

# 五段流水线CPU设计

## 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、控制器在流水工作模式下，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。

## 实验特殊说明

1. 本次实验的 *beq* 和 *bne* 跳转在 *ID* 阶段完成，因此移除了顶层设计中的 *zero* 信号。
2. 本次实验没有采用转移延迟槽的设计而实采用 *bubble* 信号解决 *ben* 等跳转信号产生的控制冒险，因此本次实验采用的 *insmem.mif* 中跳转语句的后方无需插入 *sll* 语句进行停顿。
3. 本次实验 *wpcir* 信号位高电平有效。

## 实验思路和代码实现

### *pipepc* 实现

*pipepc* 的作用是获取下一条即将执行的指令的 *pc* 值。该模块由时钟驱动，在时钟的上升沿处更新 *pc* 值。在 *reset* 信号出现时，将 *pc* 值设为无效的 -4，同时方便在一个周期后进入重新从 *pc* = 0 处读取指令开始执行。同时 *wpcir* 信号决定是否对 *pc* 的值进行更新，以解决由 *lw* 指令造成的数据冒险。

```
1 module pipepc(newpc, wpcir, clock, resetn, pc);
2     input [31:0] newpc;
3     input resetn, wpcir, clock;
4     output reg [31:0] pc;
5     always @(posedge clock or negedge resetn)
6     begin
7         if(resetn == 0)
8         begin
9             pc <= -4;
10        end
11    else
12        if(wpcir != 0)
13        begin
14            pc <= newpc;
15        end
16    end
17 endmodule
```

### *pipeif* 实现

*pipeif* 阶段实现的是根据 *pc* 值从指令ROM中取出指令。同时该阶段还将定义一个32位四选一多路器，根据控制单元给出的 *pcsource* 决定下一条指令 *pc* 值的来源。*bpc* 是 *beq* 或 *bne* 指令跳转的下一条指令地址；*da* 来自寄存器，用来解决 *jr* 指令的跳转，*jpc* 用来解决 *j* 指令和 *jal* 指令带来的跳转，而 *pc4* 则是无跳转时按照顺序执行的下一条语句。

```
1 module pipeif(pcsource,pc,bpc,da,jpc,npc,pc4,ins,mem_clock);
2     input mem_clock;
3     input [ 1:0] pcsource;
4     input [31:0] pc, bpc, da, jpc;
5     output wire [31:0] npc, pc4, ins;
6
7     assign pc4 = pc + 4;
8     mux4x32 nextpc(pc4, bpc, da, jpc, pcsource, npc);
9     sc_instmem imem(pc, ins, mem_clock);
10 endmodule
```

## *pipeir* 实现

*pipeir* 是连接 *IF* 和 *ID* 阶段的数据寄存器，合其他寄存器不同，除去清零信号之外，该寄存器还受 *wpcir* 信号的控制，用来在流水线中加入气泡以解决 *lw* 造成的数据冒险。

```
1 module pipeir(pc4,ins,wpcir,clock,resetn, dpc4,inst);
2     input [31:0] pc4, ins;
3     input wpcir, clock, resetn;
4     output reg [31:0] dpc4, inst;
5
6     always @(posedge clock or negedge resetn)
7     begin
8         if(resetn == 0)
9         begin
10             inst <= 0;
11             dpc4 <= 0;
12         end
13         else
14             if(wpcir != 0)
15             begin
16                 inst <= ins;
17                 dpc4 <= pc4;
18             end
19     end
20 endmodule
```

## *pipeid* 实现

*pipeid* 阶段几乎是整个五段流水线中最复杂也最核心的部分，一方面它要完成对寄存器堆的读写操作，另一方面该阶段还要实现控制单元对整个CPU工作信号的生成，包括但不限于后续执行阶段的信号，内存读写信号以及设定旁路直通信号和气泡解决数据冒险和跳转造成的控制冒险。

