5 段流水 CPU 设计 - 实验报告

- 学号：
- 姓名：

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。

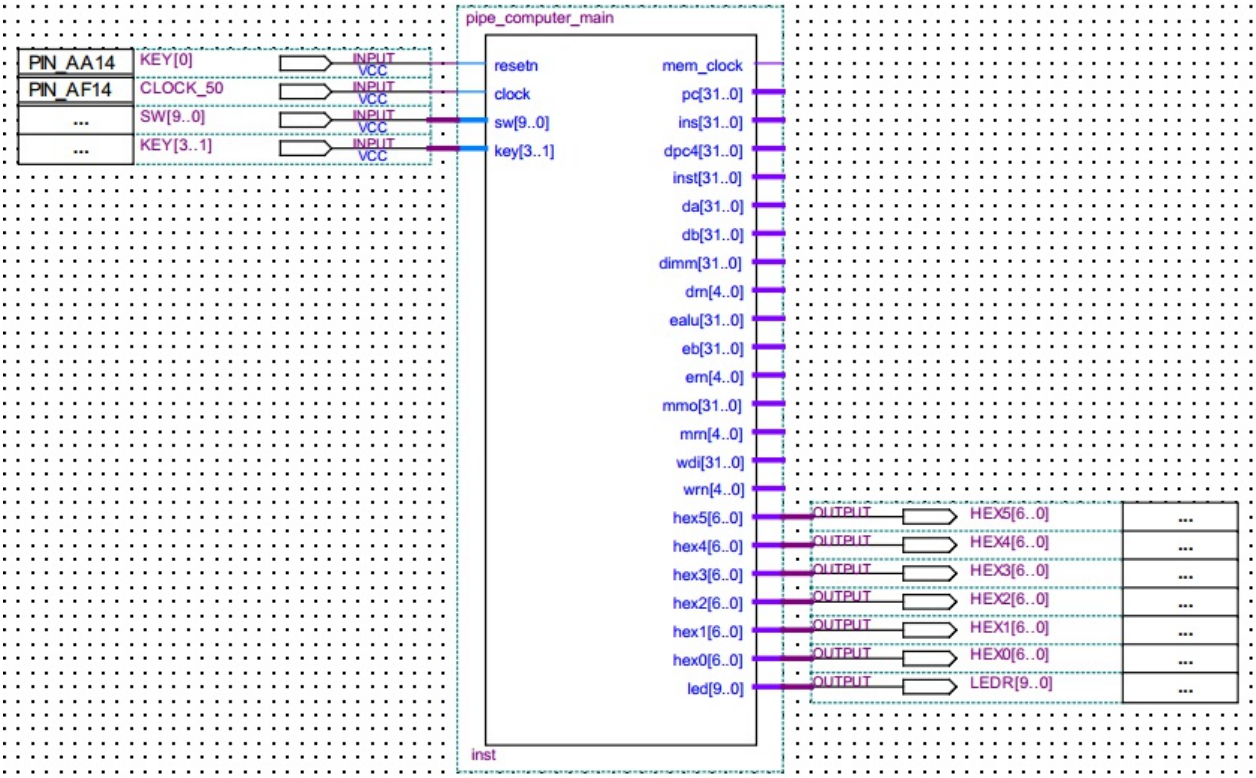
实验仪器与平台

- 硬件：DE1-SoC 实验板
- 软件：Altera Quartus Ⅱ 13.1、 Altera ModelSim 10.1d、MIPS 指令集汇编器

实验内容和任务

1. 采用 Verilog 在 Quartus Ⅱ 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 DE1 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 DE1 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。（具体任务形式不做严格规定，同学可自由创意）。
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 MIPS 指令集的应用功能的程序设计代码，并提供程序主要流程图。

顶层设计



板载 50 MHz 时钟（`CLOCK_50`）可直接作为本 CPU 的时钟信号。`KEY[0]` 作为 CPU 的 reset 信号。CPU 的输入端口包括 `SW[9..0]` 以及 `KEY[3..1]`，输出端口包括 `HEX5[6..0]`、`HEX4[6..0]`、`HEX3[6..0]`、`HEX2[6..0]`、`HEX1[6..0]`、`HEX0[6..0]` 以及 `LED[9..0]`。其余输出信号用于仿真验证时查看，故未分配引脚。

