中国科学技术大学计算机学院
《计算机组成原理实验》报告

实验题目：____流水线 CPU 设计____

学生姓名：_____徐奥_____

学生学号：_____PB20061343_____

完成日期：____2022 年 4 月 27 日___

计算机实验教学中心制

2020 年 09 月

【实验目标】

1. **理解流水线 CPU 的结构和工作原理**
2. **掌握流水线 CPU 的设计和调试方法，特别是流水线中的数据相关和控制相关的处理**
3. **熟练掌握数据通路和控制器的设计和描述方法**

【实验内容】

题目 1. 设计无数据和控制相关处理的流水线 CPU

在单周期 CPU 的基础上进行设计。相较于单周期 CPU，流水线 CPU 每个时钟周期执行一个流水段，将结果存储到当前流水段与下一流水段之间的寄存器，而当前流水段的数据来源于上一个流水段和本流水段之间的寄存器。简单而言，每个流水段在执行时，只需要考虑将前一个寄存器的值读出来，根据控制信号对数据做相应的处理，将结果存到下一组段间寄存器。

流水线共分为五段，IF、ID、EX、Mem、WB，每两个流水段之间加寄存器，传递下一流水段需要的数据和控制信号。

首先，我选择将所有控制信号逐次向后传，使用一个 32 位寄存器型变量进行存储。控制信号分别为：

```
reg [31:0] EX_ctrl,Mem_ctrl,WB_ctrl;
// 下标从0开始，分别为:
// RegWrite, ALUSrc, MemWrite, MemRead, MemtoReg, PCSrc, Branch,
// ALUop, ALU_equal, ALU_lessthan, ALU_auipc, PC_jal
```

图 1

IF 段，完成更新 PC 和取 IR。指令存储器在具体实现中使用 ROM。核心代码如下：

```
assign pc_add4 = (rstn==0) ? 32'h3000 : IF_pc + 32'h4;
assign n_pc = (PCSrc == 1) ? pc_branch : pc_add4;
assign nxt_pc = (EX_ctrl[11] == 1) ? pc_jal : n_pc;

always @(posedge clk or negedge rstn) begin
    if(!rstn) begin
        IF_pc <= 32'h3000;
    end
    else if(PCSrc) begin
        // 这里很重要，当分支指令的下一条是lw指令，并且应该跳转时，那就直接跳转
        // 如果没有这一步，那么lw指令就会讲原本的跳转给冲掉
        IF_pc <= nxt_pc;
    end
    else if(ID_IR[6:0]==7'b0000011) begin
        // lw, 需要停一个周期
        IF_pc <= IF_pc;
    end
    else begin
        IF_pc <= nxt_pc;
    end
end
```

图 2

```
// instruction
wire [7:0] real_IR_addr;
assign real_IR_addr = (IF_pc-32'h3000)>>2;
instruction IR0(.a(real_IR_addr),.spo(IF_IR));
```

图 3

然后传 PC 和 IR 到下一流水段。

ID 段，完成寄存器读操作，计算立即数，根据 IR 求出部分控制信号，还有部分控制信号需要在 EX 段执行以后才能获取，比如两操作数的大小关系。

寄存器读操作：

```
// 寄存器堆的读操作
wire [4:0] RegReadAddr1,RegReadAddr2;
wire [31:0] RegReadData1,RegReadData2;

assign RegReadAddr1 = ID_IR[19:15];
assign RegReadAddr2 = ID_IR[24:20];
assign RegReadData1 = Registers[RegReadAddr1];
assign RegReadData2 = Registers[RegReadAddr2];
```

图 4

计算立即数的模块：

```
parameter one32 = 32'hffff_f000;
parameter one32_auipc = 32'hfff0_0000;


always @(*) begin
    if(IR[6:0]==7'b0010011) begin
    // addi
        if(IR[31]==1) imm_num = one32 | IR[31:20];
        else imm_num = IR[31:20];
    end
    else if(IR[6:0]==7'b0000011) begin
    // lw
        if(IR[31]==1) imm_num = one32 | IR[31:20];
        else imm_num = IR[31:20];
    end
    else if(IR[6:0]==7'b0100011) begin
    // sw
        if(IR[31]==1) imm_num = one32 | {IR[31:25],IR[11:7]};
        else imm_num = {IR[31:25],IR[11:7]};
    end
    else if(IR[6:0]==7'b1100011) begin
    // beq,blt
        if(IR[31]==1) imm_num = one32 | {IR[31],IR[7],IR[30:25],IR[11:8]};
        else imm_num = {IR[31],IR[7],IR[30:25],IR[11:8]};
    end
```

