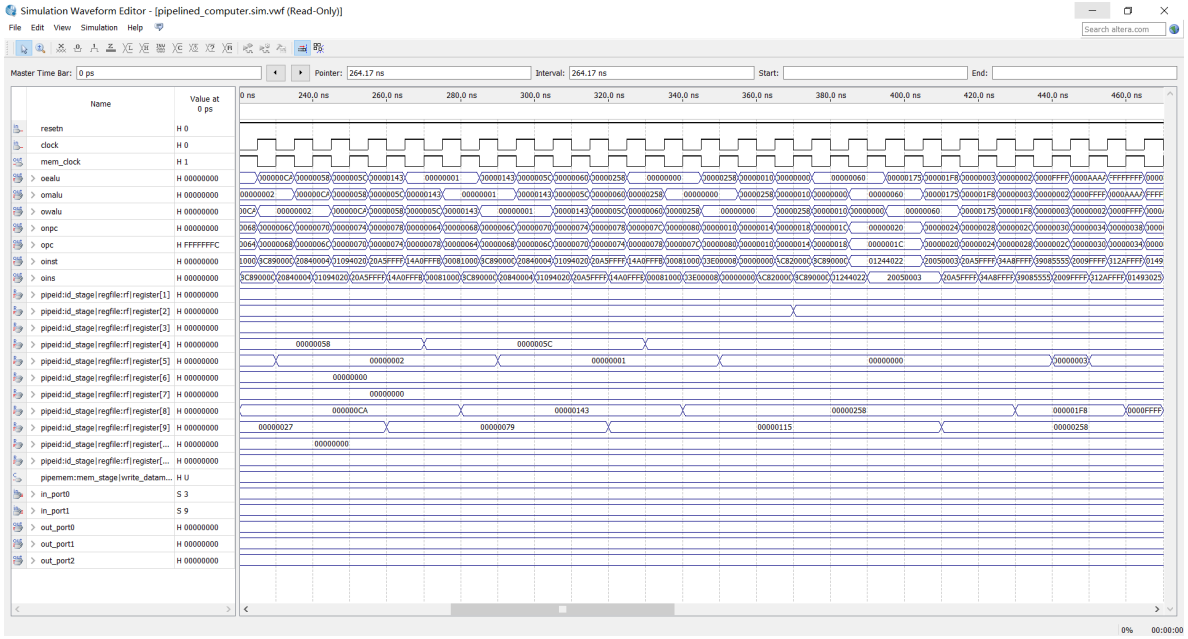
```
DEPTH = 64;           % Memory depth and width are required %
WIDTH = 32;           % Enter a decimal number %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless %
                      % otherwise specified, radixes = HEX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

 0 : 3c010000;        % (00) main:   lui $1, 0           #
      %
 1 : 34240050;        % (04)         ori $4, $1, 80      #
      %
 2 : 20050004;        % (08)         addi $5, $0, 4      #
      %
 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 %   %
```