主模块

```
module pipe_computer_main(resetn, clock, mem_clock,
    pc, ins, dpc4, inst, da, db, dimm, drn, ealu, eb, ern, mmo, mrn, wdi, wrn,
    sw, key, hex5, hex4, hex3, hex2, hex1, hex0, led);
    // 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 及时钟信号 clock。
    input         resetn, clock;

    // 模块用于仿真输出的观察信号。
    output        mem_clock;
    output [4:0]  drn, ern, mrn, wrn;
    output [31:0] pc, ins, dpc4, inst, da, db, dimm, ealu, eb, mmo, wdi;

    // 定义计算机的 I/O 端口。
    input  [9:0]  sw;
    input  [3:1]  key;
    output [6:0]  hex5, hex4, hex3, hex2, hex1, hex0;
    output [9:0]  led;

    // 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] pc, 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。ID 指令译码阶段。
    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。EXE 指令运算阶段。
    wire   [31:0] epc4, ea, eb, eimm, ealu;

    // 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] mb, mmo, malu;

    // 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   [31:0] wmo, wdi, walu;

    // 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]  drn, ern0, ern, mrn, wrn;
```

```
    // 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   [3:0]  daluc, ealuc;

    // CU 模块向 IF 阶段模块传递的 PC 选择信号，2bit。
    wire   [1:0]  pcsource;

    // 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          wpcir;

    // ID 阶段产生，需往后续流水级传播的控制信号。
    wire          dwreg, dm2reg, dwmem, daluimm, dshift, djal;

    // 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控制信号。
    wire          ewreg, em2reg, ewmem, ealuimm, eshift, ejal;

    // 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控制信号。
    wire          mwreg, mm2reg, mwmem;

    // 来自于 MEM/WB 流水线寄存器，WB 阶段使用的控制信号。
    wire          wwreg, wm2reg;

    // mem_clock 和 clock 同频率但反相，用作指令同步 ROM 和数据同步 RAM 的时钟信号，其波形需要有别于实验一。
    assign mem_clock = ~clock;

    // 程序计数器模块，是最前面一级 IF 流水段的输入。
    pipe_F_reg prog_cnt(npc, wpcir, clock, resetn, pc);

    // 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。
    // 留给信号半个节拍的传输时间。
    pipe_F_stage if_stage(pcsource, pc, bpc, da, jpc, npc, pc4, ins, mem_clock);

    // 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。
    pipe_D_reg inst_reg(pc4, ins, wpcir, clock, resetn, dpc4, inst);

    // 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。
    pipe_D_stage 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/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。
    pipe_E_reg 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);

    // EXE 运算模块。其中包含 ALU 及多个多路器等。
    pipe_E_stage exe_stage(ealuc, ealuimm, ea, eb, eimm, eshift, ern0, epc4, ejal, ern, ealu);

    // 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。
    pipe_M_reg em_reg(ewreg, em2reg, ewmem, ealu, eb, ern, clock, resetn, mwreg, mm2reg, mwmem, malu, mb, mrn);

    // MEM 数据存取模块。其中包含对数据同步 RAM 的读写访问。
    // 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。
    pipe_M_stage mem_stage(mwmem, malu, mb, mem_clock, resetn, mmo, sw, key, hex5, hex4, hex3, hex2, hex1, hex0, led);

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

    // WB 写回阶段模块。
    pipe_W_stage wb_stage(walu, wmo, wm2reg, wdi);
endmodule
```

该计算机的工作被划分为 5 个阶段（取指、译码、执行、访存、写回），每个阶段前有一个流水线寄存器用于锁存阶段之间需要传递的信息，形成 5 段流水线。

IF 流水段前寄存器

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

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

此寄存器锁存 PC 值信号。`clock`、`resetn`、`wpcir` 分别作为时钟信号、重置信号和（写入）使能信号。使能信号用于实现暂停（stall）。这里使用了一个特殊的 D 触发器（`dffe32pc`），它会在重置时把值置为 -4，以保证第一条指令（PC = 0）能正常执行。

```
// 32bit D flip-flop with an enable signal for PC.
module dffe32pc(d, clk, clrn, e, q);
    input      [31:0] d;
    input             clk, clrn, e;
    output reg [31:0] q;

    always @(negedge clrn or posedge clk)
        if (clrn == 0) begin
            q <= -4; // Make sure instruction at address 0 will be properly executed.
        end else begin
            if (e == 1)
                q <= d;
        end
endmodule
```

取指阶段

```
module pipe_F_stage(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
```

取指阶段根据译码阶段传来的 `pcsource` 信号及各个可能的新 PC 位置计算下一条指令取指的地址，同时访问指令存储取出当前指令。指令存储模块（`pipe_instmem`）与单周期的设计基本相同，唯一的区别在于时钟信号上。这里采用了 `rom_clock` 信号为其时钟信号（同步读），它与 `clock` 信号反相，也即在每个周期一半的时刻读出指令（预留半个周期的时间等待信号稳定）。

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

    rom_1port irom(addr[8:2], rom_clock, inst);
endmodule
```

IF/ID 流水线寄存器

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

    dffe32 pc4_r_d(pc4, clock, resetn, wpcir, dpc4);
    dffe32 ins_r_d(ins, clock, resetn, wpcir, inst);
endmodule
```

此寄存器锁存 IF 和 ID 阶段需要传递的信号。`clock`、`resetn`、`wpcir` 分别作为时钟信号、重置信号和（写入）使能信号。使能信号用于实现暂停（stall）。32 位 D 触发器（`dffe32`）在重置时将值清零，其余行为与前述的 `dffe32pc` 相同。