图 5

```
    else if(IR[6:0]==7'b0010111) begin
    // auipc
        if(IR[31]==1) imm_num = one32_auipc | IR[31:12];
        else imm_num = IR[31:12];
    end
    else if(IR[6:0]==7'b1101111) begin
    // jal
        if(IR[31]==1) imm_num = one32_auipc | {IR[31],IR[19:12],IR[20],IR[30:21]};
        else imm_num = {IR[31],IR[19:12],IR[20],IR[30:21]};
    end
    else if(IR[6:0]==7'b1100111) begin
    // jalr
        if(IR[31]==1) imm_num = one32 | IR[31:20];
        else imm_num = IR[31:20];
    end
end
```

图 6

计算控制信号的模块:

```
assign control[6] = (IR[6:0]==7'b1100011) ? 1 : 0; // beq blt
assign control[3] = (IR[6:0]==7'b0000011) ? 1 : 0; // lw
assign control[4] = (IR[6:0]==7'b0000011) ? 0 : 1; // lw:0; add,addi,auipc,sub:1
assign control[7] = ((IR[6:0]==7'b0110011)&&(IR[31:25]==7'b0100000)) ? 0 : 1;
//assign control[2] = (IR[6:0]==7'b0100011 && ALU_result[8]==0) ? 1 : 0; // sw,并且要写入存储器的地址小于256，也即不是IO地址
assign control[1] = (IR[6:0]==7'b0110011 || IR[6:0]==7'b1100011) ? 0 : 1;
// add,sub,beq,blt:0     addi,auipc,lw,sw:1
assign control[0] = (IR[6:0]==7'b0110011 || IR[6:0]==7'b0010011 || IR[6:0]==7'b0010111 || IR[6:0]==7'b0000011 || IR[6:0]==7'b1101111
// add,sub,addi,auipc,lw,jal,jalr: 1
assign control[10] = (IR[6:0]==7'b0010111) ? 1 : 0;
assign control[11] = (IR[6:0]==7'b1101111 || IR[6:0]==7'b1100111) ? 1 : 0;
//assign control[5] = Branch & ((IR[14:12]==3'b000) ? ALU_equal : ALU_lessthan);

assign control[2] = 0;
assign control[5] = 0;
assign control[8] = 0;
assign control[9] = 0;
assign control[31:12] = 0;
```

图 7

然后，传 PC、IR、A、B、Imm、control 到下一流水段。

EX 段，进行 ALU 的操作，如果是条件跳转指令，那么将根据 ALU 的结果决定是否跳转，如果是 jal、jalr 指令，则直接修改 nxt_pc。

```
wire [31:0] ALU_result;
wire ALU_equal,ALU_lessthan;
ALU ALU0(.a(ALU_A),.b(ALU_B),.op(EX_ctrl[7]),.result(ALU_result),.alu_equal(ALU_equal),.alu_lessthan(ALU_lessthan));

// 更新控制信号
assign PCSrc = EX_ctrl[6] & ((EX_IR[14:12]==3'b000) ? ALU_equal : ALU_lessthan);
wire MemWrite;
assign MemWrite = (EX_IR[6:0]==7'b0100011 && ALU_result[8]==0) ? 1 : 0; // sw,并且要写入存储器的地址小于256，也即不是IO

// beq,blt
wire [31:0] imm_shift;
assign imm_shift = EX_Imm << 1;
assign pc_branch = EX_pc + imm_shift;

// jal,jalr
wire jalr;
assign jalr = (EX_IR[6:0]==7'b1100111) ? 1 : 0;
assign pc_jal = (jalr==1) ? (ALU_A+EX_Imm)&~1 : (EX_pc+{EX_Imm[30:0],1'b0});
```

