5段流水CPU设计 - 实验报告

- 学号：517021910653
- 姓名：王祖来

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。

实验仪器和平台

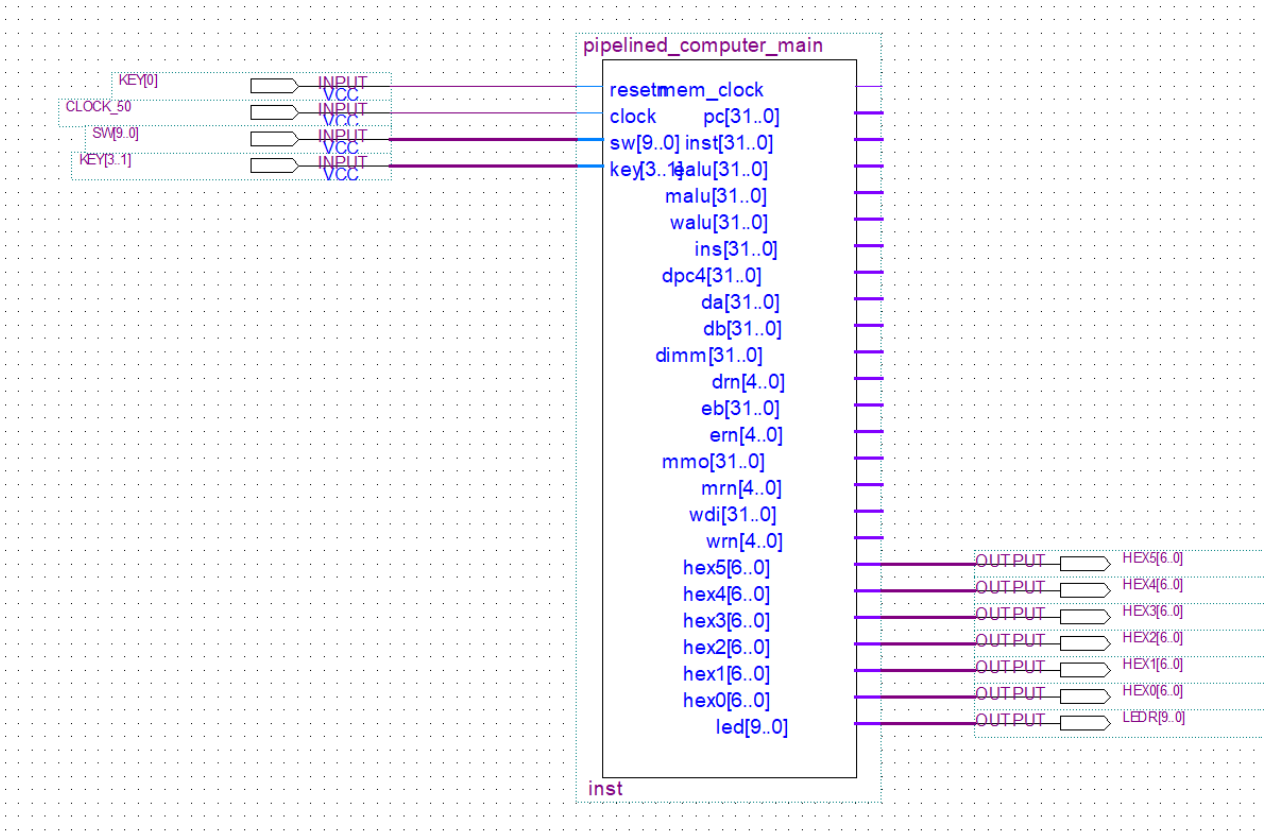
- DE1-SoC实验板
- QuartusII13.1
- ModelSim 10.1

实验内容和任务

1. 采用Verilog在quartusII中实现基本的具有20条MIPS指令的5段流水CPU设计。
2. 利用实验提供的标准测试程序代码，完成仿真测试。
3. 采用I/O统一编址方式，即将输入输出的I/O地址空间，作为数据存取空间的一部分，实现CPU与外部设备的输入输出端口设计。实验中可采用高端地址。
4. 利用设计的I/O端口，通过lw指令，输入DE2实验板上的按键等输入设备信息。即将外部设备状态，读到CPU内部寄存器。
5. 利用设计的I/O端口，通过sw指令，输出对DE2实验板上的LED灯等输出设备的控制信号（或数据信息）。即将对外部设备的控制数据，从CPU内部的寄存器，写入到外部设备的相应控制寄存器（或可直接连接至外部设备的控制输入信号）。
6. 利用自己编写的程序代码，在自己设计的CPU上，实现对板载输入开关或按键的状态输入，并将判别或处理结果，利用板载LED灯或7段LED数码管显示出来。
7. 例如，将一路4bit二进制输入与另一路4bit二进制输入相加，利用两组分别2个LED数码管以10进制形式显示“被加数”和“加数”，另外一组LED数码管以10进制形式显示“和”等。
8. 在实验报告中，汇报自己的设计思想和方法；并以汇编语言的形式，提供以上指令集（MIPS）应用功能的程序设计代码，并提供程序主要流程图。

设计过程

顶层设计



板载50MHz时钟接入流水线CPU主模块，直接作为CPU的时钟。3个按键和10个开关接入流水线CPU主模块。剩下的1个按键作为流水线CPU主模块中寄存器堆和流水线寄存器的复位按键。流水线CPU主模块输出到6个七段数码管和10个发光二极管。主模块的其他输出端口不接到输出的引脚上，只用于仿真验证时查看波形。