译码阶段

```
module pipe_D_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);
    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]}; // zero extend sa to 32 bits for shift instruction
    wire          e = sext & inst[15]; // the bit to extend
    wire   [15:0] imm_ext = {16{e}}; // high 16 sign bit when sign extend (otherwise 0)
    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};
    assign bpc = dpc4 + boffset;
    assign dimm = op == 6'b000000 ? sa : immediate; // combine sa and immediate to one signal

    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
```

译码阶段模块包括控制器（`pipe_cu`）、寄存器文件（`regfile`）、几个多路选择器以及一些链路等。各类指令的处理方法与单周期基本相同。以下是几个不同之处：

- 译码阶段提前计算的 `rsrtequ` 信号取代了原来单周期由 ALU 产生的 `z` 条件码传递给控制器。
- `sa` 和 `imm` 两个信号被合并为一个信号向后传输（它们不会同时出现）。
- 控制器需要检查是否出现流水线冒险情况，并做出相应的处理。
- 寄存器文件改为在 `clock` 下降沿写入（即在周期的一半时刻写入，方便读出同一周期内写回阶段写入的值），其余行为与单周期相同。

- `selecta` 和 `selectb` 两个多路选择器按照控制器传来的选择信号，从寄存器读取结果和几个转发源中选择应当使用的值。

下面详细展示控制器的设计：

```
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 == 6'b000000;
    wire i_add = r_type & func == 6'b100000;
    wire i_sub = r_type & func == 6'b100010;
    wire i_and = r_type & func == 6'b100100;
    wire i_or  = r_type & func == 6'b100101;
    wire i_xor = r_type & func == 6'b100110;
    wire i_sll = r_type & func == 6'b000000;
    wire i_srl = r_type & func == 6'b000010;
    wire i_sra = r_type & func == 6'b000011;
    wire i_jr  = r_type & func == 6'b001000;
    wire i_addi = op == 6'b001000;
    wire i_andi = op == 6'b001100;
    wire i_ori  = op == 6'b001101;
    wire i_xori = op == 6'b001110;
    wire i_lw   = op == 6'b100011;
    wire i_sw   = op == 6'b101011;
    wire i_beq  = op == 6'b000100;
    wire i_bne  = op == 6'b000101;
    wire i_lui  = op == 6'b001111;
    wire i_j    = op == 6'b000010;
    wire i_jal  = op == 6'b000011;

    // Determine which instructions use rs/rt.
    wire use_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 use_rt = i_add | i_sub | i_and | i_or | i_xor | i_sll | i_srl | i_sra | i_sw | i_beq | i_bne;

    wire load_use_hazard = ewreg & em2reg & (ern != 0) & ((use_rs & (ern == rs)) | (use_rt & (ern == rt)));

    // When load/use hazard happens, stall F and D registers (stall PC),
    // and generate a bubble to E register (by forbidding writing registers and memory).
    assign wpcir = ~load_use_hazard;
    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;
    assign m2reg = i_lw;
    assign wmem = i_sw & ~load_use_hazard;
    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.
    // Forward priority: Look for E stage first, then M stage.
    // Also, we should not forward r0.
    always @(*) begin
        if (ewreg & ~em2reg & (ern != 0) & (ern == rs))
            fwda = 2'b01; // Forward from ealu.
        else if (mwreg & ~mm2reg & (mrn != 0) & (mrn == rs))
            fwda = 2'b10; // Forward from malu.
```

```
        else if (mwreg & mm2reg & (mrn != 0) & (mrn == rs))
            fwda = 2'b11; // Forward from mmo.
        else
            fwda = 2'b00; // Do not forward.
    end

    always @(*) begin
        if (ewreg & ~em2reg & (ern != 0) & (ern == rt))
            fwdb = 2'b01; // Forward from ealu.
        else if (mwreg & ~mm2reg & (mrn != 0) & (mrn == rt))
            fwdb = 2'b10; // Forward from malu.
        else if (mwreg & mm2reg & (mrn != 0) & (mrn == rt))
            fwdb = 2'b11; // Forward from mmo.
        else
            fwdb = 2'b00; // Do not forward.
    end
endmodule
```

控制器产生的大部分控制信号与原来单周期的相同，新增部分用于处理流水线冒险情形。

- 数据冒险：
  - 大部分数据冒险都能用转发来解决。转发源可以是上一条指令（当前 E 阶段的 ALU 结果）或上上条指令（当前 M 阶段的 ALU 结果、读内存结果）。需要考虑到转发的优先级（由近至远），同时禁止转发 0 号寄存器（因为对 0 号寄存器的写入是无效的）。
  - 加载/使用冒险不能用转发来解决，因为必须等待一个周期才能获得内存中的数据。当检测到这种冒险情形时，控制器向 F 和 D 寄存器发出暂停信号（`wpcir` 置为 0），同时向 E 寄存器传递一个气泡（将 `wreg` 和 `wmem` 信号置为 0，禁止写入寄存器和存储器，即相当于一条 `nop` 指令）。
- 控制冒险：本 MIPS 处理器使用延迟槽（delay slot）特性消除了控制冒险。对于任意的转移指令（无条件跳转、条件跳转、调用和返回），其下一条指令（PC + 4）都会在转移指令之后被执行，然后再采取转移行动。第 N 条指令应该从哪里取是由第 N - 2 条指令决定，而非由第 N - 1 条指令决定。这样的特性简化了流水线的设计，因为不需要实现分支预测逻辑了（也不会再有分支预测错误的处罚），译码阶段一定能得到正确的转移目标。缺点是这样指令集的语义会有点奇怪。

