自选项目实验：基于流水线**CPU**增加计算**‘0’**值附加指令实验报告

姓名：Meng Yit Koh 学号：517030990022

实验目的

1. 理解计算机 5 大组成部分的协调工作原理，理解存储程序自动执行的原理。
2. 掌握运算器、存储器、控制器的设计和实现原理。重点掌握控制器设计原理 和实现方法。
3. 掌握 I/O 端口的设计方法，理解 I/O 地址空间的设计方法。
4. 会通过设计 I/O 端口与外部设备进行信息交互。
5. 了解流水线设计，并适当添加所需指令。

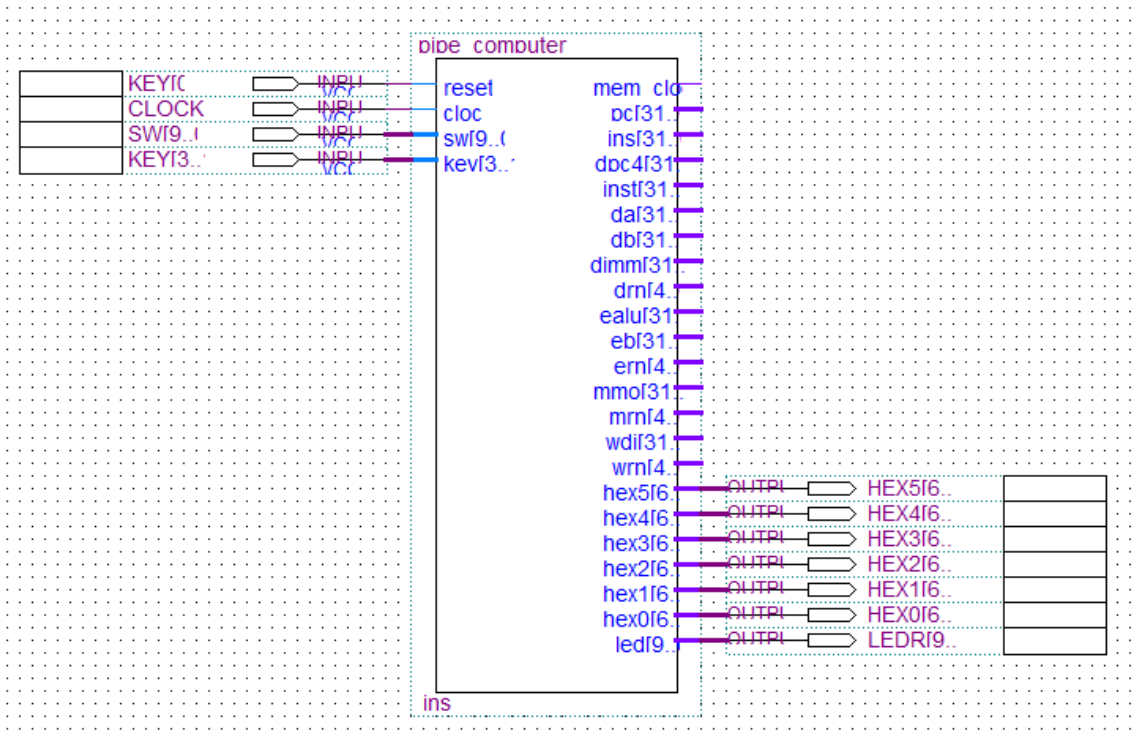
本次实验所用的仪器以及软件

- 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21条MIPS指令（一条新增指令）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。

Block Diagram



由于大部分代码和流水线实验相似，故这里只显示经过修改的代码组件。

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_hamd = r_type & func == 6'b100111;
    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_hamd | 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_hamd | 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_hamd
        | 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 | i_hamd;
    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_hamd | i_xori | i_lui;
    assign aluc[0] = i_and | i_or | i_sll | i_srl | i_sra | i_hamd | 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流水线的相同。新增了新指令的解析。

alu

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

    wire       [31:0] hmd;

    hamd hamd_calc(a, b, hmd);

    always @(*)
        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)
            4'b1011: s = hmd;
            default: s = 0;
        endcase
endmodule
```

注解：基本和原来一致，新增了对新指令的操作。逻辑模块化到了另一个模块实现。

hamd.v

```
module hamd(a, b, d);
    input  [31:0] a, b;
```

```
    output [31:0] d;

    wire   [31:0] c;

    assign c = 8'b11111111 ^ b;
    assign d = c[31] + c[30] + c[29] + c[28] + c[27] + c[26] + c[25] + c[24]
        + c[23] + c[22] + c[21] + c[20] + c[19] + c[18] + c[17] + c[16]
        + c[15] + c[14] + c[13] + c[12] + c[11] + c[10] + c[9] + c[8]
        + c[7] + c[6] + c[5] + c[4] + c[3] + c[2] + c[1] + c[0];
endmodule
```

注解：本模块实现了所有新增指令的逻辑。

流水线新指令测试

我实现了简单程序。代码如下：

```
j 4
sll $30 $30 2
jr $31
lw $29 $30 0
lw $1 $0 65280
andi $1 $1 255
addi $2 $0 202
jal 1
hamd $30 $2 $1
j 4
sw $29 $0 65312
```

运行后没问题。所有功能均运行正常。

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。