主模块

```
module pipelined_computer_main (resetn,clock,mem_clock,
pc,inst,ealu,malu,walu,
  pc4, bpc, jpc, pcsource,
  ins,dpc4,da,db,dimm,drn,eb,ern,mmo,mrn,wdi,wrn,
  sw, key, hex5, hex4, hex3, hex2, hex1, hex0, led);
//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;
//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。
  output mem_clock;
  output [31:0] pc,inst,ealu,malu,walu;
  output [31:0] pc4,bpc,jpc;
  output [1:0] pcsource;
  output [31:0] ins,dpc4,da,db,dimm,eb,mmo,wdi;
  output [4:0] drn,ern,mrn,wrn;
//模块用于仿真输出的观察信号。缺省为wire型。
```

```
  input  [9:0]  sw;
  input  [3:1]  key;
  output [6:0]  hex5, hex4, hex3, hex2, hex1, hex0;
  output [9:0]  led;
// I/O port
  wire [31:0] bpc,jpc,npc,pc4,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;
//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;
//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] drn,ern0,ern,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 [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;
// 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。

  assign mem_clock = ~clock;

  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 ); // 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,
              wrn,wdi,ealu,malu,mmo,wwreg,clock,resetn,
              bpc,jpc,pcsource,wpcir,dwreg,dm2reg,dwmem,daluc,
```

```
                daluimm,da,db,dimm,drn,dshift,djal ); // 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 ( 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 ); // 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,eshift,ern0,epc4,ejal,ern,ealu
  ); // 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,clock,resetn,
                  mwreg,mm2reg,mwmem,malu,mb,mrn); // 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,mem_clock,resetn,mmo,sw,key,
                  hex5,hex4,hex3,hex2,hex1,hex0,led); // MEM stage
  // add I/O design
  //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
```

CPU的工作被分为5个流水段，分别是取指模块（if_stage）、译码模块（id_stage）、运算模块（exe_stage）、数据存取以及I/O模块（mem_stage）、写回阶段模块（wb_stage)。每个流水线阶段之前有一个锁存器，锁存该阶段所需要的信号，分别是程序计数器模块（prog_cnt）、IF/ID流水线寄存器模块（inst_reg）、ID/EXE流水线寄存器模块（de_reg）、EXE/MEM流水线寄存器模块（em_reg）、MEM/WB流水线寄存器模块（mw_reg）。用于同步各阶段RAM存储器和ROM存储器的同步信号mem_clock在主模块中置为系统clock的反相信号，即可以留给信号半个节拍的传输时间。

程序计数器

```
module pipepc(npc, wpcir, clock, resetn, pc);
  input  [31:0] npc;
  input         wpcir, clock, resetn;
  output [31:0] pc;

  dffe32pc pc_dff(npc, clock, resetn, wpcir, pc);
endmodule
```

该阶段作为取指阶段前的流水线寄存器，锁存指令的pc值（pc），在每一个时钟上升沿传入下一条指令的pc值（npc）。wpcir为写入使能信号、clock为时钟信号、resetn为重置信号。该模块使用如下所示的一个特殊的D flip-flop实现。

```
// 32 bit D-flip-flop with enabled bit
module dffe32pc (d,clk,clrn,e,q);
   input  [31:0] d;
   input  clk,clrn,e;
   output [31:0] q;
   reg    [31:0] q;
   always @ (negedge clrn or posedge clk)
      if (clrn == 0) begin
      // pc=-4,npc=0 before CPU start work
      // pc <= 0 at the first posedge
         q <= -4;
      end else begin
         if (e == 1) q <= d;
      end
endmodule
```

该32位D flip-flop在每一个时钟上升沿将输出信号值（q）置为输入信号值（d），在其他时刻持续输出前一个输入值。clk为时钟信号、clkn为重置信号、e为写入使能信号。当重置信号为低电平时，该flip-flop的输出会被重置为-4。当写入使能信号为低电平时，该flip-flop持续输出前一个输入值，不会对时钟上升沿产生响应

- 重置的值为-4的原因：当CPU开始工作，在第一个时钟上升沿的输入信号为npc=pc+4=(-4)+4=0，即第一条指令的地址，可以正常工作。如果重置为0，那么第一个上升沿会取到npc=pc+4=0+4=4地址处的指令，会导致第一条指令被跳过。
- 使能信号的作用：由lw指令造成的load/use hazard需要CPU前两个阶段暂停一个时钟周期，即通过将该flip-flop的使能信号置为低电平实现

取指阶段

```
module pipeif(pcsource, pc, bpc, da, jpc, npc, pc4, ins, rom_clock);
  input  [1:0]  pcsource;
  input  [31:0] pc, bpc, da, jpc;
  input         rom_clock;
  output [31:0] npc, pc4, ins;

  assign pc4 = pc + 4;

  mux4x32 nextpc(pc4, bpc, da, jpc, pcsource, npc);
  pipe_instmem instmem(pc, ins, rom_clock);
endmodule
```

IF取指令模块，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_clock。实验中采用系统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。该模块根据译码阶段传来的pcsource信号，在ROM下一条指令地址（pc4）、译码阶段传来的bne/beq所跳到的地址（bpc）、译码阶段传来的jr所跳到的地址（da）、译码阶段传来的j/jal所跳到的地址（jpc）四个信号中选择一条作为下一条指令的pc值，该过程使用一个32位4路选择器实现。该模块的指令存储模块（instmem）如下所示。

```
module pipe_instmem (addr,inst,rom_clock);
   input  [31:0] addr;
   input         rom_clock;
   output [31:0] inst;

   lpm_rom_irom irom (addr[8:2],rom_clock,inst);

endmodule
```

指令存储单元使用了quartus的宏lpm_rom_irom实现，rom的存储空间大小为7位。输入地址（addr）和存储器时钟信号（ROM存储器），输出指令（inst）。该模块在每一个ROM时钟上升沿，即系统时钟下降沿时输出一条指令。

IF/ID流水线寄存器模块

```
module pipeir(pc4, ins, wpcir, clock, resetn, dpc4, inst);
  input  [31:0] pc4, ins;
  input         wpcir, clock, resetn;
  output [31:0] dpc4, inst;

  dffe32 pc4_dff(pc4, clock, resetn, wpcir, dpc4);
  dffe32 ins_dff(ins, clock, resetn, wpcir, inst);
endmodule
```

该阶段作为译码阶段前的流水线寄存器，锁存指令的pc值加4（dpc4）、指令内容（inst），在每一个时钟上升沿传入来自取指阶段的pc值+4（pc4）和指令内容（ins）。wpcir为写入使能信号、clock为时钟信号、resetn为重置信号。该模块使用两个32位D flip-flop实现。

