计算机系统结构实验Lab05:
单周期CPU整体实现

常烁晨 521021910369

2023.4.7

摘要

lab05 的实验目标是完成完整的单周期类MIPS处理器，本实验整合了 lab03 和 lab04 的内容，并对相关模块进行修改，此外还新增了多路选择器用于选择 ALU 的运算数，新增了程序计数器用于获取指令，新增了指令存储器，等等。与此同时，lab05 扩充了指令数，在完成基础的 9 条 MIPS 指令后，要求拓展实现 16 条 MIPS 指令。不难看出，本实验是一个较为综合的系统结构实验。

本实验要求实现的 16 条指令分别是：add(i)、sub、and(i)、or(i)、lw、sw、beq、slt、sll、stl、j、jr、jal。

目录

1、实验目的

（1）首先理解单周期类 MIPS 处理器的原理，了解各条指令执行时的数据通路、不同部件之前的数据传递、控制信号的传输与定义等。

（2）本实验建立在前面几个实验的基础上，基于 lab3 和 lab4 ，修改完善相关功能部件，将其整合到本实验所编写的 CPU 之中。

（3）完成 16 条 MIPS 指令，并通过仿真完成验证。

2、源代码实现

2.1 Top

首先要完成 Top 模块的编写。Top模块作为顶层模块，用来完成整个单周期 CPU 的整体功能，调用各个已经完成编写的功能部件单元，如 ALU，Register 等等。为确保正确，实验采用与指导书相同的驼峰/下划线命名规则。

首先完成 Top 模块的输入输出信号线的命名和定义。

```
`timescale 1ns / 1ps
module Top( input clk, input reset );
// pc and inst
    wire [31 : 0] INST;
    wire [31 : 0] PC_IN; wire [31 : 0] PC_OUT;
    wire [31 : 0] PC_TEMP1; wire [31 : 0] PC_TEMP2;
//ctr
    wire REG_DST;  wire REG_WRITE;  wire ALU_SRC; wire MEM_TO_REG;
    wire MEM_READ; wire MEM_WRITE; wire JUMP;
    wire EXT_SIGN;  wire JAL_SIGN; wire SHAMT_SIGN;
    wire [2 : 0] ALU_OP;
//aluctr
    wire [3 : 0] ALU_CTR_OUT;
//alu
    wire [31 : 0] ALU_INPUT_A; wire [31 : 0] ALU_INPUT_B;
    wire [31 : 0] EXT_RES; wire ALU_OUT_ZERO;
    wire [31 : 0] ALU_RES;
```

```
//registers
    wire [4 : 0] READ_REG1; wire [4 : 0] READ_REG2;
    wire [31 : 0] REG_OUT1;  wire [31 : 0] REG_OUT2;
    wire [4 : 0] WRITE_REG; wire [4 : 0] WRITE_REG_TEMP;
    wire [31 : 0] REG_WRITE_DATA; wire [31 : 0] REG_WRITE_DATA_T;
//datamemory
    wire [31 : 0] MEM_READ_DATA;
```

在Top模块中，定义了整个单周期 CPU 各个部件之间的输入与输出信号线，包括数据通路和控制信号通路。在上面的代码中，我分类书写了各个组件所关联的数据信号，方便接下来调用各个编写完成的功能部件，对其连接相应的输入输出信号线。

2.2 instMemory

指令存储器保存了 CPU 接下来要运行的指令。在 CPU 运行的过程中，指令存储器中的指令将逐条加载到 INST 中，经译码后执行相关操作。MIPS中的指令长度均为 32 位。

```
`timescale 1ns / 1ps
module InstMemory(
    input [31 : 0] address,
    output [31 : 0] inst
    );
    reg [31 : 0] instFile [0 : 63];
    assign inst = (address / 4 <= 63 ? instFile[address / 4] : 0);
endmodule
```

指令存储器输入为地址，输出为 32 位二进制数，对应一条 MIPS 指令的编码。注意指令存储器的地址按字节（8bit）编码，因此每次对应的地址都需要除以四，来选择对应的指令编号。这里指令存储器共设置了 64 个单元。

2.3 PC

PC 称为程序计数器，每次在时钟的上升沿或者重启信号时更新 PC 中的内容，将下一条指令的地址保存在 PC 中，从而不断从指令存储器中读入下一条指令。

```
`timescale 1ns / 1ps
module PC(
    input [31 : 0] pcIn,
    input clk,
    input reset,
    output reg [31 : 0] pcOut
    );
    initial pcOut = 0;
    always @ (posedge clk or reset)
    begin
        if (reset)
            pcOut = -4;
        else
            pcOut = pcIn;
    end
endmodule
```

