《计算机组成原理》实验报告

年级、专业、班级	2022 级信息安全 01 班,2022 级信息安全 02 班	**姓名**	杨小艺,姚凡
实验题目	实验二处理器译码实验		
实验时间	2024 年 4 月 27 日	**实验地点**	DS1410
实验成绩	优秀/良好/中等	**实验性质**	□验证性 ☑设计性 □综合性

教师评价：

□算法/实验过程正确; □源程序/实验内容提交; □程序结构/实验步骤合理;

□实验结果正确; □语法、语义正确; □报告规范;

其他:

评价教师: 谭玉娟

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: 2024 年 5 月 10 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。
   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

单周期 CPU 中控制器的功能是根据指令的操作码解码并生成相应的控制信号，以驱动 CPU 的各个组件（如寄存器、算术逻辑单元等）执行指令的各个阶段（如取指、译码、执行、访存、写回等），以实现指令的正确执行。

具体实现为：控制器输入指令 IR 的 opcode 和 funct 码，输出 memtoreg、memwrite、pcsrc、alusrc、regdst、regwrite、branch、jump 和 alucontrol 信号。实验根据 funct 字段实现 R、lw、sw、beq、addi 和 j 型指令的译码。Controller 包含 Main decoder 和 ALU decoder，因此需分模块编写，Controller 顶层调用，Main decoder 输入 opcode 字段，输出 memtoreg、memwrite、pcsrc、alusrc、regdst、regwrite、branch、jump 和 aluop，ALU decoder 输入 aluop 和 funct 字段，输出 alu control。

2.1.2 接口定义

表 1: 接口定义模版

信号名	方向	位宽	功能描述
funct	input	5-bits	指令操作码, 即 IR[5:0]
op	input	5-bits	判断指令类型, 即 IR[31:26]
clk	input	1-bit	时钟信号
rst	input	1-bit	复位信号
regwrite	output	1-bit	控制寄存器写入操作
regdst	output	1-bit	选择目标寄存器的地址源
alusrc	output	1-bit	选择 ALU 的第二个操作数
pcsrc	output	1-bit	控制 PC 的更新方式
memwrite	output	1-bit	控制内存写操作
memtoreg	output	1-bit	选择数据的来源
jump	output	1-bit	控制程序跳转操作
aluctrl	output	3-bits	控制 ALU 的操作
branch	output	1-bit	是否满足 branch 指令的跳转条件
alucontrol	output	1-bit	ALU 控制信号, 只要求实现 R 型指令的 add、sub、and、or、slt

2.1.3 逻辑控制

控制器根据指令的操作码和当前的系统状态生成相应的控制信号, 以协调 CPU 内部各个组件的操作, 确保指令能够正确执行。工作流程包括:

1. 指令解码: 控制器从指令中提取操作码字段, 并根据操作码识别出指令的类型和所需的操作。
2. 控制信号生成: 根据当前的系统状态和指令的操作码, 控制器生成一系列控制信号, 如寄存器写使能信号(regwrite)、ALU 控制信号(aluctrl)、内存写使能信号(memwrite)等。这些控制信号用于控制各个组件的操作, 例如寄存器、ALU、内存等。
3. 控制信号传递: 控制器将生成的控制信号通过控制总线(Control Bus)发送给相关的组件, 以通知它们应该执行何种操作。
4. 状态更新: 控制器还可能需要根据当前的指令执行情况和系统状态更新一些内部的状态寄存器, 以便在下一条指令执行时正确生成相应的控制信号。

编写顶层模块 Controller, 其中包含 Main decoder 和 ALU decoder, Main decoder 输入 IR[31:26] 作为 opcode, 输出 memtoreg、memwrite、pcsrc、alusrc、regdst、regwrite、branch、jump 和 aluop, ALU decoder 输入 aluop 和 funct 字段, 输出 alu control。

Main decoder 根据输入的 opcode 判断指令的类型，用 case 语句译码得到各元件的控制信号。

2.2 存储器 (Block Memory)

- coe 文件

```
memory initialization radix = 16;
memory initialization vector =
20020005,
00a42820,
8c020050,
ac020054,
10a7000c,
08000013,
```