## 顶层设计

顶层代码结构和单周期CPU设计并无很大差别，尤其是寄存器堆部分并无发生任何改变，因此不做过多赘述。该阶段重点在于控制单元的设计。

```
1 module pipeid(
2     mwreg,mrn,ern,ewreg,em2reg,mm2reg,dpc4,inst,
```

```

3 wrn, wdi, ealu, malu, mmo, wwreg, clock, resetn,
4 bpc, jpc, pcsource, wpcir, dwreg, dm2reg, dwmem, daluc,
5 daluimm, da, db, dimm, drn, dshift, djal, dbubble, ebubble, dsa
6 );
7 input [ 4:0] mrn, ern, wrn;
8 input      ewreg, mwreg, wwreg, mm2reg, em2reg, clock, resetn,
ebubble;
9 input [31:0] inst, wdi, ealu, malu, mmo, dpc4;
10 output [31:0] jpc, bpc, da, db, dimm, dsa;
11 output [ 1:0] pcsource;
12 output      wpcir, dwreg, dm2reg, dbubble, dwmem, daluimm, dshift,
djal;
13 output [ 3:0] daluc;
14 output [ 4:0] drn;
15
16 wire [31:0] q1, q2, da, db;
17 wire [ 1:0] forwarda, forwardb;
18 wire      z = (da == db);
19 wire      regrt, sext;
20 wire      e = sext & inst[15];
21 wire [15:0] imm = {16{e}};           // high 16 sign bit
22 wire [31:0] dimm = {imm, inst[15:0]};
23 wire [31:0] dsa = { 27'b0, inst[10:6] }; // extend to 32 bits from sa
for shift instruction
24 wire [31:0] offset = {imm[13:0], inst[15:0], 1'b0, 1'b0};
25 wire [31:0] bpc = dpc4 + offset;
26 wire [31:0] jpc = {dpc4[31:28], inst[25:0], 1'b0, 1'b0};
27 wire dbubble = (pcsource[1:0] != 2'b00);
28
29 regfile rf( inst[25:21], inst[20:16], wdi, wrn, wwreg, clock, resetn,
q1, q2 );
30 mux4x32 muxda(q1, ealu, malu, mmo, forwarda, da);
31 mux4x32 muxdb(q2, ealu, malu, mmo, forwardb, db);
32 mux2x5 muxrn(inst[15:11], inst[20:16], regrt, drn);
33 sc_cu cu_inst(inst[31:26], inst[5:0], z, dwmem, dwreg, regrt, dm2reg,
daluc, dshift,
34 daluimm, pcsource, djal, sext, forwarda, forwardb, wpcir,
35 inst[25:21], inst[20:16], mrn, mm2reg, mwreg, ern, em2reg,
ewreg, ebubble);
36
37
38 endmodule