然后传 control、ALU_result、B、IR 到下一流水段。

Mem 段，完成 Memory 的读写操作。

```
//Memory
//一个读写地址，一个写数据，一个读数据，读使能（好像没什么用），写使能
wire [31:0] MemReadData;
wire [31:0] Chk_Data;
memory memory0(
  .a(Mem_ALU_result[7:0]>>2),        // input wire [7 : 0] a
  .d(Mem_B),         // input wire [31 : 0] d
  .dpra(chk_addr[7:0]),  // input wire [7 : 0] dpra
  .clk(clk),     // input wire clk
  .we(Mem_ctrl[2]),      // input wire we
  .spo(MemReadData),    // output wire [31 : 0] spo
  .dpo(Chk_Data)    // output wire [31 : 0] dpo
);
```

图 9

然后传 control、MemReadData、ALU_result 和 IR 到下一流水段。

WB 段，寄存器的写回操作。但实际上写入寄存器的数据准备工作是在 Mem 阶段完成，写入是在进入 WB 的时钟上升沿完成。

```
always @(posedge clk or negedge rstn) begin
    if(!rstn) begin
    // 寄存器堆清零
        Registers[0] <= 0; Registers[1] <= 0; Registers[2] <= 0; Registers[3] <= 0; Re
        Registers[8] <= 0; Registers[9] <= 0; Registers[10] <= 0; Registers[11] <= 0;
        Registers[16] <= 0; Registers[17] <= 0; Registers[18] <= 0; Registers[19] <= 0
        Registers[24] <= 0; Registers[25] <= 0; Registers[26] <= 0; Registers[27] <= 0
    end
    else begin
        if(Mem_ctrl[0]==1) begin
            if(RegWriteAddr==0) Registers[0] <= 0;
            else if(Mem_IR[6:0]==7'b0000011 && Mem_ALU_result[8]==1) begin
                Registers[RegWriteAddr] <= Reg_io;
            end
            else begin
                Registers[RegWriteAddr] <= RegWriteData;
            end
        end
    end
end
```

图 10

然后是处理 IO 操作。MMIO 的起始地址是 32'h0100，这样的好处

就在于可以避免地址太大，而需要额外加指令。其他操作与单周期 CPU 区别不大，差别主要有两处：单周期 CPU 的写数据到外设，是在一条指令执行完后可以看到外设的结果，而流水线 CPU 是在那条指令 Mem 段结束时看到结果；CPU 的 IO 输入数据的准备是在指令执行的前一个周期，而流水线 CPU 的 IO 输入数据的准备是在对应指令的 Mem 段，也即 WB 流水段的上一段。

最后，处理 Debug 总线，这里与单周期基本相同，唯一的更改是 Debug 总线返回的 pc 是 WB 阶段的 pc，这样设置的好处就是可以根据返回的 pc 值设置 cpu 连续运行的断点。

还需要处理结构相关，共两个方面：存储器相关处理：哈佛结构（指令和数据存储器分开）；寄存器堆相关处理：同一寄存器读写时，写优先（Write First）。

题目 2. 设计仅有数据相关处理的流水线 CPU.

此次实现的 CPU 只会出现写后读的数据相关，也即之前指令的写回寄存器操作还没完成，之后的指令就读取了这个寄存器的值。对此的处理有两个：对于 lw 指令，在其后停止一个周期；对于其他写后读的操作，使用数据定向技术，将执行结果提前传递至之前流水线。

主要涉及的写操作有 addi,add,sub,auipc,lw，这些操作均会对寄存器有写入。而读取寄存器是在 EX 阶段，这是就需要特别判断待读取的寄存器地址是否是其之前两条指令的写回寄存器地址，如果是，那就直接将之前指令的执行结果传递过来。核心代码如下：

```
wire [31:0] ALU_a,ALU_b;
assign ALU_a = (EX_ctrl[10]==1) ? EX_pc : EX_A;
assign ALU_b = (EX_ctrl[1]==0) ? EX_B : ((EX_ctrl[10]==1) ? {EX_Imm[19:0],12'b0} : EX_Imm);
```

