实验2：单周期 MIPS 处理器

实验时间：4.19 - 5.19

实验人员：杨乙 21307130076

指导老师：王雪平

实验目的

- 设计多周期 MIPS 处理器，包括
  - 完成单周期 MIPS 处理器的设计；
  - 在 Vivado 软件上进行仿真；
  - 编写 MIPS 代码验证单周期 MIPS 处理器；

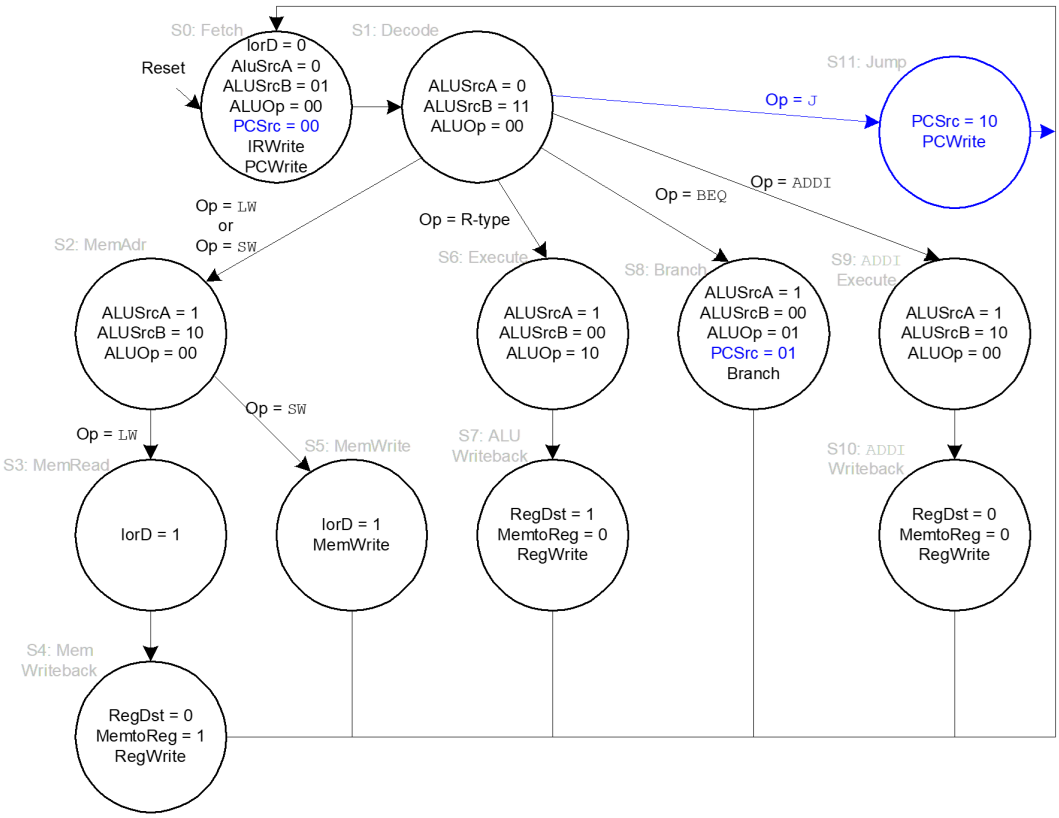
实验任务

1.设计多周期MIPS处理器

- 由于多周期处理器的基本框架、包含基本指令的 FSM 已在参考资料上给出，同时基本指令的实现原理和单周期处理器有着一定的重合。因此在指令方面本报告将重点分析 `andi`、`ori`、`bne` 指令的扩展
- maindec 模块的基本代码在参考资料上已经给出，本报告不再对其进行分析
- 数据路径的实现虽然与单周期处理器存在较大不同，但是根据教材上给出的多周期处理器电路图较易实现，故本报告不包含数据路径的实现及其代码，而是给出各个模块的 RTL 原理图，用以验证代码的正确性
- 其他模块的实现与单周期 MIPS 处理器大同小异，本报告不再对其进行分析