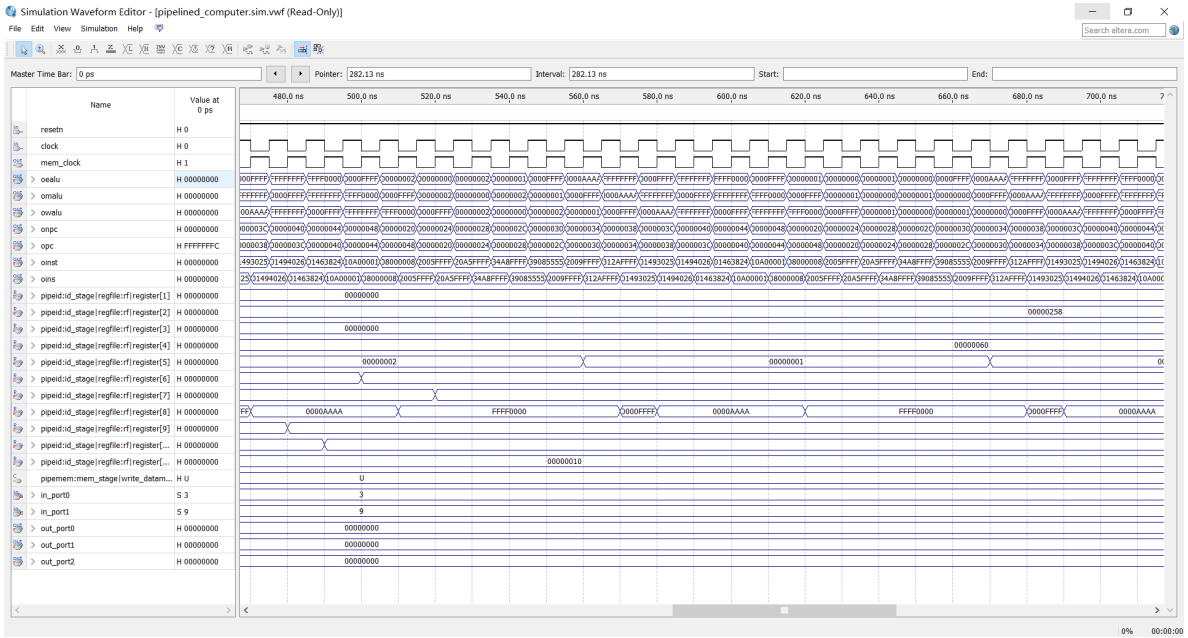
```
 C : 312affff;        % (30)         andi $10, $9, 0xffff # zero-extend:
0000ffff %   %
 D : 01493025;        % (34)         or $6, $10, $9       # or: ffffffff %
     %
 E : 01494026;        % (38)         xor $8, $10, $9      # xor: ffff0000 %
      %
 F : 01463824;        % (3c)         and $7, $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               #
      %
END ;
```