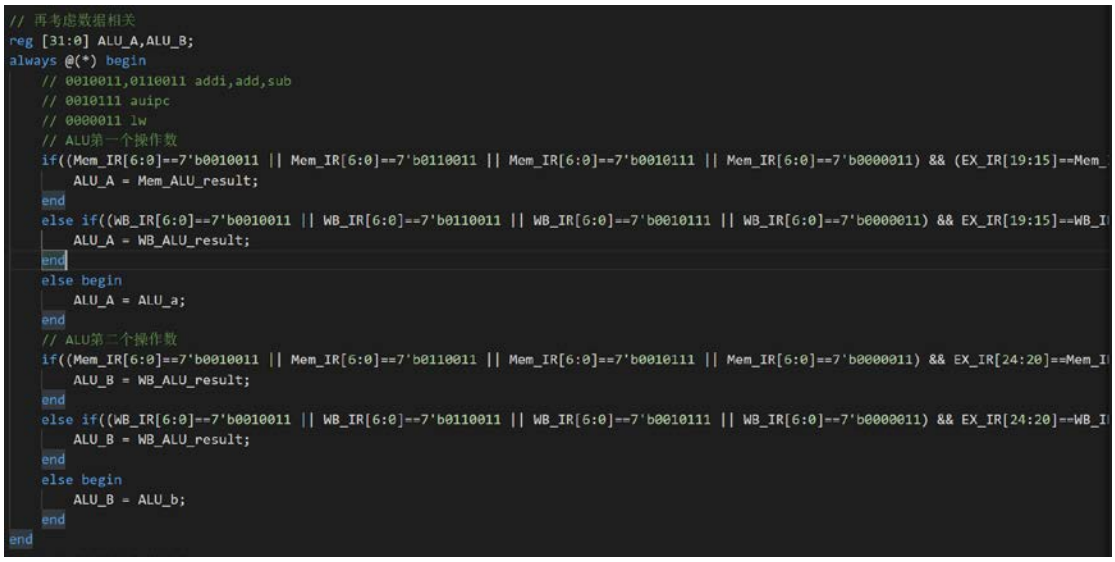

图 12

题目 3. 设计完整的有数据和控制相关处理的流水线 CPU.

也即在题目 2 的基础上增加了控制相关。当执行无条件跳转以及条件分支条件满足时，需要更改 pc 值到跳转的目标地址。跳转指令执行到 EX 段时，会更新 nxt_pc，但此时顺序排在跳转指令后的两条指令已经进入流水线，需要进行清除。具体操作是清除段间寄存器。代码如下：

```
else if(PCSrc == 1 || EX_ctrl[11] == 1) begin
    ID_pc <= 0;
    ID_IR <= 0;
end
```

图 13

```
else if(PCSrc == 1 || EX_ctrl[11] == 1) begin
    EX_pc <= 0;
    EX_IR <= 0;
    EX_A <= 0; EX_B <= 0;
    EX_Imm <= 0;
    EX_ctrl <= 0;
end
```

图 14

至此，基本实现了处理了结构相关、数据相关和控制相关的五级

流水线。用 Lab3 的排序程序进行测试，这个程序是将存储在 Memory 的数据按照升序进行排序。首先将断点设置在代码的最后一条指令，然后点击连续运行。指令执行情况如下：