FSM

参考资料给出的包含基本指令的 FSM 如图：



分析 FSM 中各个状态对应的操作对于后续确定控制信号的值很有必要。FSM 中各个状态对应编号、在程序中的命名、对应的操作如下表：

状态编号	状态名	操作
S0	FETCH	取指令和更新 PC
S1	DECODE	译码
S2	MEMADR	lw / sw 指令求存储器地址
S3	MEMRD	lw 指令读存储器
S4	MEMWB	lw 指令写寄存器文件
S5	MEMWR	sw 指令写存储器
S6	RTYPEEX	R 型指令运算
S7	RTYPEWB	R 型指令写寄存器文件
S8	BEQEX	beq 指令运算
S9	ADDIEX	addi 指令运算
S10	ADDIWB	I 型指令写寄存器
S11	JEX	J 型指令运算
S12	*BNEEX*	*bne 指令运算*
S13	*ORIEX*	*ori 指令运算*

状态编号	状态名	操作
S14	*ANDIEX*	*andi 指令运算*

寄存器使能信号

多周期处理器的指令执行需要多步，因此需要插入非体系结构寄存器来保存每一步的结果。部分非体系结构寄存器的值仅在特定的状态下更新，因此需要对应的寄存器使能信号。各寄存器使能信号及作用、信号为 1 时对应的状态如下表：

寄存器使能信号	作用	信号为 1 对应状态
IRWrite	更新指令寄存器使能	S0
MemWrite	内存写使能	S5
PCWrite	更新 PC 寄存器使能	S0、S11
Branch	beq 指令信号	S8
RegWrite	寄存器写使能	S4、S7、S10

（根据之前所述的各个状态对应的操作，可以很容易地得到各寄存器使能信号为 1 时对应的状态，在此不作分析）

复用器选择信号

在指令执行的多个步骤中，各接口需要根据状态的不同传入不同的值。因此需要用选择信号控制复用器以实现对不同值的选择。各复用器选择信号、输入、信号为 1 时对应状态、输出如下表：

复用器选择信号	为 1 输入	为 0 输入	信号为 1 对应状态	输出
MemtoReg	Data	ALUOut	S4	WD3（写入寄存器的值）
RegDst	指令 rd 字段	指令 rt 字段	S7	A3（写入目的寄存器编号）
IorD	PC	ALUOut	S3、S5	Adr（读写存储器地址）
ALUSrcA	寄存器 A 值	PC	S2、S6、S8、S9、*S12*、*S13*、*S14*	SrcA（ALU 输入 A）
ImmExt	*0 扩展立即数*	*符号扩展立即数*	*S13*、*S14*	*imm（扩展后的立即数）*

（有关以上各复用器选择信号的内容在参考资料中已有详细说明，在此不作分析）

ALUSrcB（输出 SrcB）：

信号值	输入	对应状态
00	寄存器 B 值	S6、S8
01	4	S0
10	*imm（扩展后的立即数）*	S2、S9、*S13*、*S14*
11	*imm << 2*	S1

- R 型指令和分支指令都需要对 A、B 两个寄存器值进行运算（根据单周期处理器设计实验，分支指令借助 ALU 进行减法运算），因此状态为 S6、S8 需要传入寄存器 B 值
- 状态 S0 需要投机使用 ALU 更新 PC，因此传入 4
- 状态 S2、S9、*S13*、*S14* 需要对立即数进行运算，传入立即数
- 对于 beq 指令，首先需要将 `PC + 4` 与 `imm << 2` 相加来计算目的地址。我们注意到 S1 状态未使用 ALU。为了节省硬件开销，我们在 S1 状态时将二者相加计算目的地址，并存储在 ALUOut 中。若指令不是 beq，计算后的地址后续不会使用，不影响处理器的正常工作。因此状态 S1 传入 `imm << 2`

PCSrc（输出 PCnext）：

信号值	输入	对应状态
00	ALUResult	S0
01	ALUOut	S8、*S12*
10	PCJump	S11

- 所有指令都在状态 S0 通过 ALU 更新 PC，ALUResult 的值就是 `PC + 4`。因此状态 S0 传入 ALUResult
- 如上所述，分支指令在 S1 状态通过 ALU 计算目的地址并存储在 ALUOut 中，并且在 S8、*S12* 两个状态下通过 ALU 进行减法运算。在这两个状态下，有用的信号是 Zero 而非 ALUResult，同时我们需要的目的地址已经在上一状态计算完成并存储在 ALUOut 中。因此我们可以在这时将目的地址传给 PC。因此状态 S8、*S12* 传入 ALUOut
- 跳转指令在译码后即可将 addr 左移二位和 PC 高四位拼位，得到跳转地址 PC。因此 S11 状态传入 PCJump