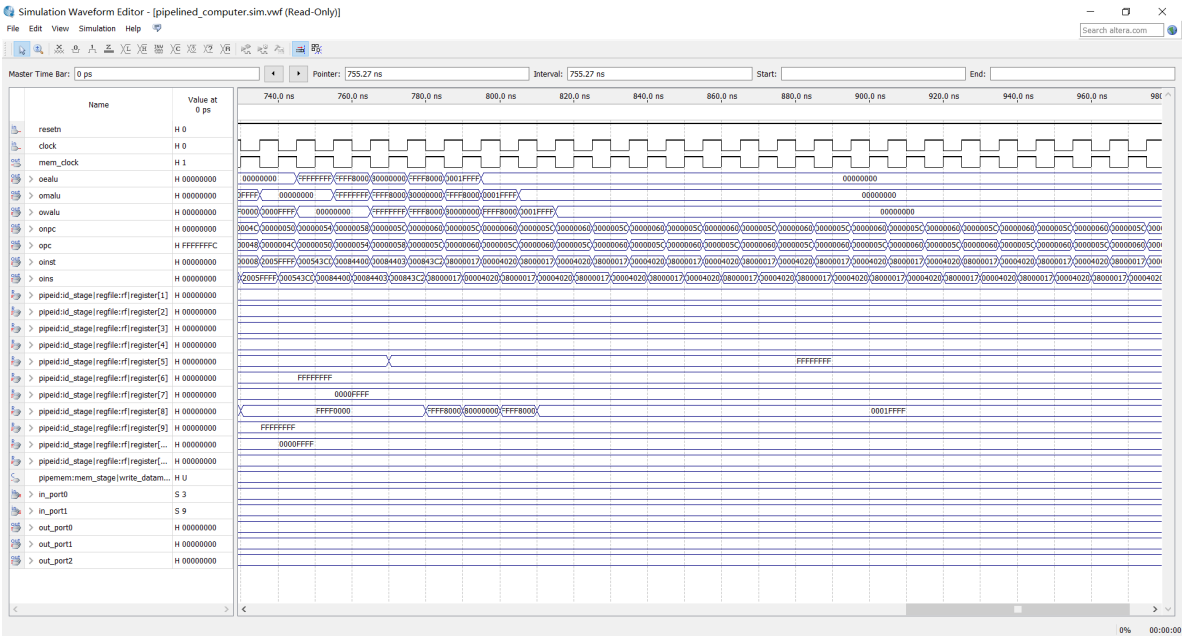
波形图如下

从图中看出，reset变高之后，第一个clock上升沿35ns处，opc信号从0开始，在40ns处clock下降沿取指令oins。从第一个指令开始，在后续每个clock上升沿时，每个时钟节拍进行一次指令处理。每条指令要经过流水线五个节拍后出结果


Simulation Waveform Editor - [pipelined_computer.sim.vwf (Read-Only)]
File Edit View Simulation Help
Search altera.com
Master Time Bar: 0 ps
Pointer: 108.73 ns
Interval: 108.73 ns
Start:
End:
Name
Value at 0 ps
resetn
clock
mem_clock
oealu
omalu
owalu
onpc
opc
oinst
oins
pipeid:id_stage|regfile:rf|register[1]
pipeid:id_stage|regfile:rf|register[2]
pipeid:id_stage|regfile:rf|register[3]
pipeid:id_stage|regfile:rf|register[4]
pipeid:id_stage|regfile:rf|register[5]
pipeid:id_stage|regfile:rf|register[6]
pipeid:id_stage|regfile:rf|register[7]
pipeid:id_stage|regfile:rf|register[8]
pipeid:id_stage|regfile:rf|register[9]
pipeid:id_stage|regfile:rf|register[...
pipeid:id_stage|regfile:rf|register[...
pipemem:mem_stage|write_datam...
in_port0
in_port1
out_port0
out_port1
out_port2
0 ps
20.0 ns
40.0 ns
60.0 ns
80.0 ns
100.0 ns
120.0 ns
140.0 ns
160.0 ns
180.0 ns
200.0 ns
220.0 ns
240.0 ns
0%
00:00:00


Simulation Waveform Editor - [pipelined_computer.sim.vwf (Read-Only)]
File Edit View Simulation Help
Search altera.com
Master Time Bar: 0 ps
Pointer: 264.17 ns
Interval: 264.17 ns
Start:
End:
Name
Value at 0 ps
resetn
clock
mem_clock
oealu
omalu
owalu
onpc
opc
oinst
oins
pipeid:id_stage|regfile:rf|register[1]
pipeid:id_stage|regfile:rf|register[2]
pipeid:id_stage|regfile:rf|register[3]
pipeid:id_stage|regfile:rf|register[4]
pipeid:id_stage|regfile:rf|register[5]
pipeid:id_stage|regfile:rf|register[6]
pipeid:id_stage|regfile:rf|register[7]
pipeid:id_stage|regfile:rf|register[8]
pipeid:id_stage|regfile:rf|register[9]
pipeid:id_stage|regfile:rf|register[...
pipeid:id_stage|regfile:rf|register[...
pipemem:mem_stage|write_datam...
in_port0
in_port1
out_port0
out_port1
out_port2
0 ns
240.0 ns
260.0 ns
280.0 ns
300.0 ns
320.0 ns
340.0 ns
360.0 ns
380.0 ns
400.0 ns
420.0 ns
440.0 ns
460.0 ns
0%
00:00:00