图 1: coe 文件

- 构造 IP 核

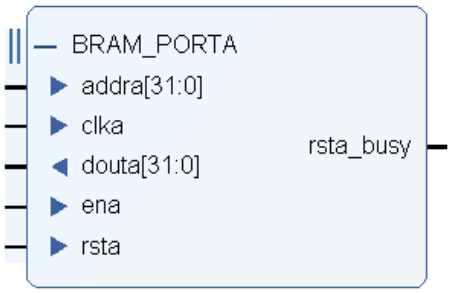

图 2: IP 核

3 实验过程记录

3.1 指令集设计和编码

根据 CPU 的需求和所支持的指令集架构，我们定义了适当的指令格式和操作码，并为每个指令分配了相应的唯一二进制编码，为后续的控制信号生成提供了基础。

```
addi $2, $0, 5       # initialize $2=5
add $5, $5, $4       # $5 = 4 + 7 = 11
lw $2, 80($0)        # $2 = [80] = 7
sw $2, 84($0)        # write adr 84=7
beq $5, $7, end      # shouldn't be taken
j end                # should be taken
```

图 3: 指令编码

3.2 实现控制器的组合逻辑电路

输入操作码，操作码通过设计的逻辑电路来根据指令类型和系统状态生成相应的控制信号。这些控制信号包括寄存器写使能信号、ALU 控制信号、内存写使能信号等，用于协调 CPU 内部各个组件的操作。

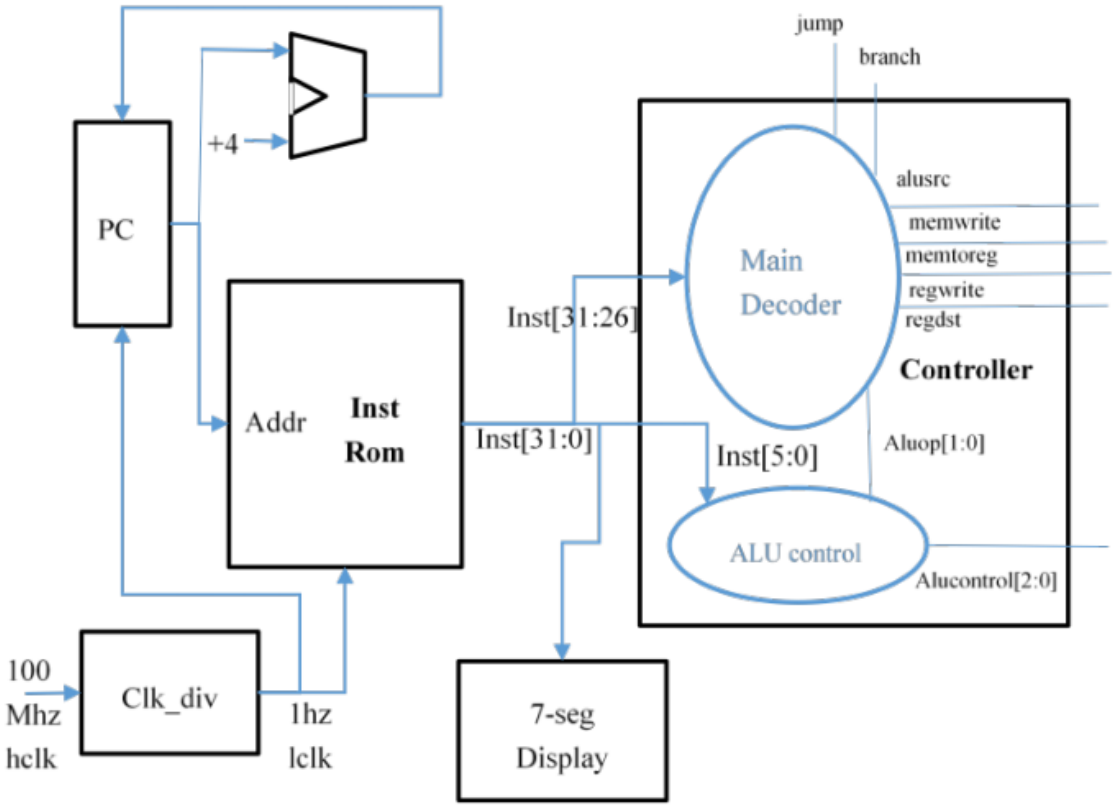

图 4: 取址译码原理图

opcode	aluop	operation	funct	alu function	alu control
lw	00	Load word	XXXXXX	Add	010
sw	00	Store word	XXXXXX	Add	010
beq	01	Branch equal	XXXXXX	Subtact	110
R-type	10	Add	100000	qdd	010
		subtract	100010	Subtract	110
		and	100100	And	000
		or	100101	Or	001
		set-on-less-than	101010	SLT	111