```

## 控制单元设计

和单周期CPU相同的是，控制单元仍然承担着解码并且进一步生成控制信号的作用。与单周期不同的是控制单元在流水线设计中要多出三个信号：

1. *wpcir* 信号，该信号用来解决 *lw* 造成的数据冒险，当这种情况发生时，CPU应当停止流水线上这条指令的操作（控制信号置零），插入一个气泡，同时阻止 *pc* 和 的更新迭代，直到下一个周期这条指令可以获得正确的操作数据。
2. *forward* 信号，这些信号用来处理除了 *lw* 之外其他造成的数据冒险，旁路直通的设计和原理在理论课程上已经完备叙述过，此处不进行过多赘述。
3. *bubble* 信号，这条信号解决由跳转产生的控制冒险问题。本次CPU设计未采用延迟槽设计，因此我们需要避免紧跟在跳转指令之后那条指令的执行，即当跳转发生时，我们应该对流水线进行冲刷，将跳转指令的下一条指令所有控制信号置零。

```

1  module sc_cu (
2      op, func, z, wmem, wreg, regrt, m2reg, aluc, shift,
3      aluimm, pcsource, jal, sext, forwarda, forwardb, wpcir,
4      rs, rt, mrn, mm2reg, mwreg, ern, em2reg, ewreg, ebubble
5  );
6      input mwreg, ewreg, mm2reg, em2reg, ebubble;
7      input [4:0] rs, rt, mrn, ern;
8      input [5:0] op, func;
9      input      z;
10     output      wreg, regrt, jal, m2reg, shift, aluimm, sext, wmem, wpcir;
11     output [3:0] aluc;
12     output [1:0] pcsource, forwarda, forwardb;
13     reg [1:0] forwarda, forwardb;
14
15     wire r_type = ~|op;
16     wire i_add = r_type & func[5] & ~func[4] & ~func[3] &
17                 ~func[2] & ~func[1] & ~func[0];          //100000
18     wire i_sub = r_type & func[5] & ~func[4] & ~func[3] &
19                 ~func[2] & func[1] & ~func[0];          //100010
20
21     // please complete the deleted code.
22
23     wire i_and = r_type & func[5] & ~func[4] & ~func[3] &
24                 func[2] & ~func[1] & ~func[0]; //100100
25     wire i_or  = r_type & func[5] & ~func[4] & ~func[3] &
26                 func[2] & ~func[1] & func[0]; //100101
27
28     wire i_xor = r_type & func[5] & ~func[4] & ~func[3] &
29                 func[2] & func[1] & ~func[0]; //100110
30     wire i_sll = r_type & ~func[5] & ~func[4] & ~func[3] &
31                 ~func[2] & ~func[1] & ~func[0]; //000000
32     wire i_srl = r_type & ~func[5] & ~func[4] & ~func[3] &
33                 ~func[2] & func[1] & ~func[0]; //000010
34     wire i_sra = r_type & ~func[5] & ~func[4] & ~func[3] &
35                 ~func[2] & func[1] & func[0]; //000011
36     wire i_jr  = r_type & ~func[5] & ~func[4] & func[3] &
37                 ~func[2] & ~func[1] & ~func[0]; //001000
38
39     wire i_addi = ~op[5] & ~op[4] & op[3] & ~op[2] & ~op[1] & ~op[0];
40     //001000
41     wire i_andi = ~op[5] & ~op[4] & op[3] & op[2] & ~op[1] & ~op[0];
42     //001100
43
44     wire i_ori  = ~op[5] & ~op[4] & op[3] & op[2] & ~op[1] & op[0];
45     //001101 // complete by yourself.
46     wire i_xori = ~op[5] & ~op[4] & op[3] & op[2] & op[1] & ~op[0];
47     //001110
48     wire i_lw   = op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0];
49     //100011
50     wire i_sw   = op[5] & ~op[4] & op[3] & ~op[2] & op[1] & op[0];
51     //101011
52     wire i_beq  = ~op[5] & ~op[4] & ~op[3] & op[2] & ~op[1] & ~op[0];
53     //000100
54     wire i_bne  = ~op[5] & ~op[4] & ~op[3] & op[2] & ~op[1] & op[0];
55     //000101
56     wire i_lui  = ~op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0];
57     //001111

```

```

49     wire i_j      = ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & ~op[0];
//000010
50     wire i_jal    = ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0];
//000011
51
52     assign wpcir = ~(em2reg & ( ern == rs | ern == rt ));
53     wire ctrlable = wpcir & ~ebubble;
54
55     assign pcsource[1] = i_jr | i_j | i_jal;
56     assign pcsource[0] = ( i_beq & z ) | ( i_bne & ~z ) | i_j | i_jal ;
57
58     assign wreg = ctrlable &( i_add | i_sub | i_and | i_or | i_xor |
59                          i_sll | i_srl | i_sra | i_addi | i_andi |
60                          i_ori | i_xori | i_lw | i_lui | i_jal );
61
62     assign aluc[3] = ctrlable & i_sra ;    // complete by yourself.
63     assign aluc[2] = ctrlable &( i_sub | i_beq | i_bne | i_or | i_ori |
i_lui | i_srl | i_sra );
64     assign aluc[1] = ctrlable &( i_xor | i_xori | i_lui | i_sll | i_srl |
i_sra );
65     assign aluc[0] = ctrlable &( i_and | i_andi | i_or | i_ori | i_sll |
i_srl | i_sra );
66     assign shift   = ctrlable &( i_sll | i_srl | i_sra );
67
68     assign aluimm   = ctrlable &( i_addi | i_andi | i_ori | i_xori | i_lw |
i_sw | i_lui);    // complete by yourself.
69     assign sext     = ctrlable &( i_addi | i_lw | i_sw | i_beq | i_bne);
70     assign wmem      = ctrlable & i_sw;
71     assign m2reg     = ctrlable & i_lw;
72     assign regrt     = ctrlable &( i_addi | i_andi | i_ori | i_xori | i_lw |
i_lui);
73     assign jal       = ctrlable & i_jal;
74
75     always @(*)
76     begin
77         if (ewreg & ~ em2reg & (ern != 0) & (ern == rs) )
78             forwarda <= 2'b01; //exe_alu
79         else
80             if (mwreg & ~ mm2reg & (mrn != 0) & (mrn == rs) )
81                 forwarda <= 2'b10; //mem_alu
82             else
83                 if (mwreg & mm2reg & (mrn != 0) & (mrn == rs) )
84                     forwarda <= 2'b11; // mem_lw
85             else
86                 forwarda <= 2'b00;
87         end
88
89     always@(*)
90     begin
91         if (ewreg & ~ em2reg & (ern != 0) & (ern == rt) )
92             forwardb <= 2'b01;
93         else
94             if (mwreg & ~ mm2reg & (mrn != 0) & (mrn == rt) )
95                 forwardb <= 2'b10;
96             else
97                 if (mwreg & mm2reg & (mrn != 0) & (mrn == rt) )
98                     forwardb <= 2'b11;
99             else