2.4 Reset信号

当遇到 reset 时，PC 和 Registers 文件清零。在模块中新增 reset 信号，需要对 lab4 中的 Registers 作出修改。

```
module Registers(
        ……
    input reset,
        ……
);      ……
```

```
always @ (negedge clk or reset)
        begin if ( reset )
            for (i = 0; i <= 31; i = i + 1)
                regFile [i] = 0;
        else    ……
```

2.5 多路选择器 Mux/MuxReg

由于ALU Registers 等部件在工作过程中往往需要在多条数据通路之中选择一条通路进行输入，因此需要设计多路选择器 Mux 和 MuxReg。

Mux 是 32 位的选择器，用于在两个 32 位二进制数之间选择器中一个向相关部件传输，而 MuxReg 是 5 位的选择器，用于选择两个寄存器编号中的其中一个。

多路选择器的实现很简单，只需要一个三目运算符。

```
`timescale 1ns / 1ps
module Mux(
    input select,
    input [31 : 0] input1,
    input [31 : 0] input2,
    output [31 : 0] out
    );
    assign out = select ? input1 : input2;
endmodule
module MuxReg(
    input select,
    input [4 : 0] input1,
    input [4 : 0] input2,
    output [4 : 0] out
    );
    assign out = select ? input1 : input2;
endmodule
```

2.6 ALU相关部件扩充

由于本实验要求完成 16 条 MIPS 指令的功能实现，因此要对 lab3 中编写的 ALU 相关代码进行补充与扩展。如 ALUOp 的位数需要进行扩展， ALU 支持的运算种类需要扩展.

（1）Ctr — 需要修改 aluop 的编码，将 lab3 中的 aluop 拓展到 3 位。同时新增控制信号 extSign 、jalSign 。下面用表格来展示 lab5 中的真值表对应情况，由于描述真值表的方式与前面 lab3 完全一样，十分简单，故不再赘述。

inst（指令）	opCode	extSign	jalSign	aluop
R-type	6’b000000	0	0	3’b111
lw	6’b100011	1	0	3’b000
sw	6’b101011	1	0	3’b000
beq	6’b000100	1	0	3’b001
addi	6’b001100	1	0	3’b010
andi	6’b001100	0	0	3’b011
ori	6’b001101	0	0	3’b100
jump	6’b000010	0	0	3’b101
jal	6’b000011	0	1	3’b101
	default	0	0	3’b000

（2） ALUCtr — 由于 aluop 的拓展导致 ALUCtr 可以发出更多控制指令，指导 ALU 进行运算。同时在 ALUCtr 中新增了控制信号 shamtSign 、 jrSign (默认都为 0 ）。由于相同的原因，此处使用表格展示真值表。

inst（指令）	{ aluop，funct }	aluctrout
lw/sw/default	9’b000xxxxxx	4’b0010
beq	9’b001xxxxxx	4’b0110
addi	9’b011xxxxxx	4’b0010
andi	9’b011xxxxxx	4’b0000
ori	9’b100xxxxxx	4’b0001
sll	9’b111000000	4’b0011/(shamtSign = 1)

srl	9'b111000010	4’b0100/(shamtSign = 1)
jr	9'b111001000	4’b0101/(jrSign = 1)
add	9’b111100000	4’b0010
sub	9'b111100010	4’b0110
and	9'b111100100	4’b0000
or	9'b111100101	4’b0001
slt	9'b111101010	4’b0111
j/jal	9’b101xxxxxx	4’b0101

（3） ALU — 根据上面扩展后的 aluctrout 信号，ALU 同样支持更多操作，由于 aluctrout 与 lab3 一样是四位二进制编码，因此此处只列举新增的信号。

ALU	aluctrout	function
左移运算（sll）	4’b0011	res = a << b
右移运算（srl）	4’b0100	res = a >> b
直接赋值（jump/jr）	4’b0101	res = a
有符号数比较（slt）	4’b0111	res = sign(a) < sign(b)

2.7 模块实例化

接下来在 Top 文件中实例化各个拓展后的功能部件。

```
Ctr main_controller (
    .opCode(INST[31 : 26]),
    .regDst(REG_DST),
    .aluSrc(ALU_SRC),
    .memToReg(MEM_TO_REG),
    .regWrite(REG_WRITE),
    .memRead(MEM_READ),        .memWrite(MEM_WRITE),
    .branch(BRANCH),
    .extSign(EXT_SIGN),        .jalSign(JAL_SIGN),
    .aluOp(ALU_OP),
    .jump(JUMP)
);
```

```
ALUCtr alu_controller (
    .aluOp(ALU_OP),
    .funct(INST[5 : 0]),
    .aluCtrOut(ALU_CTR_OUT),
    .shamtSign(SHAMT_SIGN),
    .jrSign(JR_SIGN)
);
```