ALU 运算功能

ALUControl 值、ALU 运算功能与指令的对应关系如下表：

ALUControl	运算功能	对应指令
000	Result = X + Y	**add** lw sw addi
001	Result = X - Y	**sub** beq *bne*
010	Result = X & Y	**and** *andi*
011	Result = X | Y	**or** *ori*
100	Result = (X < Y) ? 1: 0	**slt**

存储器合并

因为多周期处理器的指令执行分为多个周期，且读写数据、读写指令不会发生在同一周期中。因此可以将指令存储器和数据存储器合并成一个支持读写操作的大容量存储器（这与实际系统中的处理方法相同）。合并后的 mem 模块如下：

```
`timescale 1ns / 1ps

module mem(
    input  logic        clk, we,
    input  logic [31:0] a, wd,
    output logic [31:0] rd
    );

    logic [31:0] RAM[63:0];

    initial $readmemh("memfile.dat",RAM);

    assign rd = RAM[a[31:2]];
    always_ff @(posedge clk)
        if(we) RAM[a[31:2]] <= wd;

endmodule
```

指令扩展 - `andi`、`ori` 指令

二者对立即数的扩展方式是 0 扩展，而现有的立即数扩展指令都是符号扩展。因此需要引入控制信号 ImmExt 来选择扩展方式。同时引入二路选择器 extmux，以 signimm（符号扩展立即数）作为 0 输入，以 zeroimm（0 扩展立即数）作为 1 输入，以 ImmExt 作为选择信号，将输出信号记为 imm。同时需要在状态 S1（译码）和状态 S10（I 型指令写寄存器）之间引入两个新的状态 S13、S14，以完成对这两条指令的运算

datapath 模块修改如下：

```
    //......
    signext     se(instr[15:0], signimm);
    zeronext    ze(instr[15:0], zeroimm);                          //进行0扩展
    mux2 #(32)  extmux(signimm, zeroimm, immext, imm);             //二路选择器extmux
    mux4 #(32)  m4(b, four, imm, (imm << 2), alusrcb, srcb);       //m4的输入信号中signimm改为imm
```

加入 zeronext 模块：

```
module zeronext(
    input  logic [15:0] a,
    output logic [31:0] y );

    assign y = {{16'b0}, a};    //进行0扩展
endmodule
```

对于这两条指令，ALU 执行的运算分别与 `and`、`or` 指令相同。因此需要将 ALUOp 的值扩展为 3 位。将二者的 ALUControl 值设置如下（在 aludec 模块中添加如下代码）：

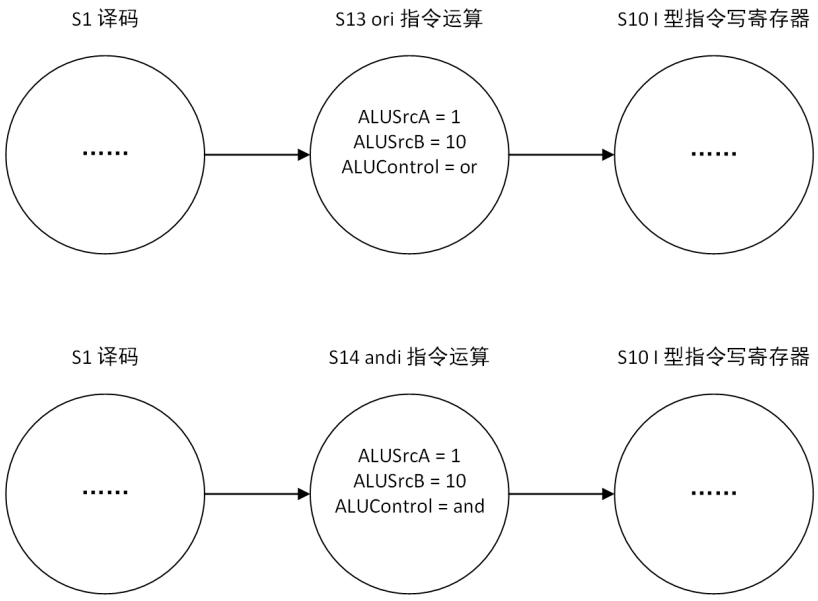
```
case (aluop)
    //......
    3'b011: alucontrol <= 3'b011;  //对ori指令，ALU执行or运算
    3'b100: alucontrol <= 3'b010;  //对andi指令，ALU执行and运算
```