指令译码模块

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

  wire          rsrtequ, regrt, sext;
  wire   [1:0]  fwda, fwdb;
  wire   [31:0] qa, qb;

  wire   [5:0]  op = inst[31:26];
  wire   [5:0]  func = inst[5:0];
  wire   [4:0]  rs = inst[25:21];
  wire   [4:0]  rt = inst[20:16];
  wire   [4:0]  rd = inst[15:11];
  wire   [15:0] imm = inst[15:0];
  wire   [25:0] addr = inst[25:0];
  wire   [31:0] sa = {27'b0, inst[10:6]}; // 32-bit zero extended sa
  wire          e = sext & inst[15]; // sign bit
  wire   [15:0] imm_ext = {16{e}}; // high 16 sign bit
  wire   [31:0] boffset = {imm_ext[13:0], imm, 2'b00}; // branch addr offset
  wire   [31:0] immediate = {imm_ext, imm}; // extend immediate to high 16

  assign rsrtequ = da == db;
  assign jpc = {dpc4[31:28], addr, 2'b00}; // j|jal: PC <-- (PC+4)
[31:28],addr<<2
  assign bpc = dpc4 + boffset; // beq|bne: PC <-- PC + 4 + (sign)imm << 2
  assign dimm = op == 6'b000000 ? sa : immediate; // unsigned sa for shift or
signed imm for alu

  pipe_cu cu(op, func, rs, rt, ern, mrn, rsrtequ, ewreg, em2reg, mwreg,
mm2reg,
    wpcir, dwreg, dm2reg, dwmem, djal, daluimm, dshift, regrt, sext, pcsource,
fwda, fwdb, daluc);
  regfile rf(rs, rt, wdi, wrn, wwreg, clock, resetn, qa, qb);
  mux4x32 selecta(qa, ealu, malu, mmo, fwda, da);
  mux4x32 selectb(qb, ealu, malu, mmo, fwdb, db);
  mux2x5 selectrn(rd, rt, regrt, drn);
```

```
endmodule
```

ID译码模块。该模块将取指阶段传来的指令内容（inst）译码为多个控制信号传到本阶段的寄存器堆、组合逻辑模块和cu模块产生之后各个阶段需要的信号。对指令的处理基本与单周期CPU相同，以下列举了不同之处。

- rsrtequ代替了alu产生的z信号。设计原因：本实验使用一个周期的延迟转移解决流水线CPU的控制相关的问题，五个会引发控制相关问题的指令（j、jal、jr、bne、beq）中，只有bne和beq是在exe运算阶段才能确定是否转移。通过提前比较两个操作数相同，输出rsrtequ，等效于z信号。
- 为了解决流水线冒险，引入了新的信号，如wpcir、fwdb、fwda，通过cu模块判断是否采用这些信号完成流水线暂停或者直通技术。
- 添加了两个32位4路选择器根据cu模块传来的选择信号，从寄存器中读出的值以及直通的值中选出正确的值传到下一个阶段
- 寄存器文件改为在时钟下降沿写入，留出半个周期的信号传输时间，并且保证写回阶段的数据能够在正确的时刻被写入寄存器文件，无需直通或者流水线暂停。

控制单元模块如下所示

```
module pipe_cu (op, func, rs, rt, ern, mrn, rsrtequ, ewreg, em2reg, mwreg,
mm2reg,
  wpcir, wreg, m2reg, wmem, jal, aluimm, shift, regrt, sext, pcsource, fwda,
fwdb, aluc);
  input            rsrtequ, ewreg, em2reg, mwreg, mm2reg;
  input      [4:0] rs, rt, ern, mrn;
  input      [5:0] op, func;
  output           wpcir, wreg, m2reg, wmem, jal, aluimm, shift, regrt, sext;
  output     [1:0] pcsource;
  output reg [1:0] fwda, fwdb;
  output     [3:0] aluc;

   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_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

  wire i_rs = i_add | i_sub | i_and | i_or | i_xor | i_jr | i_addi
         | i_andi | i_ori | i_xori | i_lw | i_sw | i_beq | i_bne;
  wire i_rt = i_add | i_sub | i_and | i_or | i_xor | i_sll | i_srl
         | i_sra | i_sw | i_beq | i_bne;
  // when E requires load and D requires use, load/use hazard happens
  wire load_use_hazard = ewreg & em2reg & (ern != 0) &
              ((i_rs & (ern == rs)) & (i_rt & (ern == rt)));

  assign wpcir = ~load_use_hazard;  // F and D need to wait when load/use
hazard happens
  assign wreg = (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)
           & ~load_use_hazard;  // stalled stage should not change system
states
  assign m2reg = i_lw;
  assign wmem = i_sw & ~load_use_hazard; //  stalled stage should not change
system states
  assign jal = i_jal;
  assign aluimm  = i_addi | i_andi | i_ori | i_xori | i_lw | i_sw  | i_lui;
  assign shift = i_sll | i_srl | i_sra;
  assign regrt = i_addi | i_andi | i_ori | i_xori | i_lw | i_lui;
  assign sext = i_addi | i_lw  | i_sw  | i_beq | i_bne;

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

   assign aluc[3] = i_sra;
   assign aluc[2] = i_sub | i_or  | i_srl | i_sra | i_ori |
            i_beq | i_bne | i_lui;
   assign aluc[1] = i_xor | i_sll | i_srl | i_sra | i_xori |
            i_lui;
   assign aluc[0] = i_and | i_or  | i_sll | i_srl | i_sra |
            i_andi | i_ori;
```

```
  // forwarding logic
  always @(*) begin
    if (ewreg & ~em2reg & (ern != 0) & (ern == rs))
      fwda = 2'b01;
    else if (mwreg & ~mm2reg & (mrn != 0) & (mrn == rs))
      fwda = 2'b10;
    else if (mwreg & mm2reg & (mrn != 0) & (mrn == rs))
      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
```

控制单元读取指令，对指令的op部分和func部分进行分析得出当前CPU所处理的指令，并产生和输出控制信号到CPU的各个部分。该模块大部分设计与单周期CPU相同。不同之处是添加了wpcir信号（发生load/use hazard时使用）、fwda/fwdb信号（判断是否需要直通）。

ID/EXE流水线寄存器模块

