

《计算机组成原理》实验报告

年级、专业、班级	2021 级计算机科学与技术 05 班与 04 班	姓名	冯宇馨,胡鑫
实验题目	实验二处理器译码实验		
实验时间	2023 年 4 月 21 日	实验地点	A 主 404
实验成绩	优秀/良好/中等	实验性质	<input type="checkbox"/> 验证性 <input checked="" type="checkbox"/> 设计性 <input type="checkbox"/> 综合性
教师评价: <input type="checkbox"/> 算法/实验过程正确; <input type="checkbox"/> 源程序/实验内容提交; <input type="checkbox"/> 程序结构/实验步骤合理; <input type="checkbox"/> 实验结果正确; <input type="checkbox"/> 语法、语义正确; <input type="checkbox"/> 报告规范; 其他: <div>评价教师: 冯永</div>			
实验目的 (1)掌握单周期 CPU 控制器的工作原理及其设计方法。 (2)掌握单周期 CPU 各个控制信号的作用和生成过程。 (3)掌握单周期 CPU 执行指令的过程。 (4)掌握取指、译码阶段数据通路执行过程。			

报告完成时间: 2023 年 4 月 22 日

1 实验内容

1. PC。D 触发器结构,用于储存 PC(一个周期)。需实现 2 个输入,分别为 *clk*, *rst*, 分别连接时钟和复位信号;需实现 2 个输出,分别为 *pc*, *inst_ce*, 分别连接指令存储器的 *addra*, *ena* 端口。其中 *addra* 位数依据 coe 文件中指令数定义;
2. 加法器。用于计算下一条指令地址,需实现 2 个输入,1 个输出,输入值分别为当前指令地址 *PC*、32'h4;
3. Controller。其中包含两部分:
 - (a). *main_decoder*。负责判断指令类型,并生成相应的控制信号。需实现 1 个输入,为指令 *inst* 的高 6 位 *op*,输出分为 2 部分,控制信号有多个,可作为多个输出,也作为一个多位输出,具体参照参考指导书进行设计;*aluop*, 传输至 *alu_decoder*, 使 *alu_decoder* 配合 *inst* 低 6 位 *funct*, 进行 ALU 模块控制信号的译码。
 - (b). *alu_decoder*。负责 ALU 模块控制信号的译码。需实现 2 个输入,1 个输出,输入分别为 *funct*, *aluop*;输出位 *alucontrol* 信号。
 - (c). 除上述两个组件,需设计 *controller* 文件调用两个 decoder, 对应实现 *op*, *funct* 输入信号,并传入调用模块;对应实现控制信号及 *alucontrol*, 并连接至调用模块相应端口。
4. 指令存储器。使用 Block Memory Generator IP 构造。(参考指导书)
注意: Basic 中 Generate address interface with 32 bits 选项不选中; PortA Options 中 Enable Port Type 选择为 Use ENA Pin
5. 时钟分频器。将板载 100Mhz 频率降低为 1hz, 连接 PC、指令存储器时钟信号 *clk*。(参考数字逻辑实验)
注意: Xilinx Clocking Wizard IP 可分的最低频率为 4.687Mhz, 因而只能使用自实现分频模块进行分频

2 实验设计

2.1 控制器 (Controller)

2.1.1 功能描述

控制器输出的控制信号,用于控制器件的使能和多路选择器的选择。

2.1.2 接口定义

如下图所示

表 1: 接口定义

信号名	方向	位宽	功能描述
op	input	6-bit	指令的 31 到 26 位
funct	input	6-bit	指令的 5 到 0 位
zero	input	1-bit	是 alu 的输出,用于 branch 指令是否跳转的决定因素
alucontrol	output	3-bit	alu 控制信号,代表不同的运算类型
memwrite	output	1-bit	是否需要写数据存储器
memtoreg	output	1-bit	回写的数据来自于 alu 计算的结果/存储器读取的数据
branch	output	1-bit	执行 branch 语句的使能
jump	output	1-bit	执行 jump 语句的使能
alusrc	output	1-bit	送入 alu 的值是立即数的 32 位扩展/寄存器堆读取的值
regdst	output	1-bit	写入寄存器堆的地址是 rt 还是 rd,0 为 rt,1 为 rd
regwrite	output	1-bit	是否需要写寄存器堆
pcsrc	output	1-bit	下一个 PC 值是 PC+4/跳转的新地址

2.1.3 逻辑控制

alucontrol: alu 的控制信号是由指令的最后六位的 funct 以及由指令前六位的 op 决定的 aluop 所共同决定的, 当 aluop=00 时, 不论 funct 取值, alucontrol 恒为 010, 执行 sw, lw, addi 指令; 当 aluop=01 时, 不论 funct 取值, alucontrol 恒为 110, 执行 beq 等指令; 当 aluop=10 时, 根据 funct 执行不同的 R-type 指令。其余逻辑控制用表格表示, 显示为表二