ID/EXE 流水线寄存器

```
module pipe_E_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);
    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_r_e(dwreg, clock, resetn, ewreg);
    dff1 m2reg_r_e(dm2reg, clock, resetn, em2reg);
    dff1 wmem_r_e(dwmem, clock, resetn, ewmem);
    dff1 aluimm_r_e(daluimm, clock, resetn, ealuimm);
    dff1 shift_r_e(dshift, clock, resetn, eshift);
    dff1 jal_r_e(djal, clock, resetn, ejal);
    dff4 aluc_r_e(daluc, clock, resetn, ealuc);
    dff5 rn_r_e(drn, clock, resetn, ern0);
    dff32 a_r_e(da, clock, resetn, ea);
    dff32 b_r_e(db, clock, resetn, eb);
    dff32 imm_r_e(dimm, clock, resetn, eimm);
    dff32 pc4_r_e(dpc4, clock, resetn, epc4);
endmodule
```

此寄存器锁存 ID 和 EXE 阶段需要传递的信号。`dff1`、`dff4`、`dff5`、`dff32` 为不同位数的 D 触发器，除无使能信号以外，其余行为与前述的 `dffe32` 相同。

执行阶段

```
module pipe_E_stage(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;

    assign pc8 = epc4 + 4;
    assign ern = ern0 | {5{ejal}}; // jal: r31 <-- pc8;

    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
```

执行阶段与原单周期的对应部分逻辑基本相同。不同之处在于：

- 对 `jal` 指令的处理被放进了这一阶段。由于延迟槽特性，写入的返回地址应由 PC + 4 改为 PC + 8。
- ALU 除不再产生 `z` 条件码以外，其余行为与原来单周期相同。

EXE/MEM 流水线寄存器

```
module pipe_M_reg(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_r_m(ewreg, clock, resetn, mwreg);
    dff1 m2reg_r_m(em2reg, clock, resetn, mm2reg);
    dff1 wmem_r_m(ewmem, clock, resetn, mwmem);
    dff5 rn_r_m(ern, clock, resetn, mrn);
    dff32 alu_r_m(ealu, clock, resetn, malu);
    dff32 b_r_m(eb, clock, resetn, mb);
endmodule
```

此寄存器锁存 EXE 和 MEM 阶段需要传递的信号。

访存阶段

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

访存阶段访问数据存储模块（`pipe_datamem`）完成内存数据及 I/O 端口的读写。同步读写 RAM 的时钟改用与 `clock` 信号反相的

`ram_clock` 信号，也即在每个周期一半的时刻读写内存（预留半个周期的时间等待信号稳定）。其余行为与单周期相同。

```
module pipe_datamem(addr, datain, dataout, we, ram_clock, resetn, sw, key, hex5, hex4, hex3, hex2, hex1, hex0, led);
    input              we, ram_clock, resetn;
    input      [31:0]  addr, datain;
    input      [9:0]   sw;
    input      [3:1]   key;
    output reg [31:0]  dataout;
    output reg [6:0]   hex5, hex4, hex3, hex2, hex1, hex0;
    output reg [9:0]   led;

    wire               write_enable;
    wire       [31:0]  mem_dataout;

    assign write_enable = we & (addr[31:8] != 24'hffffff);

    ram_1port dram(addr[6:2], ram_clock, datain, write_enable, mem_dataout);

    // IO ports design.
    always @(posedge ram_clock or negedge resetn) begin
        if (!resetn) begin // reset hexs and leds
            hex0 <= 7'b1111111;
            hex1 <= 7'b1111111;
            hex2 <= 7'b1111111;
            hex3 <= 7'b1111111;
            hex4 <= 7'b1111111;
            hex5 <= 7'b1111111;
            led <= 10'b0000000000;
        end else if (we) begin // write when ram_clock posedge comes
            case (addr)
                32'hffffff20: hex0 <= datain[6:0];
                32'hffffff30: hex1 <= datain[6:0];
                32'hffffff40: hex2 <= datain[6:0];
                32'hffffff50: hex3 <= datain[6:0];
                32'hffffff60: hex4 <= datain[6:0];
                32'hffffff70: hex5 <= datain[6:0];
                32'hffffff80: led <= datain[9:0];
            endcase
        end
    end

    always @(*) begin // read asynchonously
        case (addr)
            32'hffffff00: dataout = {22'b0, sw};
            32'hffffff10: dataout = {28'b0, key, 1'b1}; // can only read key[3..1], key0 is used for reset
            default: dataout = mem_dataout;
        endcase
    end
endmodule
```