```
module pipedereg(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);
  input         dwreg, dm2reg, dwmem, daluimm, dshift, djal, clock, resetn;
  input  [3:0]  daluc;
  input  [4:0]  drn;
  input  [31:0] da, db, dimm, dpc4;
  output        ewreg, em2reg, ewmem, ealuimm, eshift, ejal;
  output [3:0]  ealuc;
  output [4:0]  ern0;
  output [31:0] ea, eb, eimm, epc4;

  dff1 wreg_dff(dwreg, clock, resetn, ewreg);
  dff1 m2reg_dff(dm2reg, clock, resetn, em2reg);
  dff1 wmem_dff(dwmem, clock, resetn, ewmem);
```

```
  dff1 jal_dff(djal, clock, resetn, ejal);
  dff4 aluc_dff(daluc, clock, resetn, ealuc);
  dff1 aluimm_dff(daluimm, clock, resetn, ealuimm);
  dff1 shift_dff(dshift, clock, resetn, eshift);
  dff32 pc4_dff(dpc4, clock, resetn, epc4);
  dff32 a_dff(da, clock, resetn, ea);
  dff32 b_dff(db, clock, resetn, eb);
  dff32 imm_dff(dimm, clock, resetn, eimm);
  dff5 rn_dff(drn, clock, resetn, ern0);
endmodule
```

该阶段作为运算阶段的流水线寄存器，锁存写寄存器信号（ewreg）、是否从内存中取值的信号（em2reg）、写内存使能信号（ewmem）、alu控制信号（ealuc）、多路选择器信号（ealuimm、eshift）、操作数（ea/eb）、写寄存器号（ern0）、pc值+4（epc4），在每一个时钟上升沿传入来自译码阶段的各个对应信号。clock为时钟信号、resetn为重置信号。该模块使用1位、4位、32位D flip-flop实现。

运算模块

```
module pipeexe(ealuc, ealuimm, ea, eb, eimm, eshift, ern0, epc4, ejal, ern,
ealu);
  input         ealuimm, eshift, ejal;
  input  [3:0]  ealuc;
  input  [4:0]  ern0;
  input  [31:0] ea, eb, eimm, epc4;
  output [4:0]  ern;
  output [31:0] ealu;

  wire [31:0] alua, alub, alur, pc8;

  // jal : $31 <= PC + 8
  assign pc8 = epc4 + 4;
  assign ern = ern0 | {5{ejal}};

  mux2x32 selectalua(ea, eimm, eshift, alua);
  mux2x32 selectalub(eb, eimm, ealuimm, alub);
  alu al_unit(alua, alub, ealuc, alur);
  mux2x32 selectalur(alur, pc8, ejal, ealu);
endmodule
```

EXE运算模块，由alu模块和多个多路选择器组成。本模块的设计大部分与单周期CPU相同，不同之处如下：

- 计算pc8=pc+8，作为jal指令的返回地址。设计原因：本实验使用了延迟转移技术，每一个jal指令之后跟着一个会被执行的指令（如nop），因此返回后需要在pc+8处开始执行

alu模块如下所示

```
module alu (a,b,aluc,s);
  input [31:0] a,b;
  input [3:0] aluc;
  output [31:0] s;

  reg [31:0] s;

  always @ (a or b or aluc)
     begin                                    // event
        casex (aluc)
            4'bx000: s = a + b;               //x000 ADD
            4'bx100: s = a - b;               //x100 SUB
            4'bx001: s = a & b;               //x001 AND
            4'bx101: s = a | b;               //x101 OR
            4'bx010: s = a ^ b;               //x010 XOR
            4'bx110: s = b << 16;             //x110 LUI: imm << 16bit

            4'b0011: s = b << a;         //0011 SLL: rd <- (rt << sa)
            4'b0111: s = b >> a;         //0111 SRL: rd <- (rt >> sa)
(logical)
            4'b1111: s = $signed(b) >>> a;    //1111 SRA: rd <- (rt >> sa)
(arithmetic)
            default: s = 0;
        endcase
     end
endmodule
```

本模块基本与单周期CPU设计相同。不同之处在于去掉了z信号的生成，z信号的功能已经被译码阶段的rsrtequ代替。

EXE/MEM流水线寄存器模块

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

  dff1 wreg_dff(ewreg, clock, resetn, mwreg);
  dff1 m2reg_dff(em2reg, clock, resetn, mm2reg);
  dff1 wmem_dff(ewmem, clock, resetn, mwmem);
  dff32 alu_dff(ealu, clock, resetn, malu);
  dff32 b_dff(eb, clock, resetn, mb);
  dff5 rn_dff(ern, clock, resetn, mrn);
```

```
endmodule
```

该阶段作为数据存取和I/O阶段的流水线寄存器，锁存写寄存器信号（mwreg）、是否从内存中取值的信号（mm2reg）、写内存使能信号（mwmem）、alu结果值（malu）、rs寄存器读出的值（mb）、写寄存器号（mrn），在每一个时钟上升沿传入来自运算阶段的各个对应信号。clock为时钟信号、resetn为重置信号。该模块使用1位、5位、32位D flip-flop实现。

数据存取和I/O模块

```
module pipemem(mwmem, malu, mb, ram_clock, resetn,
  mmo, sw, key, hex5, hex4, hex3, hex2, hex1, hex0, led);
  input         mwmem, ram_clock, resetn;
  input  [31:0] malu, mb;
  input  [9:0]  sw;
  input  [3:1]  key;
  output [31:0] mmo;
  output [6:0]  hex5, hex4, hex3, hex2, hex1, hex0;
  output [9:0]  led;

  pipe_datamem datamem(resetn, malu, mb, mmo, mwmem, ram_clock,
                sw, key, hex5, hex4, hex3, hex2, hex1, hex0, led);
endmodule
```