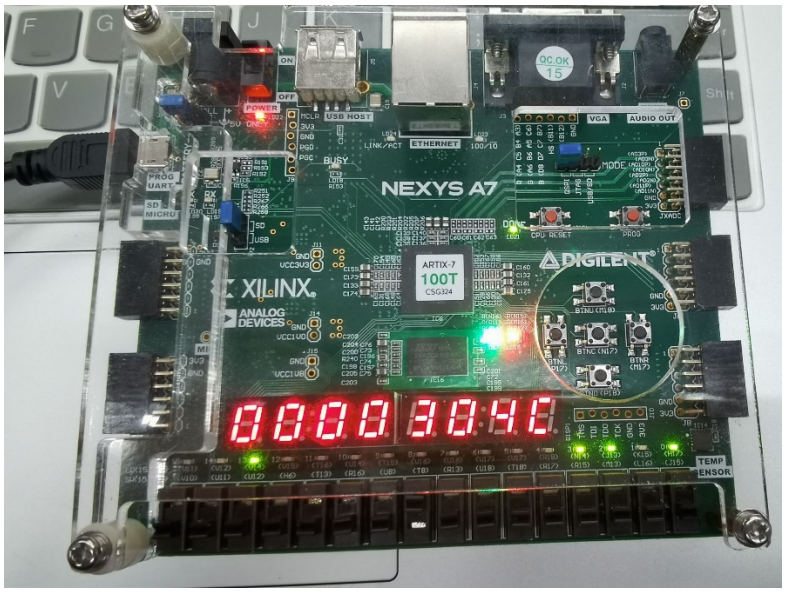

图 15

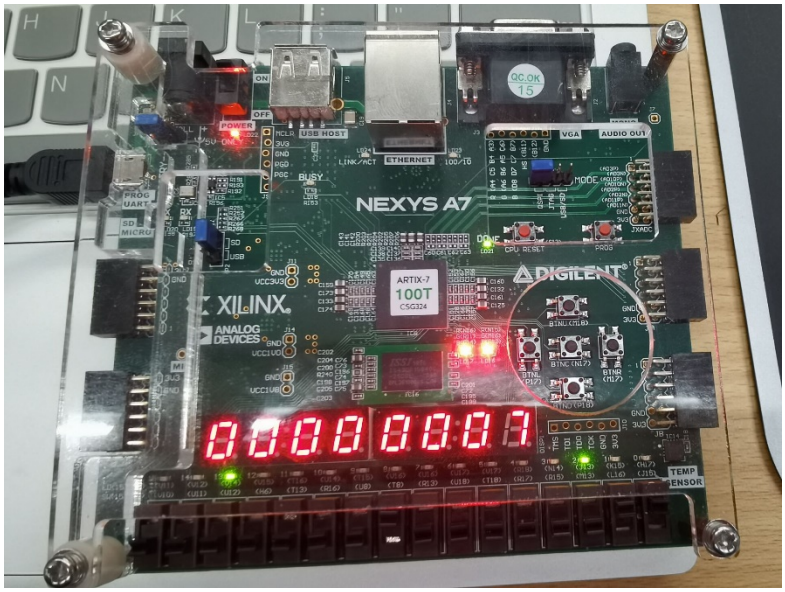

图 16

我完成本次实验所花费的时间，大概可以对半分，一半是写完其他所有内容，另一是寻找一个隐秘的 BUG。这个 BUG 来源于跳转指令+lw 指令的情况。根据我的代码逻辑，在 lw 执行到 ID 流水段时，会

将 lw 的下一条指令（此时在 IF 段）重新读一遍，具体实现是令 IF_pc 仍为 IF_pc,而不是 nxt_pc，再配合上一些清除段间寄存器的操作，即可实现 lw 后的指令暂停一周期。而如果是跳转指令+lw 指令的组合，lw 指令执行到 ID 段时，跳转指令执行到 EX 段，如果需要跳转，那么 EX 段的跳转指令就会修改 nxt_pc，来实现跳转。这时就会出现问题：跳转指令的跳转操作，是通过修改 nxt_pc 来实现，而此时处于 ID 段的 lw 指令，会阻止 IF_pc 被赋值为 nxt_pc，等 lw 的一个周期的暂停结束后，nxt_pc 又发生了变化，已经不是跳转的目标地址了，于是就出现了问题。

这个 BUG 比较隐秘，还是多亏了有 PDU，可以单步执行，然后查看各个寄存器以及 pc、IR 的值。这个 BUG 的修改很简单，如果要发生跳转，那么对 pc 赋值首先是 nxt_pc，然后才是 lw 的暂停一周期操作。核心代码如下：

```
else if(PCSrc) begin
    // 这里很重要，当分支指令的下一条是lw指令，并且应该跳转时，那就直接跳转
    // 如果没有这一步，那么lw指令就会讲原本的跳转给冲掉
    IF_pc <= nxt_pc;
end
else if(ID_IR[6:0]==7'b0000011) begin
    // lw, 需要停一个周期
    IF_pc <= IF_pc;
end
```

图 17

这样一个先后的顺序，解决了上述 BUG

【总结与思考】

1. 本次实验难度适中，流水线 CPU 可以直接基于已经实现的单周期 CPU 去做。
2. 本次实验收获较多，最重要的是可以深入到代码实现的具体细节上，体会流水线 cpu 是如何运作的。而且 verilog 是硬件编程语言，与实际的电路比较接近，所以其实是从电路的角度体会了流水线 cpu 的执行过程
3. 本次实验也帮助我更深入理解了各种相关，尤其是数据相关，也从代码层面体会到了数据定向是怎样一个操作。
4. 最后，本次实验给出的数据通路和控制器图并不全，缺少 jal、jalr、auipc 的数据通路和控制信号，建议之后的实现文档将这一部分进行补充。

【附录】

Lab5_q1.srcs 完整流水线的代码文件，包括设计文件和引脚约束文件