注：在此处修正了单周期设计中潜藏的错误，`dataout` 所在的 `always` 块应当设计为异步的组合逻辑。若采用同步的 `always @(posedge ram_clock)`，读取 RAM 和这个 `always` 块的执行同时发生，这样打入 `dataout` 锁存器的可能是 `mem_dataout` 的旧值而非刚读出的新值，进而造成读取 RAM 读出的值错误。

MEM/WB 流水线寄存器

```
module pipe_W_reg(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_r_w(mwreg, clock, resetn, wwreg);
    dff1 m2reg_r_w(mm2reg, clock, resetn, wm2reg);
    dff5 rn_r_w(mrn, clock, resetn, wrn);
    dff32 mo_r_w(mmo, clock, resetn, wmo);
    dff32 alu_r_w(malu, clock, resetn, walu);
```

```
endmodule
```

此寄存器锁存 MEM 和 WB 阶段需要传递的信号。

写回阶段

```
module pipe_W_stage(walu, wmo, wm2reg, wdi);
    input         wm2reg;
    input  [31:0] walu, wmo;
    output [31:0] wdi;

    mux2x32 select_write_data(walu, wmo, wm2reg, wdi);
endmodule
```

写回阶段按照选择信号的指示，从 ALU 结果和读取 RAM 的结果中选取应当写回到寄存器文件的值。

流水线 CPU 测试程序

在本次实验中我自行编写了一段 MIPS 测试程序用于测试本流水线 CPU 的正确性。该测试程序主要测试以下的要素：

- 延迟槽特性
- 转发
- 加载/使用冒险
- 指令集每条指令的语义（如零扩展/符号扩展）

若测试全部通过，会在 6 个七段数码管上显示出 `-PASS-` 字样；若有测试点失败，则显示 `-FAIL-` 字样。

```
start:     lw $30, 0($0)         # load sevenseg code for '-' from datamem
           lw $29, 4($0)         # load sevenseg code for 'P' from datamem
           lw $28, 8($0)         # load sevenseg code for 'A' from datamem
           lw $27, 12($0)        # load sevenseg code for 'S' from datamem
           lw $26, 16($0)        # load sevenseg code for 'F' from datamem
           lw $25, 20($0)        # load sevenseg code for 'I' from datamem
           lw $24, 24($0)        # load sevenseg code for 'L' from datamem
           sw $0, 36($0)         # datamem[0x24] <- 0
           j main                # enter test program
           add $0, $0, $0        # nop padding
test_fail: sw $30, 65392($0)     # display '-' at hex5
           sw $26, 65376($0)     # display 'F' at hex4
           sw $28, 65360($0)     # display 'A' at hex3
           sw $25, 65344($0)     # display 'I' at hex2
           sw $24, 65328($0)     # display 'L' at hex1
           sw $30, 65312($0)     # display '-' at hex0
           j end                 # halt the program
main:      lui $1, 0             # $1 <- 0
           j s1                  # test delay slot of 'j'
           addi $1, $0, 1        # $1 <- 1 should be executed before jumping
           addi $1, $0, 2        # should not come here
s1:        addi $2, $0, 1        # $2 <- 1
           bne $1, $2, test_fail # check $1 == 1
           add $0, $0, $0        # nop padding
           bne $0, $1, s2        # test delay slot of 'bne'
           and $1, $1, $0        # $1 <- 0 should be executed before jumping
s2:        bne $0, $1, test_fail # check $1 == 0
           add $0, $0, $0        # nop padding
           beq $0, $1, s3        # test delay slot of 'beq'
           ori $1, $1, 1         # $1 <- 1 should be executed before jumping
s3:        beq $0, $1, test_fail # check $1 != 0
           add $0, $0, $0        # nop padding
           jal s4                # test delay slot of 'jal'
           xor $1, $1, $1        # $1 <- 0 should be executed before call
           j s5                  # should return here (PC + 8) and goto s5
           add $0, $0, $0        # nop padding
```

```
s4:         bne $0, $1, test_fail # check $1 == 0
            add $0, $0, $0        # nop padding
            jr $ra                # test delay slot of 'jr'
            xori $1, $0, 2        # $1 <- 2 should be executed before return
s5:         addi $2, $2, 1        # $2 <- 2
            bne $1, $2, test_fail # check $1 == 2
            add $1, $1, $1        # $1 <- 4
            add $1, $1, $1        # test forwarding, $1 should be 8
            add $1, $1, $1        # test forwarding, $1 should be 16
            add $1, $1, $1        # test forwarding, $1 should be 32
            sll $2, $2, 4         # $2 <- 32
            bne $1, $2, test_fail # check $1 == 32