图 5: 译码控制信号

instruction	op5:0	regwrite	regdst	alusrc	branch	memWrite	memtoReg	aluop1:0
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00
j	000010	0	X	X	X	0	X	XX

图 6: 控制信号译码

3.3 进行功能验证和调试

我们通过编写测试程序模拟了不同的指令序列，观察 CPU 的行为是否符合预期。在发现问题或错误后，我们进行了逐步调试和修复，以确保控制器能够正确地生成适当的控制信号，在经过多次调试和修改后，控制器最终能够正确地生成正确的的控制信号。

3.4 完成控制器的实验

通过设计和实现控制器，我们成功实现了一个能够正确执行指令的 CPU，并为后续的 CPU 设计和优化奠定了基础。

3.5 实验过程中遇见的问题

问题描述 1：仿真一直出现错误

问题解决：在通过仿真验证和下板验证时，需要在 verilog 代码中更改时钟信号，下板时需要使用时钟分频信号，而仿真时则不需要，否则会出现仿真无效的情况。

问题描述 2：仿真图中，取出的两个指令之间出现时延

问题解决：在存储器 IP 引入时，ROM 设置中取消勾选 RESETMEMORYLATCH 的选项

问题描述 3：两条指令之间存在 4 个时钟周期的空白，IR 和指令全为 0

问题解决：重新设置初始值，时钟先上升沿改变调用顺序

问题描述 4：仿真图中 IR 无法取出指令且一直为高阻态

问题解决：在 ALU_DECODER 模块中只接入接口并未进行操作，数据流断流。

4 实验结果及分析

4.1 仿真图

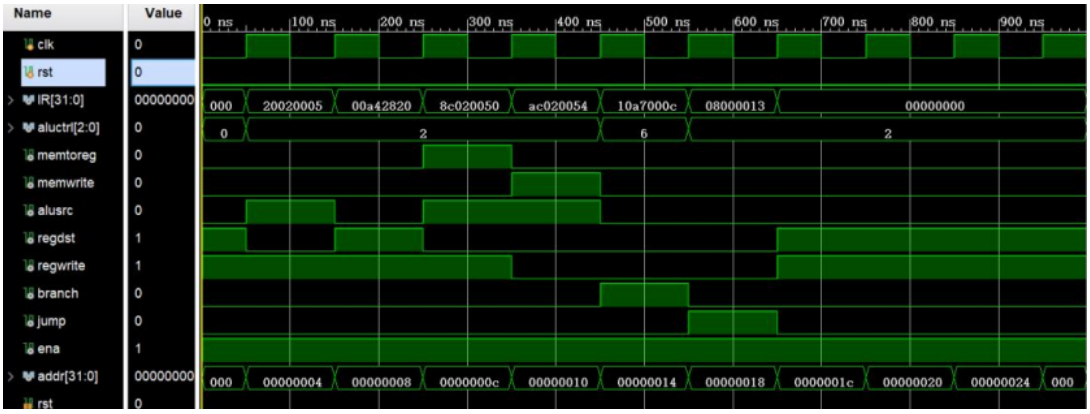

图 7: 仿真图

4.2 控制台输出

```
Time resolution is 1 ps
source test_bench.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a wave window go to 'File->New Waveform Configuration' or type 'cr
#   }
# }
# run 1000ns
 Block Memory Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator module test_bench.top.your_instance_name.inst.native_mem_mapped_module.blk_mem_gen_v8_4_2_inst is using a behavioral model for simulation which will not precisely model memor
INFO: [USF-XSim-96] XSim completed. Design snapshot 'test_bench_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:01 ; elapsed = 00:00:11 . Memory (MB): peak = 2836.551 ; gain = 0.000
```

