

《计算机组成原理》实验报告

年级、专业、班级	2023 级计算机科学与技术 06 班 2023 级计算机科学与技术 04 班 2023 级计算机科学与技术 04 班	姓名	饶格奇 刘雨霜 李隆征
实验题目	实验二处理器译码实验		
实验时间	2025 年 4 月 27 日	实验地点	DS1410
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：任骛</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: 2025 年 4 月 27 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

控制器的任务就是依据指令的 opcode 和 funct 字段译码,生成所有使能与多路选择信号,驱动数据通路各模块在一个时钟周期内正确完成对应指令的执行。

2.1.2 接口定义

表 1: maindec 接口定义

信号名	方向	位宽	功能描述
Inst_part_31_26	Output	6-bit	op 字段
jump	Input	1-bit	跳转
branch	Output	1-bit	是否为 branch 指令且满足 branch 的条件
alusrc	Output	1-bit	控制 ALU 第二操作数的来源。
memwrite	Output	1-bit	控制是否要写入数据存储器
memtoreg	Output	1-bit	控制是否要写入寄存器堆
regwrite	Output	1-bit	指示寄存器堆是否能写入
regdst	Output	1-bit	决定写回寄存器的目标号来源
ALop	Output	2-bit	提供给 ALU 控制器的运算类型提示，用于根据指令大类快速确定 ALU 应执行的基本操作

表 2: aludec 接口定义

信号名	方向	位宽	功能描述
func	Input	6-bit	在 R-type 指令中用于具体指定 ALU 要执行的运算功能
Alop	Input	2-bit	提供给 ALU 控制器的运算类型提示，用于根据指令大类快速确定 ALU 应执行的基本操作
Alcontrol	Output	3-bit	由 ALU 译码器输出的控制信号，精确指示 ALU 执行哪种具体运算

2.1.3 逻辑控制

- R 型指令 (opcode=000000): 设定控制信号为执行寄存器运算。
- lw (opcode=100011): 设定为读取内存, 并回写到寄存器。
- sw (opcode=101011): 设定为寄存器数据写入内存, 不进行寄存器回写。
- beq (opcode=000100): 设定为进行分支判断, 使用减法 (ALU 减运算)。
- addi (opcode=001000): 设定为立即数加法运算。
- jump (opcode=000010): 设定为无条件跳转。

ALU 译码器 (aludec) 根据主译码器给出的 aluop 信号和指令低 6 位 (inst[5:0]) 中的

`funct` 字段进一步确定具体的 ALU 操作类型,生成精确的 ALU 控制信号 (`alucontrol`):

- 当 `aluop=00` (例如 `lw`, `sw`, `addi`):执行加法运算。
- 当 `aluop=01` (例如 `beq`):执行减法运算。
- 当 `aluop=10` (R 型指令):根据 `funct` 字段选择执行加法、减法、与、或、比较等具体操作。

此外,为了简化数据通路,本实验未额外添加复杂的优化逻辑,仅采用标准单周期控制方案。

2.2 存储器 (Block Memory)