```

```

100         forwardb <= 2'b00; // 无需直通
101     end
102 endmodule

```

## *pipedereg* 实现

该部分由清零信号控制，若是清零信号有效则输出置零，否则在上升沿时将输入赋值给对应输出。

```

1  module
2      pipedereg(dbubble,drs,drt,dwreg,dm2reg,dwmem,daluc,daluimm,da,db,dimm,dsa,d
3      rn,dshift,djal,dpc4,clock,resetn,
4
5      ebubble,ers,ert,ewreg,em2reg,ewmem,ealuc,ealuimm,ea,eb,eimm,esa,ern0,eshift
6      ,ejal,epc4);
7      input dbubble, dwreg, dm2reg, dwmem, daluimm, dshift, djal, clock,
8      resetn;
9      input [3:0] daluc;
10     input [31:0] dimm, da, db, dpc4, dsa;
11     input [4:0] drn, drs, drt;
12     output reg ebubble, ewreg, em2reg, ewmem, ealuimm, eshift, ejal;
13     output reg [3:0] ealuc;
14     output reg [31:0] eimm, ea, eb, epc4, esa;
15     output reg [4:0] ern0, ers, ert;
16
17     always@(negedge resetn or posedge clock)
18     begin
19         if(resetn == 0)
20         begin
21             ebubble <= 0;
22             ewreg <= 0;
23             em2reg <= 0;
24             ewmem <= 0;
25             ealuimm <= 0;
26             eshift <= 0;
27             ejal <= 0;
28             ealuc <= 0;
29             eimm <= 0;
30             ea <= 0;
31             eb <= 0;
32             epc4 <= 0;
33             esa <= 0;
34             ern0 <= 0;
35             ers <= 0;
36             ert <= 0;
37         end
38         else
39         begin
40             ebubble <= dbubble;
41             ewreg <= dwreg;
42             em2reg <= dm2reg;
43             ewmem <= dwmem;
44             ealuimm <= daluimm;
45             eshift <= dshift;
46             ejal <= djal;
47             ealuc <= daluc;
48             eimm <= dimm;
49             ea <= da;

```

```

45         eb <= db;
46         epc4 <= dpc4;
47         esa <= dsa;
48         ern0 <= drn;
49         ers <= drs;
50         ert <= drt;
51     end
52 end
53 endmodule

```

## pipeexe 实现

该模块主要进行CPU内部的算术运算，调用了 *alu* 模块。该模块沿用单周期时的设计，因此不做赘述。

```

1  module alu (a,b,aluc,s,z);
2      input [31:0] a,b;
3      input [3:0] aluc;
4      output [31:0] s;
5      output      z;
6      reg [31:0] s;
7      reg      z;
8      always @ (a or b or aluc)
9          begin
10             // event
11             casex (aluc)
12                 4'bx000: s = a + b;           //x000 ADD
13                 4'bx100: s = a - b;           //x100 SUB
14                 4'bx001: s = a & b;           //x001 AND
15                 4'bx101: s = a | b;           //x101 OR
16                 4'bx010: s = a ^ b;           //x010 XOR
17                 4'bx110: s = b << 16;         //x110 LUI: imm << 16bit
18
19                 4'b0011: s = $signed(b) << a; //0011 SLL: rd <- (rt << sa)
20                 4'b0111: s = $signed(b) >> a; //0111 SRL: rd <- (rt >> sa)
21             (logical)
22                 4'b1111: s = $signed(b) >>> a; //1111 SRA: rd <- (rt >> sa)
23             (arithmetic)
24             default: s = 0;
25         endcase
26         if (s == 0 ) z = 1;
27         else z = 0;
28     end
29 endmodule