表 2: 逻辑控制

instruction	op5:0	regwrite	regdst	alusrc	branch	memwrite	memtoreg	pcsrc	j
R-type	000000	1	1	0	0	0	0	0	0
lw	100011	1	0	1	0	0	1	0	0
sw	101011	0	x	0	1	0	x	0	0
beq	000100	0	x	0	1	0	x	1	0
addi	001000	1	0	1	0	0	0	0	0
j	000010	0	x	x	x	0	x	0	1

2.2 存储器 (Block Memory)

类型选择

Component Name
blk_mem_gen_0

Basic
Port A Options
Other Options
Summary

Interface Type
Native
Generate address interface with 32 bits
Memory Type
Single Port RAM
Common Clock

ECC Options
ECC Type
No ECC
Error Injection Pins
Single Bit Error Injection

Write Enable
Byte Write Enable
Byte Size (bits)
9

Algorithm Options
Defines the algorithm used to concatenate the block RAM primitives. Refer datasheet for more information.
Algorithm
Minimum Area
Primitive
8kx2

参数设置

Component Name blk_mem_gen_0

Basic Port A Options Other Options Summary

Memory Size

Write Width 32 Range: 1 to 4608 (bits)

Read Width 32

Write Depth 2048 Range: 2 to 1048576

Read Depth 2048

Operating Mode Write First Enable Port Type Use ENA Pin

Port A Optional Output Registers

☐ Primitives Output Register ☐ Core Output Register

☒ SoftECC Input Register ☒ REGCEA Pin

Port A Output Reset Options

☐ RSTA Pin (set/reset pin) Output Reset Value (Hex) 0

☐ Reset Memory Latch Reset Priority CE (Latch or Register Enable)

READ Address Change A

☐ Read Address Change A

3 实验过程记录

3.1 问题 1:IP 核的实例化调用

问题描述:开始不知道如何使用 IP 核,以及 IP 核的输入输出不清除。

解决方法:直接调用 IP 核,并查看 IP 的 veo 文件,了解 ram 的端口定义,然后根据端口进行输入。

3.2 问题 2:打印输出到控制台

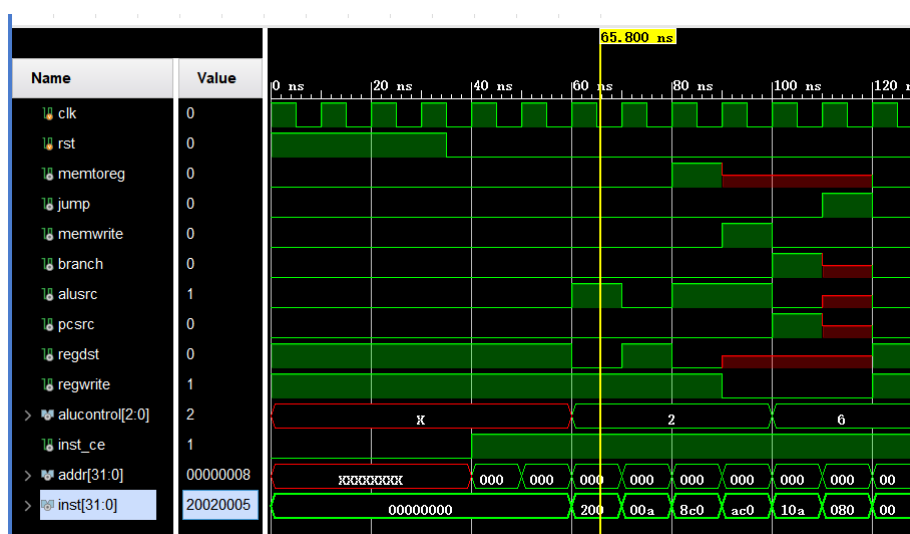
问题描述:在实验仿真的时候,不知道如何将输出结果打印到控制台。

解决方法:通过学习,在每次运行时,使用'display' 进行打印输出

4 实验结果及分析

实验依次输出 coe 文件中存放的指令,并在控制台中打印指令与控制信号。

4.1 仿真图



4.2 控制台输出

```
Tcl Console x Messages Log
Instruction: 00000000, astoreg: 0, aawrite: 0, pcsrc: 0, alusrc: 0, regdst: 1, regwrite: 1, jump: 0, branch: 0, alucontrol: xxx
Instruction: 00000000, astoreg: 0, aawrite: 0, pcsrc: 0, alusrc: 0, regdst: 1, regwrite: 1, jump: 0, branch: 0, alucontrol: xxx
Instruction: 20020005, astoreg: 0, aawrite: 0, pcsrc: 0, alusrc: 1, regdst: 0, regwrite: 1, jump: 0, branch: 0, alucontrol: 010
Instruction: 00a42820, astoreg: 0, aawrite: 0, pcsrc: 0, alusrc: 0, regdst: 1, regwrite: 1, jump: 0, branch: 0, alucontrol: 010
Instruction: 8c020050, astoreg: 1, aawrite: 0, pcsrc: 0, alusrc: 1, regdst: 0, regwrite: 1, jump: 0, branch: 0, alucontrol: 010
Instruction: ac020054, astoreg: x, aawrite: 1, pcsrc: 0, alusrc: 1, regdst: x, regwrite: 0, jump: 0, branch: 0, alucontrol: 010
Instruction: 10a7000c, astoreg: x, aawrite: 0, pcsrc: 1, alusrc: 0, regdst: x, regwrite: 0, jump: 0, branch: 1, alucontrol: 110
Instruction: 08000013, astoreg: x, aawrite: 0, pcsrc: x, alusrc: x, regdst: x, regwrite: 0, jump: 1, branch: x, alucontrol: 110
```