Simulation Waveform Editor - [pipelined_computer.sim.vwf (Read-Only)]
File Edit View Simulation Help
Search altera.com
Master Time Bar: 0 ps
Pointer: 282.13 ns
Interval: 282.13 ns
Start:
End:
Name
Value at 0 ps
resetn
clock
mem_clock
oealu
omalu
owalu
onpc
opc
oinst
oins
pipeid:id_stage|regfile:rf|register[1]
pipeid:id_stage|regfile:rf|register[2]
pipeid:id_stage|regfile:rf|register[3]
pipeid:id_stage|regfile:rf|register[4]
pipeid:id_stage|regfile:rf|register[5]
pipeid:id_stage|regfile:rf|register[6]
pipeid:id_stage|regfile:rf|register[7]
pipeid:id_stage|regfile:rf|register[8]
pipeid:id_stage|regfile:rf|register[9]
pipeid:id_stage|regfile:rf|register[...
pipeid:id_stage|regfile:rf|register[...
pipemem:mem_stage|write_datam...
in_port0
in_port1
out_port0
out_port1
out_port2
480.0 ns
500.0 ns
520.0 ns
540.0 ns
560.0 ns
580.0 ns
600.0 ns
620.0 ns
640.0 ns
660.0 ns
680.0 ns
700.0 ns
0%
00:00:00