本模块通过datamem数据存取和I/O模块实现，如下所示：

```
module pipe_datamem (resetn,addr,datain,dataout,we,ram_clock,
  sw,key,hex5,hex4,hex3,hex2,hex1,hex0,led);

  input       resetn;
   input  [31:0]  addr;
   input  [31:0]  datain;

   input          we, ram_clock;
  input  [9:0]  sw;
  input  [3:1]  key;
   output [31:0]  dataout;
  output [6:0]  hex5, hex4, hex3, hex2, hex1, hex0;
  output [9:0]  led;
  reg  [31:0] dataout;
  reg  [6:0]  hex5, hex4, hex3, hex2, hex1, hex0;
  reg  [9:0]  led;

  wire   [31:0] mem_dataout;
   wire          write_enable;
   assign        write_enable = we & (addr[31:8] != 24'hffffff);  // leave 8
bit address for I/O port
```

```
  // synchronously input data to dram
   lpm_ram_dq_dram  dram(addr[6:2],ram_clock,datain,write_enable,mem_dataout
);

  // synchronously input data to I/O device(hex and led) or reset data on I/O
device
  always @ (posedge ram_clock or negedge resetn)
    begin
      if (!resetn) begin
        hex5 <= 7'b1111111;
        hex4 <= 7'b1111111;
        hex3 <= 7'b1111111;
        hex2 <= 7'b1111111;
        hex1 <= 7'b1111111;
        hex0 <= 7'b1111111;
        led <= 10'b0;
      end
      else if (we) begin
        case (addr)
          32'hffffff00: hex0 <= datain[6:0];
          32'hffffff10: hex1 <= datain[6:0];
          32'hffffff20: hex2 <= datain[6:0];
          32'hffffff30: hex3 <= datain[6:0];
          32'hffffff40: hex4 <= datain[6:0];
          32'hffffff50: hex5 <= datain[6:0];
          32'hffffff60: led <= datain[9:0];
        endcase
      end
    end

  // asynchronously output data from I/O device(switch or key) or dram
  always @ (*)
    begin
      case (addr)
        32'hffffff70:  dataout <= {29'b0, key};
        32'hffffff80: dataout <= {22'b0, sw};
        default: dataout <= mem_dataout;
      endcase
    end
endmodule
```

数据存储单元使用了quartus的宏lpm_ram_dq_dram实现，rom的存储空间大小为5位。输入地址（addr）、数据（datain）和写信号（we），输出数据（dataout）。

I/O设计：预留地址（0xffffff00 - 0xffffffff）作为I/O端口

- 地址0xffffff00 - 0xffffff50的写入信号会分别输入到6个七段数码管上
- 地址0xffffff60 - 0xffffff69的写入信号会输入到10个发光二极管上

- 对地址0xffffff70的读信号会将key[3:1]的按键状态作为低3位与高29位的0合并输出
- 对地址0xffffff80的读信号会将sw[9:0]的开关状态作为低10位与高22位的0合并输出
- 对非预留地址的读或写时，write_enable=1，会在lpm_ram_dq_dram中进行读写

MEM/WB流水线寄存器模块

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

  dff1 wreg_dff(mwreg, clock, resetn, wwreg);
  dff1 m2reg_dff(mm2reg, clock, resetn, wm2reg);
  dff32 alu_dff(malu, clock, resetn, walu);
  dff32 mo_dff(mmo, clock, resetn, wmo);
  dff5 rn_dff(mrn, clock, resetn, wrn);
endmodule
```

该阶段作为写回阶段的流水线寄存器，锁存写寄存器信号（wwreg）、是否从内存中取值的信号（wm2reg）、从内存中读出的值（wmo）、alu结果值（walu）、写寄存器号（wrn），在每一个时钟上升沿传入来自数据存储和I/O阶段的各个对应信号。clock为时钟信号、resetn为重置信号。该模块使用1位、5位、32位D flip-flop实现。

写回阶段

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

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，所以仅用一个多路器的实例即可实现该部分。

MIPS计算器程序设计

指令设计

```
-- Copyright (C) 1991-2013 Altera Corporation
-- Your use of Altera Corporation's design tools, logic functions
-- and other software and tools, and its AMPP partner logic
```

```
-- functions, and any output files from any of the foregoing
-- (including device programming or simulation files), and any
-- associated documentation or information are expressly subject
-- to the terms and conditions of the Altera Program License
-- Subscription Agreement, Altera MegaCore Function License
-- Agreement, or other applicable license agreement, including,
-- without limitation, that your use is for the sole purpose of
-- programming logic devices manufactured by Altera and sold by
-- Altera or its authorized distributors.  Please refer to the
-- applicable agreement for further details.

-- Quartus II generated Memory Initialization File (.mif)

WIDTH=32;
DEPTH=128;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
 0 : 3c070000;  % (000) start:  lui $7, 0   # reg7 stores op  add=0 sub=1
xor=2   %
 1 : 08000047;  % (004)   j main    # start main program        %
 2 : 00000020;  % (008)   add $0, $0, $0  # nop           %
 3 : 001ef080;  % (00c) sevenseg:   sll $30, $30, 2 # calculate sevenseg table
item addr to load    %
 4 : 8fdd0000;  % (010)   lw $29, 0($30)  # load sevenseg code of arg($30)
from data memory to $29 %
 5 : 03e00008;  % (014)   jr $31    # return            %
 6 : 00000020;  % (018)   add $0, $0, $0  # nop           %
 7 : 0000e820;  % (01c) split:  add $29, $0, $0 # set reg29 to 0          %
 8 : 23defff6;  % (020) split_loop: addi $30, $30, -10  # reg30 = result - 10
%
 9 : 001ee7c3;  % (024)   sra $28, $30, 31  # extend sign digit of the result
%
 A : 17800004;  % (028)   bne $28, $0, split_done # if reg30 is negative, goto
split_done   %
 B : 00000020;  % (02c)   add $0, $0, $0  # nop           %
 C : 23bd0001;  % (030)   addi $29, $29, 1  # reg29++         %
 D : 08000008;  % (034)   j split_loop  # continue loop         %
 E : 00000020;  % (038)   add $0, $0, $0  # nop           %
 F : 23dc000a;  % (03c) split_done: addi $28, $30, 10 # get units digit and
store to $28      %
10 : 03e00008;  % (040)   jr $31    # return            %
11 : 00000020;  % (044)   add $0, $0, $0  # nop           %
12 : 03e0a020;  % (048) display:  add $20, $31, $0  # store return address to
reg20     %
13 : 001dd140;  % (04c)   sll $26, $29, 5 # $26 = 32 * $29          %
```