存储器的功能是存储 CPU 执行的指令,并根据地址(PC)输出对应地址的指令内容。

2.2.1 类型选择

存储器类型选择 Single Port ROM。

2.2.2 参数设置

端口宽度设置为 32-bit,加载 `coe` 文件。

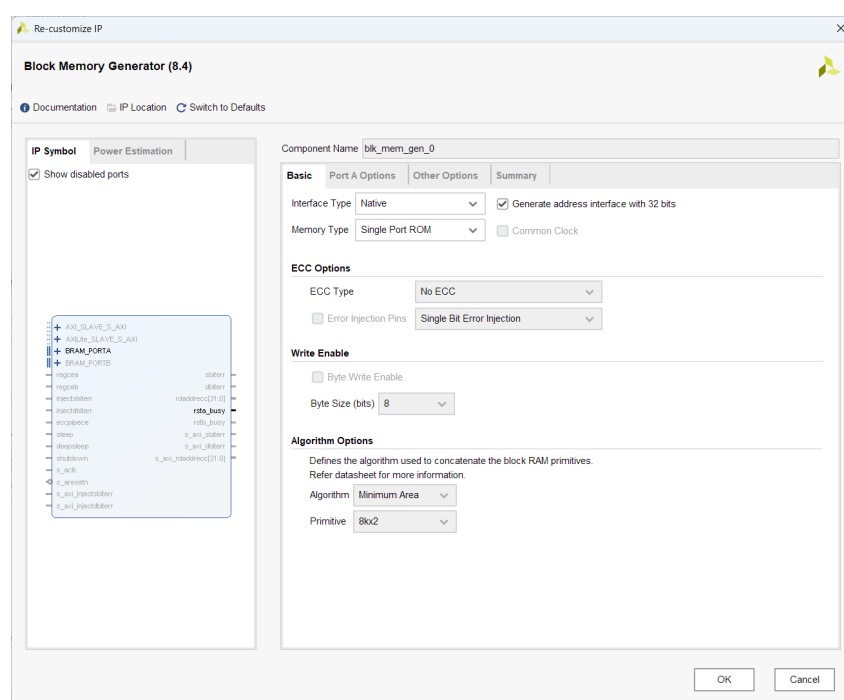


图 1: 存储器 Basic 界面

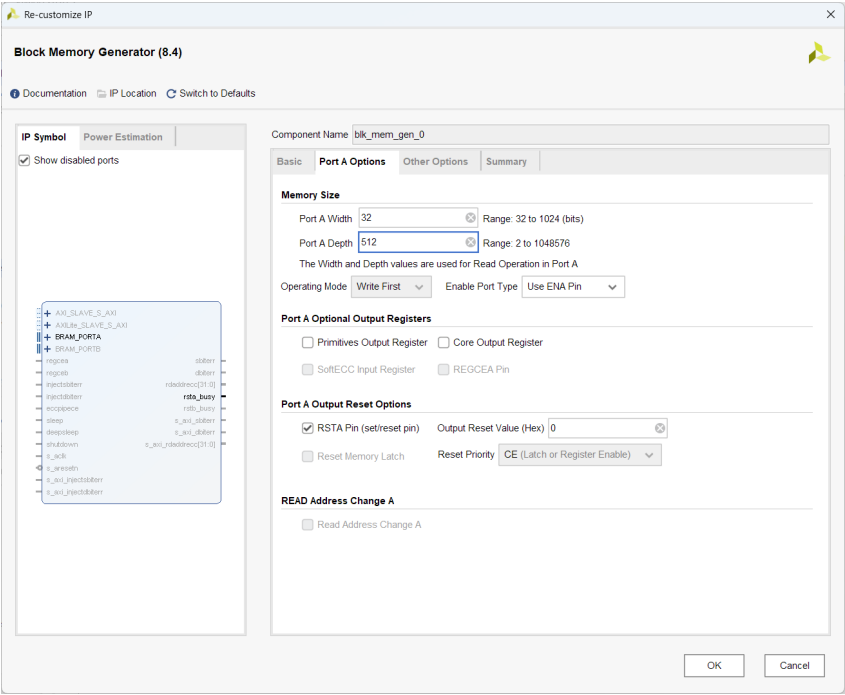


图 2: 存储器 PartAOptions 界面

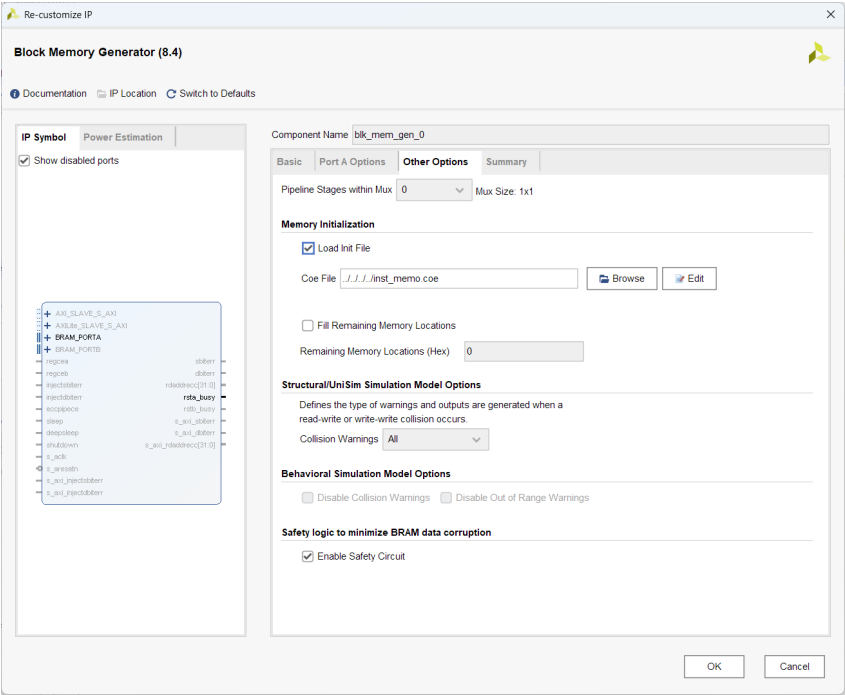


图 3: 存储器 OtherOptions 界面

3 实验过程记录

3.1 问题 1:Main Decoder 的实现

问题描述:Main Decoder 需要根据指令高 6 位 opcode,快速判定指令类型(R-type、lw、sw、beq、addi、jump 等),并正确生成多个控制信号,指导后续数据通路的行为。需要同时输出控制 ALU 的粗略操作类型 aluop,供后级 ALU Decoder 精细译码。