图 8: 控制台输出

4.3 上板验证

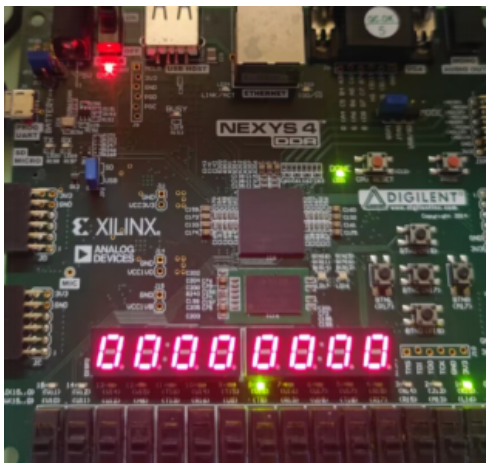

图 9: 初始状态

A Controller 代码

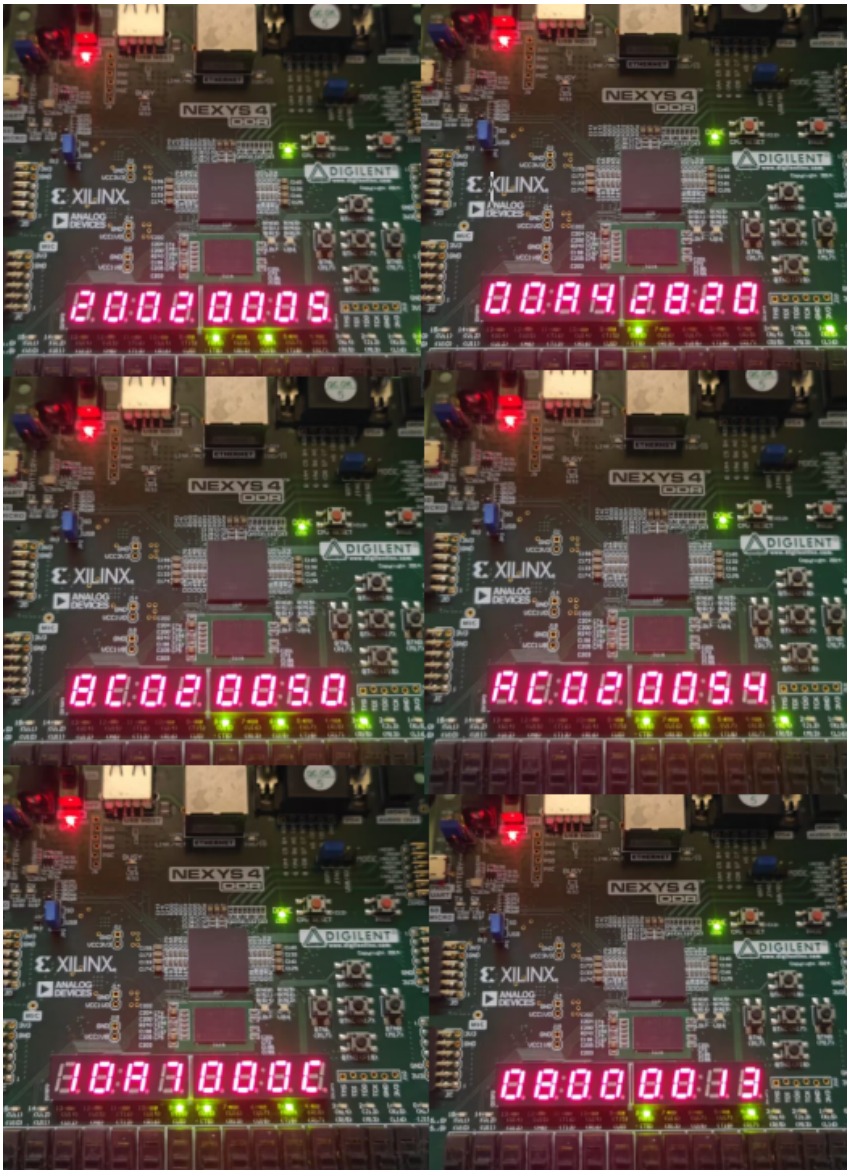

图 10: 读取.coe 文件内容

```
————————————-/*top*/————————————-
    module top(
    input wire clk,rst,
    output wire[2:0] aluctrl,
    output wire memtoreg,memwrite,alusrc,regdst,regwrite,branch,jump,
    output wire[6:0] seg,
    output wire[7:0] ans
    );
    wire clk_division;
    clk_div gate1(clk,clk_division);
    wire ena;
    wire[31:0] addr;                    //32bits 指令的地址（0f023）
    pc gate2(clk_division,rst,addr,ena);
    wire[31:0] IR;                      //32bits 指令
    blk_mem_gen_0 gate3(
    .addra(addr»2),
    .clka(clk),
    .ena(ena),
    .rsta(rst),
    .douta(IR)
    );  //传入 ROM 的是字节地址,因此要右移 2 位
    controller gate4(IR[31:26],IR[5:0],aluctrl,memtoreg,memwrite,alusrc,regdst,regwrite,branch,jump);
display gate5(clk,rst,IR,seg,ans);  //display 中已经分频
    endmodule
    ————————————/*clk_div*/————————————
    module clk_div(
    input clk,
```

```
output clk_division
);
reg[27:0]cnt=0;
assign clk_division=cnt[26];
always@(posedge clk)
begin
cnt=cnt+1;
end
endmodule
————————————-/*pc*/————————————————
module pc(
input clk,rst,
output reg[31:0] addr=0,    //连接指令寄存器的输入地址端口
output wire ena              //连接指令寄存器的使能端口
);
reg [31:0] cnt=0;            //地址计数,不断 +4
assign ena=(addr<1024)?1:0; //ROM 设置的端口 Depth 为 1024, 因此 >1024 的地址为无效地址
//    assign ena =1;
always@(posedge clk)begin
if(rst)begin
cnt=0;
addr=cnt;
end
else begin
cnt=cnt+4;
addr=cnt;
```

```
end
end
endmodule
————————————/*controller*/————————————-
module controller(
input wire[5:0] op,
input wire[5:0] funct,
output wire[2:0] aluctrl,