            srl $2, $2, 2         # $2 <- 8
            add $0, $0, $0        # nop padding
            add $1, $2, $2        # test forwarding, $1 should be 16
            addi $3, $0, 1        # $3 <- 1
            sll $3, $3, 4         # $3 <- 16
            bne $1, $3, test_fail # check $1 == 16
            lw $1, 28($0)         # $1 <- 204 (from datamem)
            addi $2, $0, 0xcc     # $2 <- 0xcc (204)
            bne $1, $2, test_fail # check $1 == 204 (test forwarding)
            add $0, $1, $1        # should cause no effect
            add $1, $0, $0        # $1 should be 0
            bne $0, $1, test_fail # check $1 == 0 (test forbid forwarding $0)
            addi $1, $0, 1        # $1 <- 1
            add $0, $1, $1        # should cause no effect
            add $1, $1, $1        # $1 <- 2
            add $1, $0, $0        # $1 should be 0
            bne $0, $1, test_fail # check $1 == 0 (test forbid forwarding $0)
            lw $0, 28($0)         # should cause no effect
            add $1, $1, $1        # $1 <- 0
            add $1, $0, $0        # $1 should be 0
            bne $0, $1, test_fail # check $1 == 0 (test forbid forwarding $0)
            addi $1, $0, 1        # $1 <- 1
            sub $1, $1, $1        # $1 <- 0
            bne $0, $1, test_fail # check $1 == 0
            sll $2, $2, 24        # $2 <- 0xcc000000
            srl $3, $2, 31        # $3 <- 0x00000001 (test zero extension)
            sra $2, $2, 31        # $2 <- 0xffffffff (test sign extension)
            lw $1, 32($0)         # $1 <- 0xffffffff (from datamem)
            bne $1, $2, test_fail # check $1 == -1, test load/use hazard
            add $1, $1, $3        # $1 <- 0
            bne $0, $1, test_fail # check $1 == 0
            addi $1, $0, 0xffff   # $1 <- 0xffffffff (test sign extension)
            bne $1, $2, test_fail # check $1 == -1
            xori $1, $1, 0xffff   # $1 <- 0xffff0000 (test zero extension)
            beq $0, $1, test_fail # check $1 != 0
            andi $1, $1, 0        # $1 <- 0
            bne $0, $1, test_fail # check $1 == 0
            or $1, $1, $2         # $1 <- 0xffffffff
            beq $0, $1, test_fail # check $1 != 0
            lui $1, 1             # $1 <- 0x00010000
            sw $1, 36($0)         # datamem[0x24] <- 0x00010000
            lw $2, 36($0)         # $2 <- datamem[0x24] (0x00010000)
            bne $1, $2, test_fail # check $1 == $2 (test load/store)
            addi $3, $0, 1        # $3 <- 1
            sll $3, $3, 16        # $3 <- 0x00010000
            bne $1, $3, test_fail # check $1 == 0x00010000 (test lui)
            sw $30, 65392($0)     # display '-' at hex5
            sw $29, 65376($0)     # display 'P' at hex4
            sw $28, 65360($0)     # display 'A' at hex3
            sw $27, 65344($0)     # display 'S' at hex2
            sw $27, 65328($0)     # display 'S' at hex1
            sw $30, 65312($0)     # display '-' at hex0
end:        j end                 # halt the program
            add $0, $0, $0        # nop padding
```

```
DEPTH = 32;           % 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 %
```

```
CONTENT              % otherwise specified, radixes = HEX %

BEGIN
[0..1F] : 00000000;   % Range--Every address from 0 to 1F = 00000000 %

0 : 0000003F;         % (0)  sevenseg code for '-' %
1 : 0000000C;         % (4)  sevenseg code for 'P' %
2 : 00000008;         % (8)  sevenseg code for 'A' %
3 : 00000012;         % (C)  sevenseg code for 'S' %
4 : 0000000E;         % (10) sevenseg code for 'F' %
5 : 00000079;         % (14) sevenseg code for 'I' %
6 : 00000047;         % (18) sevenseg code for 'L' %
7 : 000000CC;         % (1C) data: 0xcc            %
8 : FFFFFFFF;         % (20) data: 0xffffffff      %
END ;
```