从波形图780ns可以看出，R8此时变成了FFFF8000，通过在跳转指令后添加bubble的操作，成功避免了跳转指令引起的流水线填充问题

最后R8得到正确的值0001FFFF

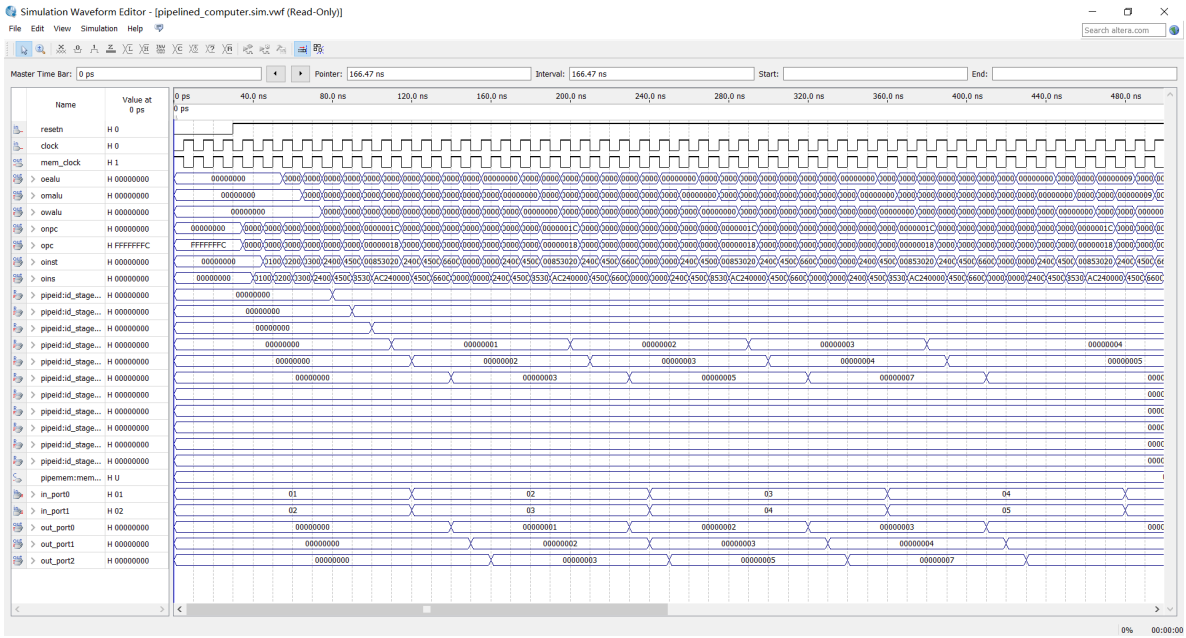
四、 流水线IO实验

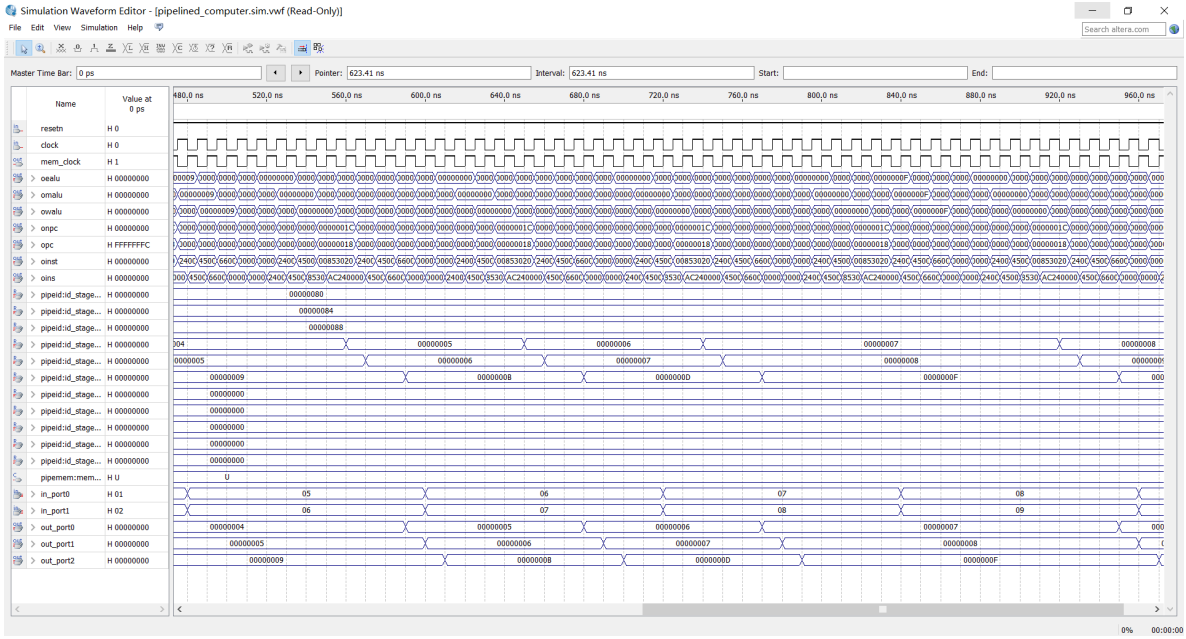
以下是实验二的IO测试程序代码，mif文件如下

```
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 #  %
1 : 20020084;        % (04)       addi $2,$0,132 #  %
2 : 20030088;        % (08)       addi $3,$0,136 #  %
3 : 8c240000;        % (0c) loop: lw $4,0($1)    #  %
4 : 8c450000;        % (10)       lw $5,0($2)    #  %
5 : 00853020;        % (14)       add $6,$4,$5   #  %
6 : ac240000;        % (18)       sw $4,0($1)    #  %
7 : ac450000;        % (1c)       sw $5,0($2)    #  %
8 : ac660000;        % (20)       sw $6,0($3)    #  %
9 : 08000003;        % (24)       j loop         #  %
END ;
```

波形图如下

观察波形图可以发现，第一条指令从40ns开始执行，经过流水线五个clock后R1改变，110ns时in_port0的值被读入到R4，120ns时in_port1的值被读入到R5，130ns没有动作，140ns才将求和结果赋给R6。这是因为流水线CPU为lw的数据冒险采取的bubble操作。160ns时R6的求和结果被输出到out_port2端口。





五、实验收获感想

这次实验我基本实现了实验目标，达到了实验目的。初步利用了Verilog硬件描述语言和仿真软件，设计实现了一个五段流水线CPU，并进行了IO扩展。

流水线CPU和单周期最大的不同就是需要考虑冒险，特别是数据冒险和控制冒险。对于数据冒险，使用直通和插入气泡可以完美解决，直通中fwda和fwdb的逻辑表达式和如何实现插入气泡是难点；对于控制冒险，其实没有完美的解决方法，要么是用转移延迟槽要么是冲刷流水线，这次实验我用的是冲刷流水线的方法。