解决方案:采用组合逻辑 case 块,根据不同 opcode 设置一组打包的控制信号(如 regwrite、memtoreg、memwrite、branch、jump 等),并输出对应的 aluop 类型编码。对于 jump 指令,由于不涉及 ALU 运算,特殊处理 aluop 赋值为无关。整体保证不同指令类型生成的信号组合正确无冲突。

3.2 问题 2:ALU Decoder 的实现

问题描述:ALU Decoder 需要根据 aluop(主译码器传入)和指令低 6 位 funct 字段(仅对 R-type 有效),确定 ALU 执行的具体运算类型(如加法、减法、逻辑与、或、比较等),输出准确的 alucontrol 控制信号。

解决方案:采用组合逻辑优先判断 aluop 值:当为 Load/Store 类型时直接输出加法控制码,当为 Branch 类型时直接输出减法控制码;当为 R-type 类型时根据 funct 字段进一步精确匹配 ALU 操作(如加、减、与、或、SLT),并输出对应的 alucontrol 编码。若遇非法或未知 funct,输出默认非法操作码(如 3'bxxx 或其他处理)。

4 实验结果及分析

4.1 仿真图

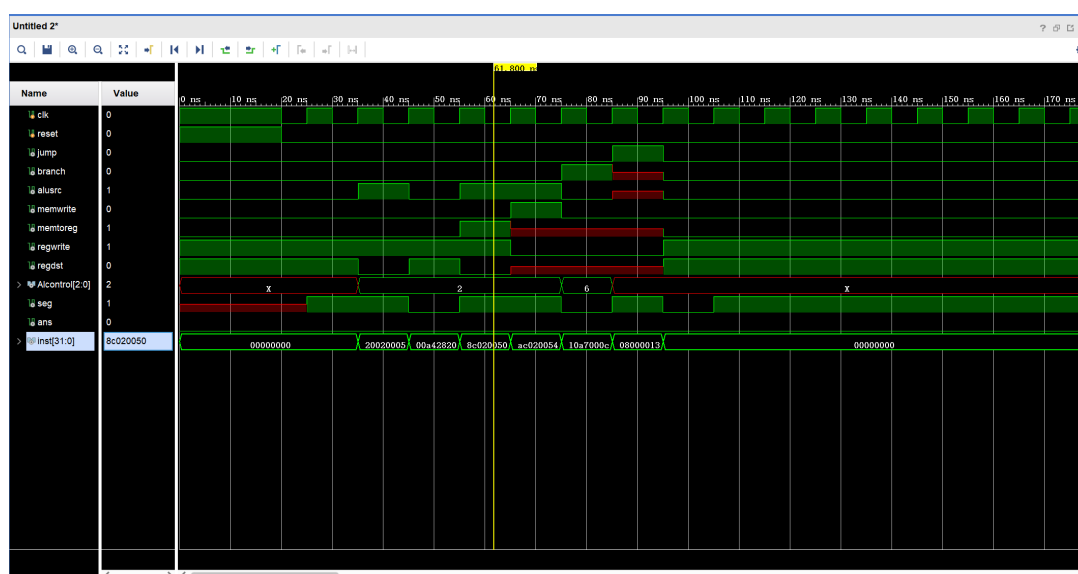
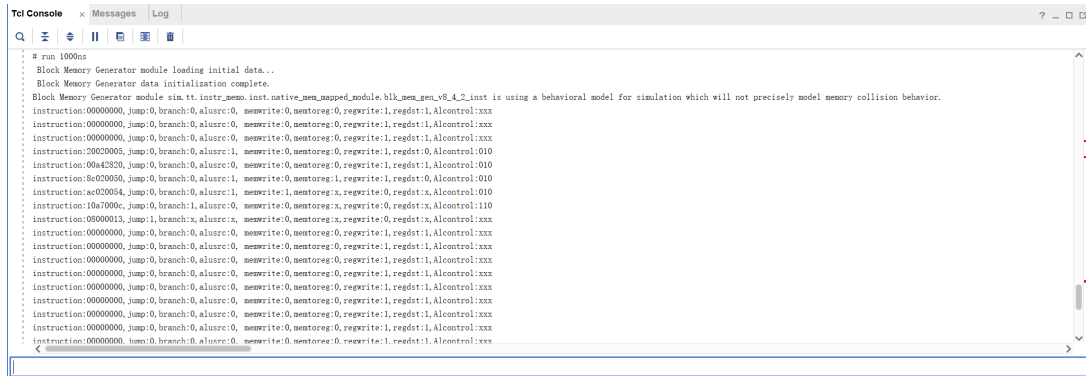