使用 CPU 实现计算器

此计算器程序与单周期实验中设计的行为基本相同（能执行加、减、异或三种运算），唯一的不同点在于调整了部分代码以适应延迟槽特性，故不再赘述。

```
start:       j main_loop              # enter main loop, delay slot underneath
             lui $7, 0                # $7 stores op (0->add (default), 1->sub, 2->xor)
sevenseg:    sll $30, $30, 2          # calculate sevenseg table item addr to load
             jr $ra                   # return, delay slot underneath
             lw $29, 0($30)           # load sevenseg code of arg($30) from data memory to $29
split:       add $29, $0, $0          # $29 stores tens digit
split_loop:  addi $30, $30, -10       # decrement arg($30) by 10
             sra $28, $30, 31         # extend sign digit of the result
             bne $28, $0, split_done  # if $30 has become negative, goto split_done
             add $0, $0, $0           # nop padding
             j split_loop             # continue loop, delay slot underneath
             addi $29, $29, 1         # increment tens digit
split_done:  jr $ra                   # return, delay slot underneath
             addi $28, $30, 10        # get units digit and store to $28
show:        add $20, $31, $0         # store return address to $20
             sll $26, $29, 5          # $26 = 32 * $29(arg2, pos)
             jal split                # call split (passing $30, arg1, value), delay slot underneath
             addi $26, $26, 0xff20    # calculate sevenseg pair base addr and store to $26
             jal sevenseg             # call split (passing $30), delay slot underneath
             add $30, $29, $0         # move $29(returned tens digit) to $30
             sw $29, 16($26)          # show sevenseg tens digit
             jal sevenseg             # call split (passing $30), delay slot underneath
             add $30, $28, $0         # move $28(returned units digit) to $30
             sw $29, 0($26)           # show sevenseg units digit
             add $31, $20, $0         # restore return address
             jr $ra                   # return
get_op:      lw $5, 65296($0)         # load state of keys to $5
             addi $6, $0, -1          # store 32'bffffffff to $6
             xor $5, $5, $6           # $5 = ~$5
             andi $6, $5, 0x8         # get state of key3
             bne $6, $0, add_op       # if key3 is pressed, change op to add
             add $0, $0, $0           # nop padding
             andi $6, $5, 0x4         # get state of key2
             bne $6, $0, sub_op       # if key2 is pressed, change op to sub
             add $0, $0, $0           # nop padding
             andi $6, $5, 0x2         # get state of key1
             bne $6, $0, xor_op       # if key1 is pressed, change op to xor
             add $0, $0, $0           # nop padding
             jr $ra                   # no key pressed, no op change, return
add_op:      addi $6, $6, -5          # calculate new opcode
sub_op:      addi $6, $6, -3          # calculate new opcode
xor_op:      jr $ra                   # return, delay slot underneath
             add $7, $6, $0           # calculate new opcode and store to $7
do_op:       bne $7, $0, not_add      # check if op is add
             add $0, $0, $0           # nop padding
             jr $ra                   # return, delay slot underneath
             add $4, $2, $3           # do add
not_add:     addi $8, $7, -1          # check if op is sub
```

```
            bne $8, $0, not_sub      # check if op is sub
            sub $4, $2, $3           # do sub
            sra $5, $4, 31           # extend sign digit of the result
            beq $5, $0, sub_done     # result is positive, done
            add $0, $0, $0           # nop padding
            sub $4, $0, $4           # result = -result (get abs of result)
sub_done:   jr $ra                   # return
            add $0, $0, $0           # nop padding
not_sub:    jr $ra                   # return, delay slot underneath
            xor $4, $2, $3           # do xor
main_loop:  lw $1, 65280($0)         # load state of switches to $1
            sw $1, 65408($0)         # store $1 to state of leds
            andi $2, $1, 0x3e0       # calculate value1 and store to $2
            jal get_op               # call get_op, delay slot underneath
            srl $2, $1, 5            # calculate value1 and store to $2
            jal do_op                # call do_op (passing $2 and $3), delay slot underneath
            andi $3, $1, 0x1f        # calculate value2 and store to $3
            add $30, $4, $0          # move $4(result) to $30
            jal show                 # call show (passing $30 and $29), delay slot underneath
            addi $29, $0, 0          # set pos to 0 (right pair)
            add $30, $2, $0          # move $2(value1) to $30
            jal show                 # call show (passing $30 and $29), delay slot underneath
            addi $29, $0, 2          # set pos to 2 (left pair)
            add $30, $3, $0          # move $3(value2) to $30
            jal show                 # call show (passing $30 and $29), delay slot underneath
            addi $29, $0, 1          # set pos to 1 (middle pair)
            j main_loop              # loop forever
            add $0, $0, $0           # nop padding
```

```
DEPTH = 32;           % 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 %
CONTENT               % otherwise specified, radixes = HEX %

BEGIN
[0..1F] : 00000000;   % Range--Every address from 0 to 1F = 00000000 %

0 : 00000040;         % (0)  sevenseg code for '0' %
1 : 00000079;         % (4)  sevenseg code for '1' %
2 : 00000024;         % (8)  sevenseg code for '2' %
3 : 00000030;         % (C)  sevenseg code for '3' %
4 : 00000019;         % (10) sevenseg code for '4' %
5 : 00000012;         % (14) sevenseg code for '5' %
6 : 00000002;         % (18) sevenseg code for '6' %
7 : 00000078;         % (1C) sevenseg code for '7' %
8 : 00000000;         % (20) sevenseg code for '8' %
9 : 00000010;         % (24) sevenseg code for '9' %
END;
```