```
14 : 235aff00;  % (050)   addi $26, $26, 0xff00 # store sevenseg base addr to
reg26     %
15 : 0c000007;  % (054)   jal split   # call split          %
16 : 00000020;  % (058)   add $0, $0, $0  # nop           %
17 : 03a0f020;  % (05c)   add $30, $29, $0  # move $29(returned tens digit) to
$30      %
18 : 0c000003;  % (060)   jal sevenseg  # call sevenseg         %
19 : 00000020;  % (064)   add $0, $0, $0  # nop           %
1A : af5d0010;  % (068)   sw $29, 0x10($26) # show sevenseg tens digit
%
1B : 0380f020;  % (06c)   add $30, $28, $0  # move $28(returned units digit)
to $30     %
1C : 0c000003;  % (070)   jal sevenseg  # call sevenseg         %
1D : 00000020;  % (074)   add $0, $0, $0  # nop           %
1E : af5d0000;  % (078)   sw $29, 0x0($26)  # show sevenseg units digit
%
1F : 0280f820;  % (07c)   add $31, $20, $0  # restore return address        %
20 : 03e00008;  % (080)   jr $31    # return            %
21 : 00000020;  % (084)   add $0, $0, $0  # nop           %
22 : 8c05ff70;  % (088) get_op: lw $5, 0xff70($0) # load keys to reg5
%
23 : 2006ffff;  % (08c)   addi $6, $0, -1 # store 0xffffffff to reg6        %
24 : 00a62826;  % (090)   xor $5, $5, $6  # reg5 = ~reg5          %
25 : 30a60004;  % (094)   andi $6, $5, 0x4  # store key3 to reg6        %
26 : 14c00009;  % (098)   bne $6, $0, add_op  # key3 is pressed, change op to
add     %
27 : 00000020;  % (09c)   add $0, $0, $0  # nop           %
28 : 30a60002;  % (0a0)   andi $6, $5, 0x2  # store key2 to reg6        %
29 : 14c00007;  % (0a4)   bne $6, $0, sub_op  # key2 is pressed, change op to
sub     %
2A : 00000020;  % (0a8)   add $0, $0, $0  # nop           %
2B : 30a60001;  % (0ac)   andi $6, $5, 0x1  # store key1 to reg6        %
2C : 14c00005;  % (0b0)   bne $6, $0, xor_op  # key1 is pressed, change op to
xor     %
2D : 00000020;  % (0b4)   add $0, $0, $0  # nop           %
2E : 03e00008;  % (0b8)   jr $31    # no key pressed, return        %
2F : 00000020;  % (0bc)   add $0, $0, $0  # nop           %
30 : 20c6fffd;  % (0c0) add_op: addi $6, $6, -3 # calculate new opcode(4-3-
2+1=0)     %
31 : 20c6fffe;  % (0c4) sub_op: addi $6, $6, -2 # calculate new opcode(2-
2+1=1)     %
32 : 20c70001;  % (0c8) xor_op: addi $7, $6, 1  # calculate new opcode and
store to $7(1+1=2)   %
33 : 03e00008;  % (0cc)   jr $31    # return            %
34 : 00000020;  % (0d0)   add $0, $0, $0  # nop           %
35 : 14e00004;  % (0d4) calc: bne $7, $0, not_add # check if op is add
%
36 : 00000020;  % (0d8)   add $0, $0, $0  # nop           %
```

```
37 : 00432020;  % (0dc)   add $4, $2, $3  # add a and b, store result to reg4
%
38 : 03e00008;  % (0e0)   jr $31    # return            %
39 : 00000020;  % (0e4)   add $0, $0, $0  # nop           %
3A : 20e8ffff;  % (0e8) not_add:  addi $8, $7, -1 # op--, for checking if op
is sub     %
3B : 15000008;  % (0ec)   bne $8, $0, not_sub # check if op is sub          %
3C : 00000020;  % (0f0)   add $0, $0, $0  # nop           %
3D : 00432022;  % (0f4)   sub $4, $2, $3  # sub a and b, store result to reg4
%
3E : 00042fc3;  % (0f8)   sra $5, $4, 31  # extend sign digit of the result
%
3F : 10a00002;  % (0fc)   beq $5, $0, sub_done  # result is positive, done
%
40 : 00000020;  % (100)   add $0, $0, $0  # nop           %
41 : 00042022;  % (104)   sub $4, $0, $4  # get abs of result         %
42 : 03e00008;  % (108) sub_done: jr $31    # return            %
43 : 00000020;  % (10c)   add $0, $0, $0  # nop           %
44 : 00432026;  % (110) not_sub:  xor $4, $2, $3  # xor a and b, store result
to reg4     %
45 : 03e00008;  % (114)   jr $31    # return            %
46 : 00000020;  % (118)   add $0, $0, $0  # nop           %
47 : 8c01ff80;  % (11c) main: lw $1, 0xff80($0) # load switches to reg1
%
48 : ac01ff60;  % (120)   sw $1, 0xff60($0) # display reg1 to leds        %
49 : 302203e0;  % (124)   andi $2, $1, 0x3e0  # get high bits for value a and
store to reg2   %
4A : 00011142;  % (128)   srl $2, $1, 5 # calculate value a in reg2       %
4B : 3023001f;  % (12c)   andi $3, $1, 0x1f # get low bits as value b and
store to reg3   %
4C : 0c000022;  % (130)   jal get_op    # call get_op         %
4D : 00000020;  % (134)   add $0, $0, $0  # nop           %
4E : 0c000035;  % (138)   jal calc    # call calc           %
4F : 00000020;  % (13c)   add $0, $0, $0  # nop           %
50 : 0080f020;  % (140)   add $30, $4, $0 # move result to reg30        %
51 : 201d0000;  % (144)   addi $29, $0, 0 # set pos to 0          %
52 : 0c000012;  % (148)   jal display # call display          %
53 : 00000020;  % (14c)   add $0, $0, $0  # nop           %
54 : 0040f020;  % (150)   add $30, $2, $0 # store value a to reg30        %
55 : 201d0002;  % (154)   addi $29, $0, 2 # set pos to 2          %
56 : 0c000012;  % (158)   jal display # call display          %
57 : 00000020;  % (15c)   add $0, $0, $0  # nop           %
58 : 0060f020;  % (160)   add $30, $3, $0 # move value b to reg30       %
59 : 201d0001;  % (164)   addi $29, $0, 1 # set pos to 1          %
5A : 0c000012;  % (168)   jal display # call display          %
5B : 00000020;  % (16c)   add $0, $0, $0  # nop           %
5C : 08000047;  % (170)   j main    # continue main           %
5D : 00000020;  % (174)   add $0, $0, $0  # nop           %
END ;
```

数据内存设计