```

值得注意的是，本次CPU没有采用延迟槽的设计结构，因此 *jal* 在回跳时从 \$31 寄存器里读取的地址应该是  $pc + 4$  而不是  $pc + 8$ 。

```

1  module pipeexe( ealuc,ealuimm,ea,eb,eimm, esa,
2      eshift,ern0,epc4,ejal,ern,ealu);
3      input [3:0] ealuc;
4      input [31:0] ea, eb, eimm, esa, epc4;
5      input [4:0] ern0;
6      input ealuimm, eshift, ejal;
7      output [4:0] ern;
8      output [31:0] ealu;
9      wire [31:0] alua, alub, alures;
10     wire [31:0] epc8 = epc4 + 4;

```

```

10     wire [4:0] ern = ern0 | {5{eja1}};
11     wire iszero;
12
13     mux2x32 a_mux(ea, esa, eshift, alua);
14     mux2x32 b_mux(eb, eimm, ealuimm, alub);
15     alu alu_inst(alua, alub, ealuc, alures, iszero);
16     //mux2x32 res_mux(alures, epc8, eja1, ealu);
17     mux2x32 res_mux(alures, epc4, eja1, ealu);
18 endmodule

```

## pipeemreg 实现

该部分由清零信号控制，若是清零信号有效则输出置零，否则在上升沿时将输入赋值给对应输出。

```

1  module pipeemreg(ewreg,em2reg,ewmem,ealu,eb,ern,clock,resetn,
   mwreg,mm2reg,mwmem,malu,mb,mrn);
2      input ewreg, em2reg, ewmem, clock, resetn;
3      input [31:0] ealu, eb;
4      input [4:0] ern;
5      output reg mwreg, mm2reg, mwmem;
6      output reg [31:0] malu, mb;
7      output reg [4:0] mrn;
8
9      always @(posedge clock or negedge resetn)
10     begin
11         if (resetn == 0)
12             begin
13                 mwreg <= 0;
14                 mm2reg <= 0;
15                 mwmem <= 0;
16                 malu <= 0;
17                 mb <= 0;
18                 mrn <= 0;
19             end
20         else
21             begin
22                 mwreg <= ewreg;
23                 mm2reg <= em2reg;
24                 mwmem <= ewmem;
25                 malu <= ealu;
26                 mb <= eb;
27                 mrn <= ern;
28             end
29     end
30 endmodule

```

## pipemem 实现

*pipemem* 的主要功能就是从数据RAM读取数据或将数据写入到数据RAM中，而sc\_datamem沿用实验2中的定义。本次实验定义三个输出端口，分别用来表示两个操作数和运算结果。同时还有两个输入端口，用来从I/O设备读取操作数。由于该部分整体沿用单周期CPU I/O实验中的设计，因此不做赘述。

```

1  module pipemem(
2      resetn,mwmem,malu,mb,clock,mem_clock,mmo,
3      real_in_port0,real_in_port1, real_out_port0, real_out_port1,
   real_out_port2

```



```

4 );
5     input  resetn, mwmem, clock, mem_clock;
6     input  [31:0] malu, mb;
7     input  [31:0] real_in_port0, real_in_port1;
8     output [31:0] mmo, real_out_port0, real_out_port1, real_out_port2;
9     wire   [31:0] mem_dataout, io_read_data;
10    sc_datamem dmem(malu, mb, mmo, mwmem, mem_clock, resetn,
11        real_out_port0, real_out_port1, real_out_port2, real_in_port0,
12        real_in_port1,
13        mem_dataout, io_read_data);
14 endmodule

```

由于本次所有的memory都用由统一的 *memclock* 驱动，因此datamem的内部部分信号和接线要进行修改。

```

1 module sc_datamem (
2     addr, datain, dataout, we, clock, resetn,
3     out_port0, out_port1, out_port2, in_port0, in_port1, mem_dataout, io_read_data
4 ); //添加了 2 个 32 位输入端口, 3 个 32 位输出端口
5     input  [31:0] addr;
6     input  [31:0] datain;
7     input  [31:0] in_port0, in_port1;
8     input  we, clock, resetn;
9     output [31:0] dataout;
10    output [31:0] out_port0, out_port1, out_port2;
11    output [31:0] mem_dataout;
12    output [31:0] io_read_data;
13    wire dmem_clk;
14    wire write_enable;
15    wire [31:0] dataout;
16    wire [31:0] mem_dataout;
17    wire write_data_enable;
18    wire write_io_enable;
19    /*
20    assign write_enable = we & ~clock; //注意
21    */
22    assign write_enable = we;
23    /*
24    assign dmem_clk = mem_clk & ( ~ clock) ; //注意
25    */
26    //100000-111111:I/O ; 000000-011111:data
27    assign write_data_enable = write_enable & (~addr[7]); //注意
28    assign write_io_enable = write_enable & addr[7]; //注意
29    mux2x32 io_data_mux(mem_dataout, io_read_data, addr[7], dataout);
30    //添加子模块, 用于选择输出数据来源于内部数据存储器还是 IO,
31    //module mux2x32 (a0,a1,s,y);
32    // when address[7]=0, means the access is to the datamem.
33    // that is, the address space of datamem is from 000000 to 011111
word(4 bytes)
34    // when address[7]=1, means the access is to the I/O space.
35    // that is, the address space of I/O is from 100000 to 111111 word(4
bytes)
36    lpm_ram_dq_dram dram (addr[6:2], clock, datain, write_data_enable,
mem_dataout );