仿真验证

```
`timescale 1ps/1ps

module pipe_computer_sim;
    reg         resetn, clock;
    wire        mem_clock;
    wire [31:0] pc, ins, dpc4, inst, da, db, dimm, ealu, eb, mmo, wdi;
    wire [4:0]  drn, ern, mrn, wrn;
    reg  [9:0]  sw;
    reg  [3:1]  key;
    wire [6:0]  hex5, hex4, hex3, hex2, hex1, hex0;
    wire [9:0]  led;

    pipe_computer_main pipe_computer_instance(resetn, clock, mem_clock,
        pc, ins, dpc4, inst, da, db, dimm, drn, ealu, eb, ern, mmo, mrn, wdi, wrn,
        sw, key, hex5, hex4, hex3, hex2, hex1, hex0, led);
```

```
    initial begin // Generate clock.
        clock = 1;
        while (1)
            #2 clock = ~clock;
    end

    initial begin // Generate a reset signal at the start.
        resetn = 1;
        #1 resetn = 0;
        #5 resetn = 1;
        // while (1) begin // Reset and run pipe test again.
        //  #400 resetn = 0;
        //  #5   resetn = 1;
        // end
    end

    initial begin // Simulate switch changes.
        sw <= 10'b1010101010;
        while (1)
            #2400 sw = ~sw;
    end

    initial begin // Simulate key presses.
        key <= 3'b111;
        while (1) begin
            #800 key <= 3'b101; // key2 pressed, should change to sub mode
            #800 key <= 3'b110; // key1 pressed, should change to xor mode
            #800 key <= 3'b011; // key3 pressed, should change to add mode
        end
    end
endmodule
```

该仿真测试代码模拟出 CPU 工作所需要的时钟信号和复位信号，此外在运行不同汇编程序时有不同的仿真功能：

- 在运行流水线测试程序时，可通过周期性的 reset 来反复运行测试代码验证流水线 CPU 的正确性。（通过取消注释上面的 4 行代码开启）
- 在运行计算器程序时，可在适当的时刻模拟开关和按钮的输入。

可在仿真过程中观察各个输出信号的值是否正确，以验证流水线 CPU 工作是否正常。

实验总结

实验结果

实验代码经过仿真验证，分别运行流水线测试程序和计算器程序，各个输出信号的值均正确无误。代码经过编译综合，载入到开发板后，分别运行流水线测试程序和计算器程序，前者所有测试点全部通过，显示出 `-PASS-` 字样，后者能正常完成预期的多功能计算器的功能。故我们可以认为本次流水线 CPU 的设计是符合预期的。

一些总结和拓展

1. Y86 和 MIPS 在流水线阶段职责上的不同之处：Y86 的指令解析（从指令中抽取出各个字段）是在 F 阶段执行的，而 MIPS 的指令解析是在 D 阶段执行的。究其原因，是因为 Y86 没有延迟槽特性，需要在 F 阶段中完成分支预测逻辑（也就要先解析指令），由第 N - 1 条指令决定第 N 条指令应该从哪里取。而 MIPS 的延迟槽特性使得分支预测不再必要了（详见上文译码阶段对控制器的描述），也就没必要在 F 阶段就早早地把指令解析出来了。
2. 延迟槽特性是为了方便流水线 CPU 的设计，上次实现的单周期 MIPS CPU 并没有延迟槽特性，如果要实现反而会变得麻烦而且奇怪。
3. 关于此设计中各读写操作的同步/异步：指令 ROM 是同步读，寄存器文件是异步读、同步写，数据 RAM 是同步读写，I/O 端口是异步读、同步写。
4. 如何插入气泡：只要一条指令没有写入寄存器和存储器，没有更改条件码（对于有条件码的设计），也没有导致 PC 错误地转移，那么这条指令就没有对整个体系产生副作用，相当于一条空指令（`nop`）。PC 转移的控制由 F 和 D 阶段处理，后续阶段不对此进行干涉，故控制器能阻止其出错。我们只需要在向后续流水线寄存器写入的控制信号中禁止对寄存器、存储器和条件码的写入，就能实现插入一个气泡的目标。

5. 在本 CPU 的设计中，仅 D 阶段能接收转发。但实际上我们也能让 E 阶段接收转发，以提升某种情形下的执行性能。考察以下指令序列，其中第一条指令从内存中读出的值会被第二条指令再写入内存：

```
lw $1, 0($0)         # $1 <- datamem[0x0]
sw $1, 4($0)         # datamem[0x4] <- $1
```

我们可以将第一条指令的内存读取结果 `mmo` 转发到 E 阶段的末尾 `eb` 处，因为它在第二条指令的 M 阶段才会被用到。这样可以避免对加载/使用冒险的处理所带来的一个周期的暂停。

注：这种转发技术即 ICS 课程所介绍的加载转发（load forwarding）技术。

经验教训

在仿真测试中，若发现从寄存器文件中读出的值不符合预期，可以把仿真步长调小（如 1 ps），然后每向前运行一步都使用 ModelSim 的 Memory List 页面查看寄存器文件里面的值，这样就能方便地定位到问题所在之处。

感受

在本次实验中我实现了一个 5 段流水线 CPU，自行编写了一个测试程序证明了其工作的正确性，并在其上用汇编码实现出了一个简单的多功能计算器，更清晰深刻地理解了流水线各个阶段之间的协调配合以及各种流水线冒险的处理方式。这次经验更加丰富了，写得要比前两次顺利了很多。下手写代码之前应该仔细阅读和思考整体的架构，充分理解每一处细节，然后再开始，便会顺利很多。同时，遇到问题不能心急，想出有效的办法定位问题之所在才是关键，必要时和老师、同学沟通能更有效地解决问题。