由于 ALU 的输入信号有多种选择，因此需要根据上面新增的控制信号，添加多路选择器，用于选择 ALU 的两路输入情况。首先是 rs 和 左移右移运算之间选择 ALU 的输入信号。接下来是 rt 和 立即数拓展之间选择输入信号。

完成这些输入信号的选择后，最后完成 ALU 的实例化。

```
Mux rs_shamt_selector (
    .select(SHAMT_SIGN),
    .input1({27'b000000000000000000000000000, INST[10 : 6]}),
    .input2(REG_OUT1),
    .out(ALU_INPUT_A)
);
```

```
SignExt signExt (
    .extSign(EXT_SIGN),
    .inst(INST[15 : 0]),
    .data(EXT_RES)
);
Mux rt_ext_selector (
    .select(ALU_SRC),
    .input1(EXT_RES),
    .input2(REG_OUT2),
    .out(ALU_INPUT_B)
);

ALU alu (
    .inputA(ALU_INPUT_A),
    .inputB(ALU_INPUT_B),
    .aluCtrOut(ALU_CTR_OUT),
    .zero(ALU_OUT_ZERO),
    .aluRes(ALU_RES)
);
```

接下来是 Registers 寄存器文件的实例化连接。首先确定 rt 和 rd 的选择，之后再确定是否要跳转到 31 号寄存器（jal指令），最终确定目标寄存器后，完成寄存器文件的实例化。

```
MuxReg rt_rd_selector (
    .select(REG_DST),
    .input1(INST[15 : 11]),
    .input2(INST[20 : 16]),
    .out(WRITE_REG_TEMP)
);
MuxReg rtrd_31_selector (
    .select(JAL_SIGN),
    .input1(5'b11111),
    .input2(WRITE_REG_TEMP),
    .out(WRITE_REG)
);
Registers registers (
    .readReg1(INST[25 : 21]),
    .readReg2(INST[20 : 16]),
    .writeReg(WRITE_REG),
    .writeData(REG_WRITE_DATA),
    .regWrite(REG_WRITE & (~JR_SIGN)),
    .clk(clk),
    .reset(reset),
    .readData1(REG_OUT1),
    .readData2(REG_OUT2)
);
```

下面是 PC 模块、指令存储器以及内存的实例化连接。当发生访存操作时，由前面的数据以及控制通路可见，内存地址由 ALU 完成计算，写数据由读寄存器产生。

```
PC pc_controller (
    .pcIn(PC_IN),
    .clk(clk),
    .reset(reset),
    .pcOut(PC_OUT)
);
InstMemory inst_memory (
    .address(PC_OUT),
    .inst(INST)
);
DataMemory data_memory (
    .clk(clk),
    .address(ALU_RES),
    .writeData(REG_OUT2),
    .memWrite(MEM_WRITE),
    .memRead(MEM_READ),
    .readData(MEM_READ_DATA)
);
Mux mem_alu_selector (
    .select(MEM_TO_REG),
    .input1(MEM_READ_DATA),
    .input2(ALU_RES),
    .out(REG_WRITE_DATA_T)
);
```

不难发现上面的 PC 要想不断更新，还需要 PC 相关的输入输出信号，通过多路选择器，选择顺序执行或者跳转执行来不断更新。下面介绍 PC 更新的过程。

我们所要做的就是通过多路选择器，依次判断是否有 branch 、jump 等指

令，最后得到最终 PC 更新后的值。

```
Mux jal_selector (
    .select(JAL_SIGN),
    .input1(PC_OUT + 4),
    .input2(REG_WRITE_DATA_T),
    .out(REG_WRITE_DATA)
);
Mux branch_selector (
    .select(BRANCH & ALU_OUT_ZERO),
    .input1(PC_OUT + 4 + (EXT_RES << 2)),
    .input2(PC_OUT + 4),
    .out(PC_TEMP1)
);
Mux jump_selector (
    .select(JUMP),
    .input1(((PC_OUT + 4) & 32'hf0000000) + (INST[25 : 0] << 2)),
    .input2(PC_TEMP1),
    .out(PC_TEMP2)
);
Mux jr_selector (
    .select(JR_SIGN),
    .input1(REG_OUT1),
    .input2(PC_TEMP2),
    .out(PC_IN)
);
```