```
WIDTH=32;
DEPTH=32;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
  00  :   00000040;
  01  :   00000079;
  02  :   00000024;
  03  :   00000030;
  04  :   00000019;
  05  :   00000012;
  06  :   00000002;
  07  :   00000078;
  08  :   00000000;
  09  :   00000010;
END;
```

功能

主函数中循环调用get_op、calc、display，达到持续读取按键和开关的输入、进行and/or/xor运算并在七段数码管上输出运算结果的效果。其中按键指定运算符，开关高5位和低5位作为两个运算数。发光二极管显示开关输入，左侧一对七段数码管显示运算数a的十进制，中间一对七段数码管显示运算数b的十进制，右边一对数码管显示运算结果的十进制。

- main: 从0xff80端口读入开关状态，并输入到0xff60端口，达到led灯显示对应按键状态的效果；将开关高五位的状态作为操作数a，存于2号寄存器中；将开关低五位的状态作为操作数b，存于3号寄存器中
- get_op：从按键的I/O端口0xff70中读入按键，并逐位判断按键是否按下，分别跳转到add_op/sub_op/xor_op，并在7号寄存器中记录此运算符
  - key[3]: add_op $7=0
  - key[2]: sub_op $7=1
  - key[1]: xor_op $7=2
- calc: 判断$7中的运算符并跳转到对应指令段进行运算，将结果存于4号寄存器中
- display：split将输入的数字分为十位和个位，结合main函数中存入29号寄存器的数据所指定显示的位置，调用两次sevenseg显示到对应位置的七段数码管中

在编写汇编代码时，为了满足延迟转移技术的需求，该汇编代码的每一条j/jr/jal/bne/beq指令之后插了一条nop指令（add %0,%0,%0）作为延迟槽。

仿真验证

TestBench

```
//================================================
//
// 该 Verilog HDL 代码，是用于对设计模块进行仿真时，对输入信号的模拟输入值的设定。
// 否则，待仿真的对象模块，会因为缺少输入信号，而“不知所措”。
// 该文件可设定若干对目标设计功能进行各种情况下测试的输入用例，以判断自己的功能设计是否正确。
//
// 对于CPU设计来说，基本输入量只有：复位信号、时钟信号。
//
// 对于带I/O设计，则需要设定各输入信号值。
//
//
// ================================================


// `timescale 10ns/10ns             // 仿真时间单位/时间精度
`timescale 1ps/1ps             // 仿真时间单位/时间精度

//
//  (1) 仿真时间单位/时间精度：数字必须为1、10、100
//  (2) 仿真时间单位：模块仿真时间和延时的基准单位
//  (3) 仿真时间精度：模块仿真时间和延时的精确程度，必须小于或等于仿真单位时间
//
//       时间单位：s/秒、ms/毫秒、us/微秒、ns/纳秒、ps/皮秒、fs/飞秒（10负15次方）。


module pipelined_computer_sim;

    reg           resetn_sim;
    reg           clock_50M_sim;
   reg    [9:0] sw_sim;
   reg    [3:1] key_sim;

   wire       mem_clock_sim;
    wire   [6:0]  hex0_sim,hex1_sim,hex2_sim,hex3_sim,hex4_sim,hex5_sim;
   wire   [9:0] led_sim;

   wire   [31:0]  pc_sim,inst_sim,ealu_sim,malu_sim,walu_sim;
   wire   [31:0]  pc4_sim, bpc_sim, jpc_sim;
   wire   [1:0] pcsource_sim;
   wire   [31:0]
ins_sim,dpc4_sim,da_sim,db_sim,dimm_sim,eb_sim,mmo_sim,wdi_sim;
   wire   [4:0]  drn_sim,ern_sim,mrn_sim,wrn_sim;
```

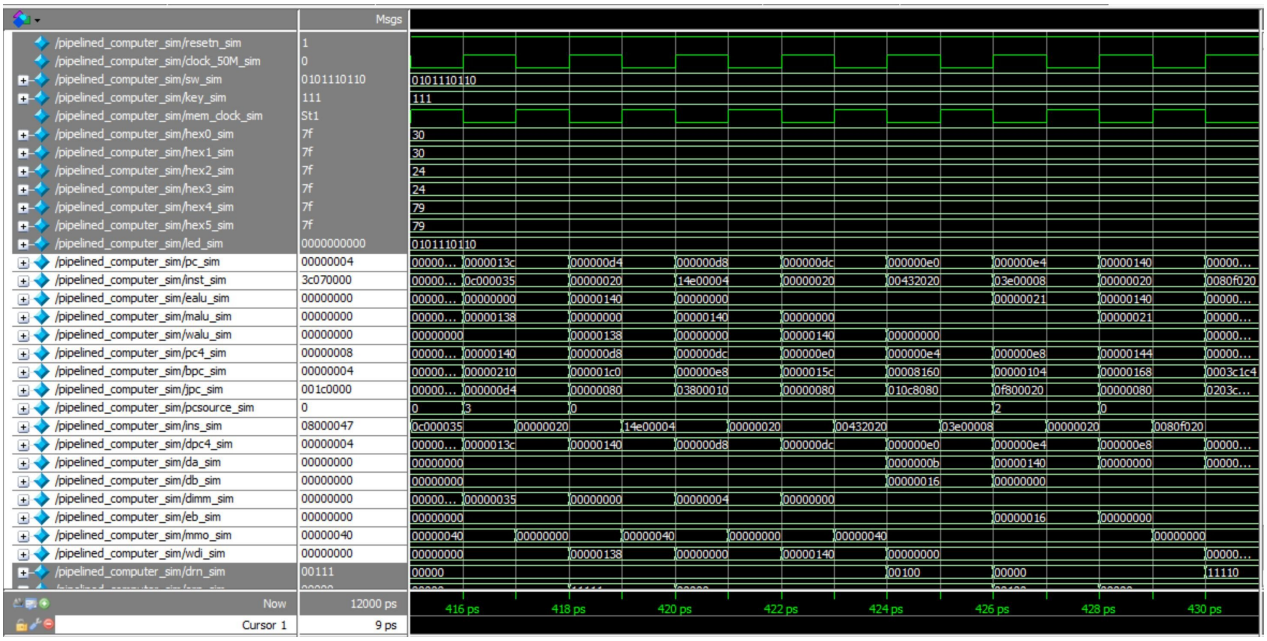
```
   initial
    begin
      key_sim <= 3'b111;
      sw_sim <= 10'b0101110110;
    end