```

```

37     io_output io_output_reg (addr, datain, write_io_enable, clock,
out_port0, out_port1,out_port2, resetn);
38     //添加子模块, 用于完成对 IO 地址空间的译码, 以及构成 IO 和 CPU 内部之间的数据输出
通道
39     //module
io_output(addr,datain,write_io_enable,io_clk,out_port0,out_port1,out_port2
);
40     io_input io_input_reg(addr, clock, io_read_data, in_port0, in_port1);
41     //添加子模块, 用于完成对 IO 地址空间的译码, 以及构成 IO 和 CPU 内部之间的数据输入
通道
42     //module io_input( addr,io_clk,io_read_data,in_port0,in_port1);
43     endmodule

```

## *pipemwreg* 实现

该部分由清零信号控制, 若是清零信号有效则输出置零, 否则在上升沿时将输入赋值给对应输出。

```

1  module pipemwreg(
2      mwreg,mm2reg,mmo,malu,mrn,clock,resetn,
3      wwreg,wm2reg,wmo,walu,wrn
4  );
5      input mwreg, mm2reg, clock, resetn;
6      input [4:0] mrn;
7      input [31:0] malu, mmo;
8      output reg wwreg, wm2reg;
9      output reg [4:0] wrn;
10     output reg [31:0] walu, wmo;
11
12     always@(posedge clock or negedge resetn)
13     begin
14         if(resetn == 0)
15         begin
16             wwreg <= 0;
17             wm2reg <= 0;
18             wrn <= 0;
19             walu <= 0;
20             wmo <= 0;
21         end
22         else
23         begin
24             wwreg <= mwreg;
25             wm2reg <= mm2reg;
26             wrn <= mrn;
27             walu <= malu;
28             wmo <= mmo;
29         end
30     end
31 endmodule

```

## *pipelined\_computer* 顶层总设计

与原本给出的实验顶层设计有所不同, 本次由于将 *bne* 和 *beq* 语句的跳转判断阶段提早到了 *ID* 阶段, 因此我移除了顶层设计中部分未用到的信号和接线, 同时根据自己的设计, 修改了部分模块的接口。

```

1  ////////////////////////////////////////////
2  //

```

```

3 // School of Software of SJTU //
4 // //
5 ///////////////////////////////////////////////////////////////////
6
7 module pipelined_computer
8 (resetrn,clock,mem_clock,opc,oinst,oins,oealu,omalu,owalu,onpc,/*da,db,
9 pcsource*/,in_port0,in_port1,out_port0,out_port1,out_port2,out_port3
10 );
11 input resetrn,clock/*,mem_clock*/;
12 input [5:0] in_port0,in_port1;
13 output [31:0] out_port0,out_port1,out_port2,out_port3;// output [6:0]
14 out_port0,out_port1,out_port2,out_port3;
15
16 wire [31:0] real_out_port0,real_out_port1,real_out_port2,real_out_port3;
17
18 wire [31:0] real_in_port0 = {26'b000000000000000000000000,in_port0};
19 wire [31:0] real_in_port1 = {26'b000000000000000000000000,in_port1};
20
21 assign out_port0 = real_out_port0[31:0];//assign out_port0 =
22 real_out_port0[6:0];
23 assign out_port1 = real_out_port1[31:0];//assign out_port0 =
24 real_out_port1[6:0];
25 assign out_port2 = real_out_port2[31:0];//assign out_port0 =
26 real_out_port2[6:0];
27 assign out_port3 = real_out_port3[31:0];//assign out_port0 =
28 real_out_port3[6:0];
29
30 output mem_clock;
31
32 assign mem_clock = ~clock;
33
34 wire [31:0] pc,ealu,malu,walu;
35 output [31:0] opc,oealu,omalu,owalu;// for watch
36 assign opc = pc;
37 assign oعالu = ealu;
38 assign omalu = malu ;
39 assign owalu = walu ;
40
41 wire [31:0] bpc,jpc,pc4,npc,ins,inst;
42 output [31:0] onpc,oins,oinst;// for watch
43 assign onpc=npc;
44 assign oins=ins;
45 assign oinst=inst;
46
47 // for test
48
49 wire [31:0] dpc4,da,db,dimm,dsa;
50 wire [31:0] epc4,ea,eb,eimm,esa;
51 wire [31:0] mb,mmo;
52 wire [31:0] wmo,wdi;
53 wire [4:0] ern0,ern,drn,mrn,wrn;
54 wire [4:0] drs,drt,ers,ert;
55 wire [3:0] daluc,ealuc;
56 wire [1:0] pcsource;
57 wire wpcir;
58 wire dwreg,dm2reg,dwmem,daluimm,dshift,djal; //id stage