引入复用器选择信号 ImmExt，在 maindec 模块中添加如下代码：

```
        //......
        localparam ORIEX   = 4'b1101; //state 13
        localparam ANDIEX  = 4'b1110; //state 14

        //......

        //opcode
        localparam ANDI    = 6'b001100;
        localparam ORI     = 6'b001101;

        //next state logic
        always_comb
            case(state)
                //......
                DECODE:  case(op)
                    //......
                    ORI:     nextstate = ORIEX;
                    ANDI:    nextstate = ANDIEX;
                endcase
```

```
                //......
                ANDIEX:   nextstate = ADDIWB;
                ORIEX:    nextstate = ADDIWB;
                //......
            endcase

        assign { pcwrite, memwrite, irwrite, regwrite,
                 alusrca, branch, iord, memtoreg, regdst,
                 alusrcb, pcsrc, aluop, branchbne, immext } = controls;
        //对于控制信号branchbne的具体分析将在后续展开

        always_comb
            case(state)
                //......
                ANDIEX:    controls = 18'b0000_10000_1000_10001;
                ORIEX:     controls = 18'b0000_10000_1000_01101;
                //......
            endcase
```

FSM 中添加如下内容：



指令扩展 - `bne` 指令

`bne` 指令和 `beq` 指令只有转移条件相反这一处不同，因此不必设置新的 ALUControl 值，而是引入控制信号 BranchBne 来指示 `bne` 指令。和 `beq` 指令一样，对于 `bne` 指令，ALU 执行的运算与 `sub` 指令相同，因此将 `bne` 的 ALUOp 设置为与 `beq` 相同，都为 subu。因此除了 BranchBne 控制信号与 `beq` 不同以外，其余控制信号取值都与 `beq` 相同。且需要在状态 S1（译码）和状态 S0（取指令和更新 PC）之间引入新的状态 S12，以完成 bne 指令的运算

在 maindec 模块中添加如下代码：

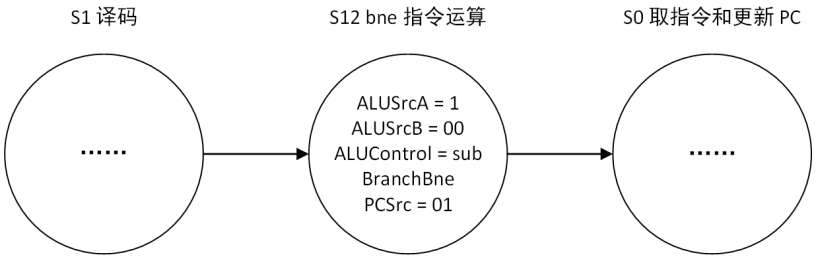
```
        //......
```

```
        localparam BNEEX    = 4'b1100; //state 12

        //opcode
        //......
        localparam BNE      = 6'b000101;
        //......

        always_comb
            case(state)
                //......
                DECODE:  case(op)
                    //......
                    BNE:     nextstate = BNEEX;
                    //......
                endcase

                //......
                BNEEX:   nextstate = FETCH;
            endcase

        assign { pcwrite, memwrite, irwrite, regwrite,
                 alusrca, branch, iord, memtoreg, regdst,
                 alusrcb, pcsrc, aluop, branchbne, immext } = controls;

        always_comb
            case(state)
                //......
                BNEEX:    controls = 18'b0000_10000_0001_00110;
                //......
            endcase
```

`beq` 指令是通过组合逻辑来控制 PCEn，当 Zero 和 Branch 值都为真时 PCEn 为真。扩展 `bne` 指令后，Zero 为 0、BranchBne 为真时PCEn 也为真。因此对 controller 模块进行如下修改：

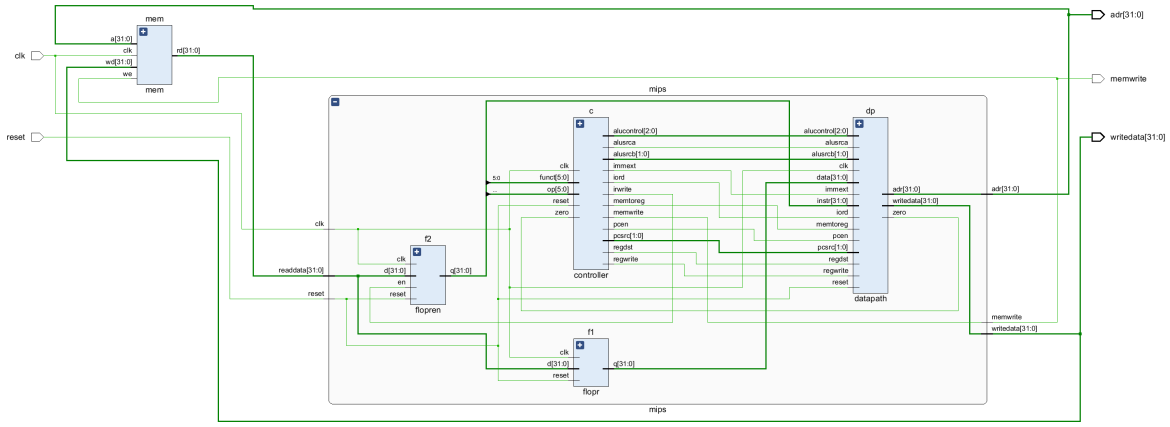
```
    // pcen
    assign pcen = (branch & zero) | ((~zero) & branchbne) | pcwrite;
```