output wire memtoreg,memwrite,alusrc,regdst,regwrite,branch,jump
);
wire[1:0] aluop;
maindec
gate1(op,aluop,memtoreg,memwrite,alusrc,regdst,regwrite,branch,jump);
aludec gate2(aluop,funct,aluctrl);
endmodule
————————————/*maindec*/————————————-
module maindec(
input [5:0]op,
output reg [1:0] aluop,
output reg memtoreg ,memwrite,alusrc,regdst,regwrite,branch,jump
);
always @(*)
begin
case (op)
6'b000000:begin    //R
{ regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump}=7'b1100000;
aluop=2'b10;
```

```
end
6’b100011:begin    //lw
{regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump}=7’b1010010;
aluop=2’b00;
end
6’b101011:begin     //sw
{regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump}=7’b1010010;
aluop=2’b00;
end
6’b000100:begin     //beq
{regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump}=7’b0001000;
aluop=2’b01;
end
6’b001000:begin    //addi
{regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump}=7’b1010000;
aluop=2’b00;
end
6’b000010:begin     //jump
{regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump}=7’b0000001;
aluop=2’b00;
end
default:begin
{regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump}=’b0;
aluop=2’b00;
end
endcase
end
```

```
endmodule
————————————/*aludec*/————————————
module aludec(
input wire[1:0] aluop,
input wire[5:0] funct,
output reg [2:0] aluctrl
);
always@(*)begin
case(aluop)
2’b00:aluctrl=010;                //lw、sw
2’b01:aluctrl=110;                //beq
2’b10:begin                       //R 型指令
case(funct)
6’b100000:aluctrl=010;  //add
6’b100010:aluctrl=110;  //sub
6’b100100:aluctrl=000;  //and
6’b100101:aluctrl=001;  //or
6’b101010:aluctrl=111;  //slt
endcase
end
default:aluctrl=’b0;
endcase
end
endmodule
```