    pipelined_computer_main    pipelined_computer_instance
(resetn_sim,clock_50M_sim,mem_clock_sim, pc_sim,
                                          inst_sim,ealu_sim,malu_sim,walu_sim,
                                          pc4_sim, bpc_sim, jpc_sim,
pcsource_sim,

ins_sim,dpc4_sim,da_sim,db_sim,dimm_sim,

drn_sim,eb_sim,ern_sim,mmo_sim,mrn_sim,wdi_sim,wrn_sim,
                                          sw_sim, key_sim, hex5_sim, hex4_sim,
hex3_sim, hex2_sim,
                                          hex1_sim, hex0_sim, led_sim);

   initial
        begin
            clock_50M_sim = 1;
            while (1)
                #1  clock_50M_sim = ~clock_50M_sim;
        end

   initial
        begin
            resetn_sim = 0;             // 低电平持续10个时间单位，后一直为1。
            while (1)
                #5 resetn_sim = 1;
        end


   // change all switch state every 1000 time unit
   initial
       begin
        while (1)
          #2000 sw_sim = ~sw_sim;
      end


   // press key in turn every 500 time unit
   initial
    begin
      while (1) begin
        #1000 key_sim <= 3'b110;
        #1000 key_sim <= 3'b101;
```

```
        #1000 key_sim <= 3'b011;
      end
    end


    initial
        begin
          $display($time,"resetn=%b clock_50M=%b", resetn_sim, clock_50M_sim);
        end

endmodule
```

生成信号

- 周期为1个时间单位的clock_50M_sim信号作为CPU的clock信号
- resetn信号输入CPU，持续5个时间单位的低电平，然后改变为高电平并保持
- sw信号在0101110110和1010001001之间做周期为2000个时间单位的变换，作为单周期CPU的开关输入
- key信号在110、101、011之间做周期为3000个时间单位的变换，作为单周期CPU的按键输入

验证结果



在ModelSim中进行仿真验证，观察可得一个mem_clk和clk互为反相信号，能够在每一个流水段中留出半个周期的信号传播时间，达到了预期的效果。对各阶段中信号输出的观察可知，流水线各阶段工作正常，数据能够按照设计正确的在各个流水线阶段之间传播。对发光二极管和七段数码管的波形数据分析可得，运行于单周期CPU上的MIPS计算器程序能够正确的输出。

实验总结

实验结果

标准测试程序代码在ModelSim中与其他实验代码经过编译，能正常完成仿真验证，能正确输出预期的波形。实验代码经过编译，载入到开发板后，能正常完成预期的流水线CPU功能。运行在流水线CPU上的MIPS计算器能正常完成预期的功能。通过延迟槽和转发的设计和实现，能够解决加载/使用冒险问题。

经验教训

- 完成代码实现并且通过编译之后的仿真验证阶段，我发现当有两条连续的jal指令进入流水线时，译码阶段的jal传值到程序计数器寄存器之前的npc。此时取指阶段的jal指令会继续进入译码阶段，并将控制流改为它所转移到的指令地址。经过分析，我发现问题CPU的延迟转移和转发逻辑都被正确的实现，问题在于汇编代码的设计。复习了延迟转移技术后我得知MIPS指令系统原本就定义j/jr/jal/bne/beq五条指令是一个周期的延迟转移指令，并且带有一个延迟槽。于是我在每条相应的指令后插入了nop指令（add %0,%0,%0），发现连续的jal指令导致控制流错误的问题被解决。提前充分学习并掌握实验设计的原理和具体实现会对实验带来比较大的帮助。由上述观察分析，我也深刻了解了流水线CPU设计图中jal指令的返回地址需要经过两次+4的加法器的原因。由于jal返回后需要在nop之后的指令开始处理，所以需要存储两条指令后的指令地址。
- 必须注意寄存器文件、ROM指令存储、RAM数据存储以及I/O需要在CPU时钟下降沿进行写入，否则会导致其写入延迟一条指令。 实验过程中，ROM、RAM和I/O的时钟都是由clk的反相信号mem_clock作为输入，能够正确运行。在仿真验证时我发现寄存器文件的写入会慢一个周期。经过调试后发现，由于语义上的问题，我在设计和实现时没有使用提前置反的mem_clock作为寄存器文件的时钟，又忘记取下降沿作为寄存器文件的写入时刻，导致写回阶段无法在正确的时刻传给译码阶段需要用相应寄存器的值。将寄存器文件的写入时刻改为CPU时钟的下降沿后，上述问题被解决，仿真验证能够正确输出结果。
- 观察编译的警告，在modelSim上进行仿真验证，都能提前发现并解决实验设计和实现中所存在的问题。
- 延迟转发技术是解决流水线CPU的加载/使用冒险问题的方法之一。在以前的学习过程中我所了解的插入气泡和暂停以及转发技术是由CPU设计来彻底解决加载/使用冒险问题，无需对编译器提出要求。而延迟转发技术需要编译器在生成汇编指令时，每一条j/jr/jal/bne/beq指令之后跟随一条延迟槽，才能解决加载/使用冒险问题。这种设计给编译器系统和CPU系统的设计带来了耦合，但是简化了CPU的设计和线路消耗。

感受

作为一名软件工程专业大三的学生，在本次试验中我亲身体会到设计流水线CPU、指令集，以及将汇编语言所写的程序运行在流水线CPU上的过程。对流水线CPU的结构，各个阶段的信号输入和输出，各个流水线寄存器的作用、CPU各个模块的功能和设计，作为硬件与软件的接口的指令集的作用有了更加清晰的了解。同时在实验过程中查阅用户手册、verilog语法、使用仿真验证调试硬件需要很多耐心，在此过程中我定位问题、解决问题的能力得到了锻炼和提升。提前了解整体的设计思想以及充分理解各个部分的实现细节会使得实验的设计和实现的过程更加顺利。必要时和同学、老师的交流也带来了很多宝贵的经验和有效的思路。