当 instruction=20020005 时, 执行 addi 指令, 所以 alucontrol 为 010, aluwrite 和 regwrite 为 1, 其他为 0;

当 instruction=00a42820 时, 执行 add 指令, 所以 alucontrol 应为 010, regdst 和 regwrite 为 1, 其他都为 0;

当 instruction=8c020050 时, 执行 lw 指令, 所以 alucontrol 为 010, memtoreg, regwrite, alusrc 为 1, 其他都为 0;

当 instruction=ac020054 时, 执行 sw 指令, 所以 alucontrol 为 010, memwrite 和 alusrc 为 1; memtoreg 和 regdst 为 x, 其他为 0;

当 instruction=10a7000c 时, 执行 beq 指令, 所以 alucontrol 为 110, branch 为 1; 由于本次实验中无法得出 alu 模块的 zero 端口值, 所以我们假定默认为 1, 此点在后续实验中将被改进, 因此 pcsrc 为 1, memtoreg 和 regdst 为 x, 其他为 0;

当 instruction=08000013 时, 执行 j 指令, 所以 j 为 1, 除了 memwrite 和 regwrite 为 0 外, 其余都不定。

综上所述, 仿真正确。

A Controller 代码

```
module Controller(  
    input [5:0] op,  
    input [5:0] funct,  
    input zero,  
    output [2:0] alucontrol,  
    output memtoreg,  
    output memwrite,  
    output branch,  
    output alusrc,  
    output regdst,  
    output regwrite,  
    output jump,  
    output pcsrc//下一个 PC 值是 PC+4/跳转的新地址  
);  
    wire [1:0] aluop;  
    main_decde u1(.op(op),.aluop(aluop),.memtoreg(memtoreg),.memwrite(memwrite),.  
        branch(branch),.jump(jump),.alusrc(alusrc),.regdst(regdst),.regwrite(  
            regwrite));  
    alu_decode1 u2(.funct(funct),.aluop(aluop),.alucontrol(alucontrol));  
    assign pcsrc=zero&branch;  
endmodule
```

```
module main_decde(  
    input [5:0] op,//指令inst 的高 6 位 op  
    output memtoreg,//回写的数据来自于 ALU 计算的结果0/存储器读取的数据1  
    output memwrite,//是否需要写数据存储器  
    output regdst,//写入寄存器堆的地址是 rt 还是 rd,0 为 rt,1 为 rd  
    output regwrite,//是否需要写寄存器堆  
    output alusrc,//送入 ALU B 端口的值是立即数的 32 位扩展1/寄存器堆读取的值0  
    output branch,//是否为 branch 指令,且满足 branch 的条件  
    output jump,//是否为 jump 指令  
    output [1:0] aluop//ALU 控制信号,代表不同的运算类型  
);  
    reg [8:0] con;  
    assign {regwrite,regdst,alusrc,branch,memwrite,memtoreg,aluop,jump}=con;  
    always@(*)  
    begin  
        case(op)  
            6'b000000: con<=9'b110000100;//R-type  
            6'b100011: con<=9'b101001000;//lw  
            6'b101011: con<=9'b0X101X000;//sw  
            6'b000100: con<=9'b0X010X010;//beq  
            6'b001000: con<=9'b101000000;//addi  
            6'b000010: con<=9'b0XXX0XXX1;//j  
            default: con<=8'b00000000;  
        endcase  
    end
```

```

endmodule

module alu_decode1(
    input  [1:0] aluop ,
    input  [5:0] funct ,
    output [2:0] alucontrol
);
    reg [2:0] a;
    always@(*)
    begin
        if (aluop==2'b00) //sw, lw, addi
            begin
                a<=3'b010;
            end
        else if (aluop==2'b01) //beq
            begin
                a<=3'b110;
            end
        else if (aluop==2'b10) //R- type
            begin
                if (funct==6'b100000) a<=3'b010; //add
                else if (funct==6'b100010) a<=3'b110; //sub
                else if (funct==6'b100100) a<=3'b000; //and
                else if (funct==6'b100101) a<=3'b001; //or
                else if (funct==6'b101010) a<=3'b111; //slt
            end
        end
        assign alucontrol=a;
    end
endmodule

```