图 4: 仿真结果

4.2 控制台输出



```
# run 1000ns
Block Memory Generator module loading initial data...
Block Memory Generator data initialization complete.
Block Memory Generator module via tt_instr_mem0_inst.native_mem_mapped_module.blk_mem_gen_v8_4_2_inst is using a behavioral model for simulation which will not precisely model memory collision behavior.
Instruction:00000000, jump:0, branch:0, alusrc:0, memwrite:0, memtoreg:0, regwrite:1, regdst:1, Alcontrol:xxx
Instruction:00000000, jump:0, branch:0, alusrc:0, memwrite:0, memtoreg:0, regwrite:1, regdst:1, Alcontrol:xxx
Instruction:00000000, jump:0, branch:0, alusrc:0, memwrite:0, memtoreg:0, regwrite:1, regdst:1, Alcontrol:xxx
Instruction:00000000, jump:0, branch:0, alusrc:1, memwrite:0, memtoreg:0, regwrite:1, regdst:0, Alcontrol:010
Instruction:00423250, jump:0, branch:0, alusrc:0, memwrite:0, memtoreg:0, regwrite:1, regdst:1, Alcontrol:010
Instruction:0c020050, jump:0, branch:0, alusrc:1, memwrite:0, memtoreg:1, regwrite:1, regdst:0, Alcontrol:010
Instruction:ac020054, jump:0, branch:0, alusrc:1, memwrite:1, memtoreg:x, regwrite:0, regdst:x, Alcontrol:010
Instruction:10a7000c, jump:0, branch:1, alusrc:0, memwrite:0, memtoreg:x, regwrite:0, regdst:x, Alcontrol:110
Instruction:08000013, jump:1, branch:x, alusrc:x, memwrite:0, memtoreg:x, regwrite:0, regdst:x, Alcontrol:xxx
Instruction:00000000, jump:0, branch:0, alusrc:0, memwrite:0, memtoreg:0, regwrite:1, regdst:1, Alcontrol:xxx
Instruction:00000000, jump:0, branch:0, alusrc:0, memwrite:0, memtoreg:0, regwrite:1, regdst:1, Alcontrol:xxx
Instruction:00000000, jump:0, branch:0, alusrc:0, memwrite:0, memtoreg:0, regwrite:1, regdst:1, Alcontrol:xxx
Instruction:00000000, jump:0, branch:0, alusrc:0, memwrite:0, memtoreg:0, regwrite:1, regdst:1, Alcontrol:xxx
Instruction:00000000, jump:0, branch:0, alusrc:0, memwrite:0, memtoreg:0, regwrite:1, regdst:1, Alcontrol:xxx
Instruction:00000000, jump:0, branch:0, alusrc:0, memwrite:0, memtoreg:0, regwrite:1, regdst:1, Alcontrol:xxx
Instruction:00000000, jump:0, branch:0, alusrc:0, memwrite:0, memtoreg:0, regwrite:1, regdst:1, Alcontrol:xxx
Instruction:00000000, jump:0, branch:0, alusrc:0, memwrite:0, memtoreg:0, regwrite:1, regdst:1, Alcontrol:xxx
Instruction:00000000, jump:0, branch:0, alusrc:0, memwrite:0, memtoreg:0, regwrite:1, regdst:1, Alcontrol:xxx
```