```

```

54     wire          ewreg,em2reg,ewmem,ealuimm,eshift,ejal; //exe stage
55     wire          mwreg,mm2reg,mwmem; //mem stage
56     wire          wwreg,wm2reg; //wb stage
57     wire          dbubble, ebubble;
58
59
60     //fortest
61
62     pipepc prog_cnt ( npc,wpcir,clock,resetn,pc );
63
64     pipeif if_stage ( pcsource,pc,bpc,da,jpc,npc,pc4,ins,mem_clock ); // IF
stage
65
66     pipeir inst_reg ( pc4,ins,wpcir,clock,resetn,dpc4,inst ); // IF/ID流水线
寄存器
67
68     pipeid id_stage ( mwreg,mrn,ern,ewreg,em2reg,mm2reg,dpc4,inst,
69 wrn,wdi,ealu,malu,mmo,wwreg,mem_clock,resetn,
70 bpc,jpc,pcsource,wpcir,dwreg,dm2reg,dwmem,daluc,
71 daluimm,da,db,dimm,drn,dshift,djal,dbubble, ebubble, dsa); // ID stage
72
73     /*
74     pipedereg de_reg (
75 dwreg,dm2reg,dwmem,daluc,daluimm,da,db,dimm,drn,dshift,djal,dpc4,clock,rese
76 tn, drs, drt, dbubble,
77 ewreg,em2reg,ewmem,ealuc,ealuimm, ea,eb,eimm,ern0,eshift,ejal,epc4,
78 ers, ert, ebubble ); // ID/EXE流水线寄存器
79
80     /*
81     pipedereg de_reg
82 (dbubble,drs,drt,dwreg,dm2reg,dwmem,daluc,daluimm,da,db,dimm,dsa,drn,dshift
83 ,djal,dpc4,clock,resetn,
84 ebubble,ers,ert,ewreg,em2reg,ewmem,ealuc,ealuimm,ea,eb,eimm,esa,ern0,eshift
85 ,ejal,epc4);
86
87     pipeexe exe_stage ( ealuc,ealuimm,ea,eb,eimm, esa,
88 eshift,ern0,epc4,ejal,ern,ealu); // EXE stage
89
90     pipeemreg em_reg ( ewreg,em2reg,ewmem,ealu,eb,ern,clock,resetn,
91 mwreg,mm2reg,mwmem,malu,mb,mrn);
92
93     pipemem mem_stage ( resetn,mwmem,malu,mb,clock,mem_clock,mmo,
94 real_in_port0,real_in_port1, real_out_port0, real_out_port1,
95 real_out_port2); // MEM stage
96
97     pipemwreg mw_reg (
98 mwreg,mm2reg,mmo,malu,mrn,clock,resetn,wwreg,wm2reg,wmo,walu,wrn); //
99 MEM/WB流水线寄存器
100
101     mux2x32 wb_stage ( walu,wmo,wm2reg,wdi ); // WB stage
102
103 endmodule