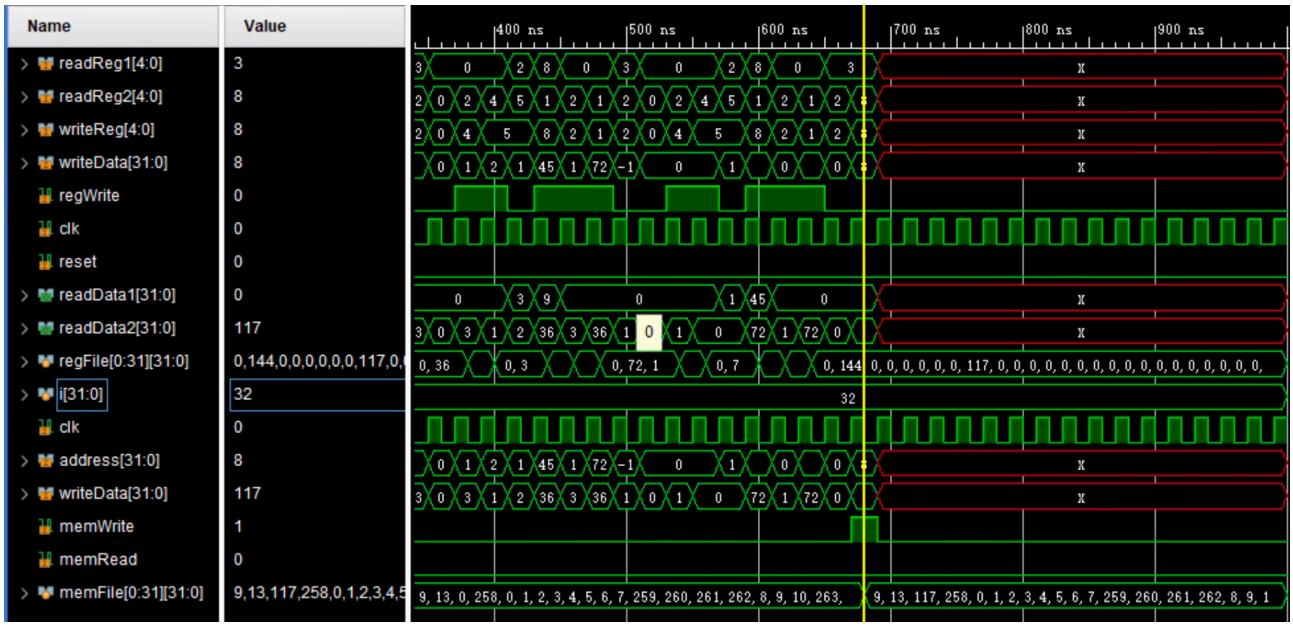
3、仿真实现

本实验中，助教学长提供了一个简单的单周期乘法运算程序，将内存中前两个位置的数相乘，得到的结果保存在内存的第三个位置中，将课程组提供的汇编以及机器代码保存在 lab05 的相关目录下，并在激励文件中将指令存储器和内存的路径添加到初始化模块中进行初始化。

激励文件如下所示。

```
`timescale 1ns / 1ps
module single_cycle_cpu_tb(
    );
    reg clk;
    reg reset;

    Top processor(.clk(clk), .reset(reset));

    initial begin
        $readmemb("D:/archlabs/lab05/inst_data.dat",
processor.inst_memory.instFile);
        $readmemh("D:/archlabs/lab05/data.dat", processor.data_memory.memFile);
        reset = 1;
        clk = 0;
    end

    always #10 clk = ~clk;

    initial begin
        #30 reset = 0;
        #2000;
        $finish;
    end
endmodule
```

课程组提供的汇编代码、机器代码、指令和内存的初始化文件略，下面展示运行仿真的结果示意图，可以看到程序正确运行，在 680ns 时计算完成，此时内存第三个单元保存了前两个单元中数据的乘积 9 * 13 = 117 。



4、致谢

感谢刘老师以及三位实验助教学长学姐的悉心指导和帮助。

本实验是第一个较为综合的计算机系统结构实验，借助本实验，我们完成了一个单周期类 MIPS 指令的CPU，实现了其支持的 16 条指令，完成这样的实验是比较困难的，感谢助教学长学姐提供的仿真检测代码，这大大减轻了我们的工作量，使我们能够把精力用在最关键的逻辑设计和代码编写上。

感谢上海交通大学的计算机系统结构实验室，为我们准备了相关的实验环境以及 FPGA 开发板，学校和学院提供的优秀设备让我们受益匪浅。