图 5: 控制台输出

A Controller 代码

A.1 controller

```
'timescale 1ns / 1ps
module controller(
input [5:0] Inst_part_31_26,
input [5:0] Inst_part_5_0,
output wire jump, branch, alusrc, memwrite, memtoreg, regwrite, regdst,
output wire [2:0] Alcontrol
);

wire [1:0] Alop;

maindec MD(
.Inst_part_31_26(Inst_part_31_26),
.jump(jump),
.branch(branch),
.alusrc(alusrc),
.memwrite(memwrite),
.memtoreg(memtoreg),
.regwrite(regwrite),
.regdst(regdst),
.Alop(Alop)
);

aludec AL(
.Inst_part_5_0(Inst_part_5_0),
.Alop(Alop),
.Alcontrol(Alcontrol)
);
```

```
endmodule
```

A.2 maindec

```
'timescale 1ns / 1ps
module maindec(
input  [5:0] Inst_part_31_26 ,
output wire jump,branch,alusrc,memwrite,memtoreg,regwrite,regdst ,
output wire [1:0] Alop
);

reg [1:0] Alop_reg;
reg [6:0] Signs;
assign {regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump}=Signs;
assign Alop=Alop_reg;

always @(*) begin
    case (Inst_part_31_26)
        6'b000000 : begin //R-type
            Signs<=7'b1100000;
            Alop_reg<=2'b10;
        end
        6'b100011 : begin //lw
            Signs<=7'b1010010;
            Alop_reg<=2'b00;
        end
        6'b101011 : begin //sw
            Signs<=7'b0x101x0;
            Alop_reg<=2'b00;
        end
        6'b000100 : begin //beq
            Signs<=7'b0x010x0;
            Alop_reg<=2'b01;
        end
        6'b001000 : begin //addi
            Signs<=7'b1010000;
            Alop_reg<=2'b00;
        end
        6'b000010 : begin //jump
            Signs<=7'b0xxx0x1;
            Alop_reg<=2'bxx;
        end
        default : begin
            Signs<=7'b0000000;
            Alop_reg<=2'b00;
        end
    endcase
end
```



```

end

endmodule

```

A.3 aludec

```

`timescale 1ns / 1ps
module aludec(
input wire [5:0] Inst_part_5_0,
input wire [1:0] Alop,
output wire [2:0] Alcontrol
);

assign Alcontrol=(Alop==2'b00)?(3'b010):      //lw or sw
                (Alop==2'b01)?(3'b110):      //beq
                (Alop==2'b10)?(
                (Inst_part_5_0==6'b100000)?(3'b010): //R-type
                (Inst_part_5_0==6'b100010)?(3'b110):
                (Inst_part_5_0==6'b100100)?(3'b000):
                (Inst_part_5_0==6'b100101)?(3'b001):
                (Inst_part_5_0==6'b101010)?(3'b111):(3'bxxx)
                ):(3'bxxx);

endmodule

```

B PC 代码

```

`timescale 1ns / 1ps
module pc(
input clk,
input rst,
input [31:0] next_pc,
output wire [31:0] pc
);

reg [31:0] pc_reg;

always @(posedge clk) begin
    if(rst) begin
        pc_reg<=32'b0;
    end
    else begin
        pc_reg<=next_pc;
    end
end

assign pc=pc_reg;

```

```
endmodule

endmodule

//written in top:
assign next_pc = pc+4'b0100;
pc P(
    .clk(clk),
    .rst(reset),
    .next_pc(next_pc),
    .pc(pc)
);
```