```

## 实验仿真结果

[illegible][illegible]

```

1 DEPTH = 32;           % Memory depth and width are required %
2 WIDTH = 32;           % Enter a decimal number %
3 ADDRESS_RADIX = HEX;  % Address and value radices are optional %
4 DATA_RADIX = HEX;    % Enter BIN, DEC, HEX, or OCT; unless %
5                        % otherwise specified, radices = HEX %
6 CONTENT
7 BEGIN
8
9 0 : 20010080;          % (00) main: addi $1, $0, 128 # %

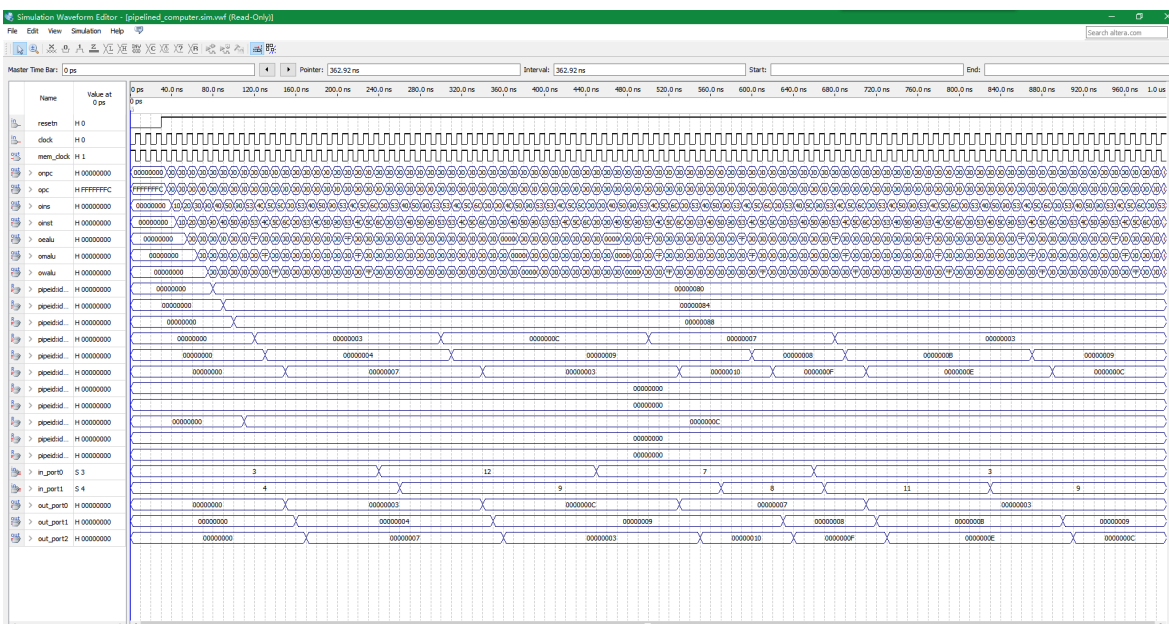
```

```

10 1 : 20020084;          % (04)      addi $2, $0, 132 # %
11 2 : 20030088;          % (08)      addi $3, $0, 136 # %
12 3 : 2009000c;          % (0c)      addi $9, $0, 12  # %
13 4 : 8c240000;          % (10) loop: lw   $4, 0($1)  # %
14 5 : 8c450000;          % (14)      lw   $5, 0($2)  # %
15 6 : 10890005;          % (18)      beq  $4, $9, ssub # %
16 7 : 00853020;          % (1c)      add  $6, $4, $5  # %
17 8 : ac240000;          % (20)      sw   $4, 0($1)  # %
18 9 : ac450000;          % (24)      sw   $5, 0($2)  # %
19 A : ac660000;          % (28)      sw   $6, 0($3)  # %
20 B : 08000004;          % (2c)      j    loop          # %
21 C : 00853022;          % (30) ssub: sub  $6, $4, $5  # %
22 D : ac240000;          % (34)      sw   $4, 0($1)  # %
23 E : ac450000;          % (38)      sw   $5, 0($2)  # %
24 F : ac660000;          % (3c)      sw   $6, 0($3)  # %
25 10 : 08000004;         % (40)      j    loop          # %
26 END ;

```

首先从全局上进行验证，当第一个操作数和12相等的时候，我们输出的结果时两个操作数相加，否则是两个操作数相减。同时我们采用多次循环来验证多组数据，起到检查CPU运行是否正确的作用。



从整体上来看本次实验三个端口的表现正常。现在进入细节部分查看关于数据冒险和控制冒险是否被解决。由于整体运行结果正常，因此我们可以人围除去  $lw$  造成的数据冒险和控制冒险之外，余下部分没有问题。对于跳转产生的控制冒险问题，通过查看如下具体波形

