实验：5段流水CPU设计实验报告

姓名：Meng Yit Koh 学号：517030990022

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。

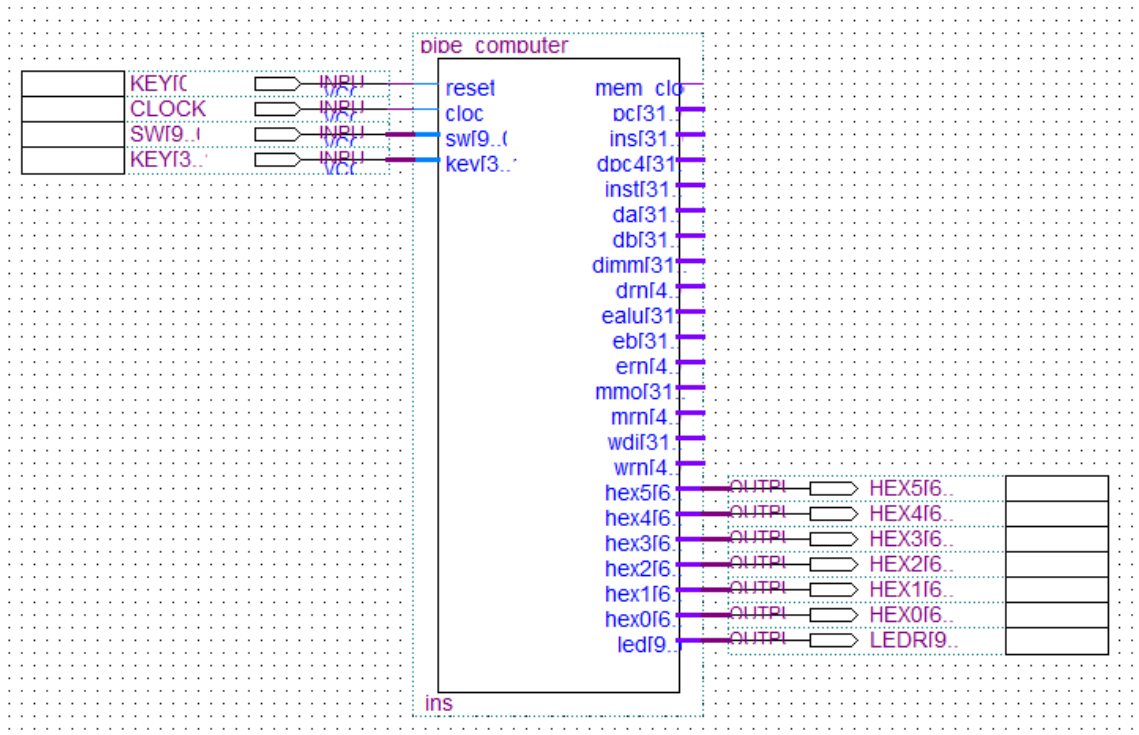
本次实验所用的仪器以及软件

- DE1-SOC 实验板
- Quartus Prime Lite 18.1.0
- ModelSim - Intel FPGA Starter Edition 10.5b

实验内容

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。（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，提供以 上两种指令集（MIPS）应用功能的程序设计代码，并提供程序主要流程图。

Block Diagram



pipe 主函数

```
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);

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

    wire   [31:0] pc, bpc, jpc, npc, pc4, ins, inst;
    wire   [31:0] dpc4, da, db, dimm;
    wire   [31:0] epc4, ea, eb, eimm, ealu;
    wire   [31:0] mb, mmo, malu;
    wire   [31:0] wmo, wdi, walu;
    wire   [4:0]  drn, ern0, ern, mrn, wrn;
    wire   [3:0]  daluc, ealuc;
    wire   [1:0]  pcsource;
    wire          wpcir;
    wire          dwreg, dm2reg, dwmem, daluimm, dshift, djal;
    wire          ewreg, em2reg, ewmem, ealuimm, eshift, ejal;
    wire          mwreg, mm2reg, mwmem;
    wire          wwreg, wm2reg;
```

```
    assign mem_clock = ~clock;

    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
```

注解：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_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
```

注解：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。

取指阶段

```
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
```

注解：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。

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
```

注解：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）。

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
```

控制器和forwarding

```
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
```

注解：控制器产生的大部分控制信号与原来单周期的相同。新增了data hazard forwarding。

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
```

注解：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
```

注解：执行阶段与原单周期的对应部分逻辑基本相同。

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
```

注解：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
```

注解：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。

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
```

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
```

注解：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

我实现了简单加法程序。代码和上实验一致，故不重复说明。代码可参阅提交的文件。由于代码完全一致，也没遇到什么问题，仿真也和上实验一样。具体可以参阅上个实验。

实验总结

实验代码经过编译综合，载入到开发板后，能正常完成预期的 CPU 功能。I/O 处理良好，未出现奇怪显示状况。

感谢上海交通大学软件学院开设这门课，让我学习如何使用 FPGA 实现流水线 CPU 和基本 MIPS 指令。感谢王老师的教导。感谢热心的助教。