FSM 中添加如下内容：

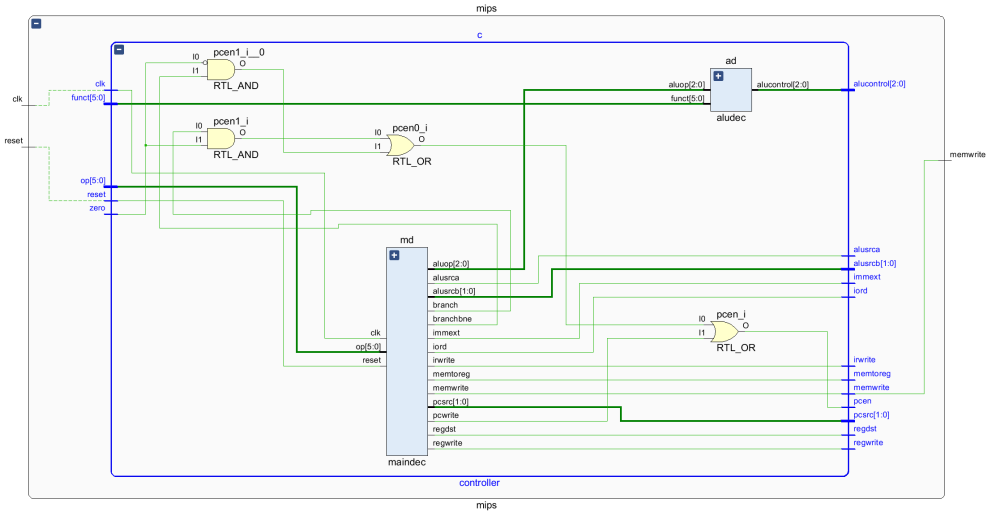


至此已经完成了所有指令的拓展。各个模块的 RTL原理图如下：

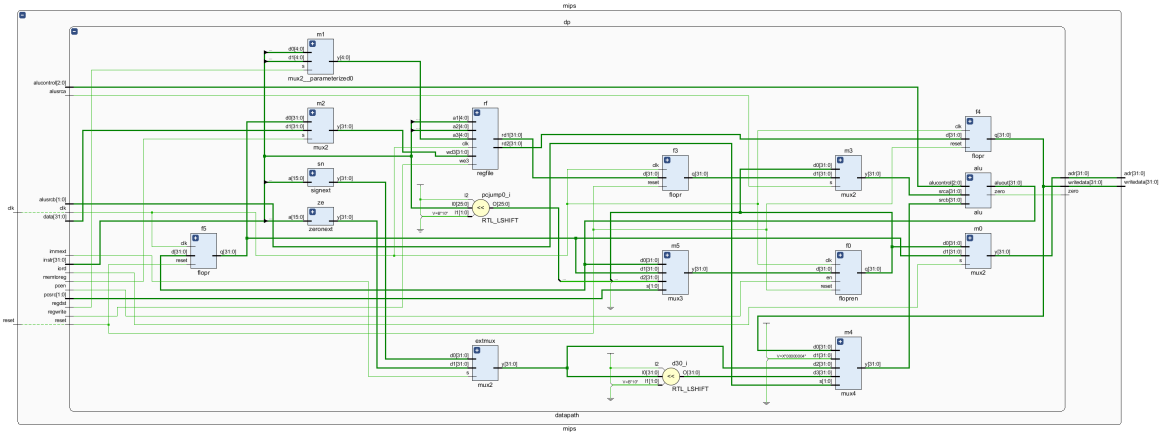
Top:



controller:



datapath:



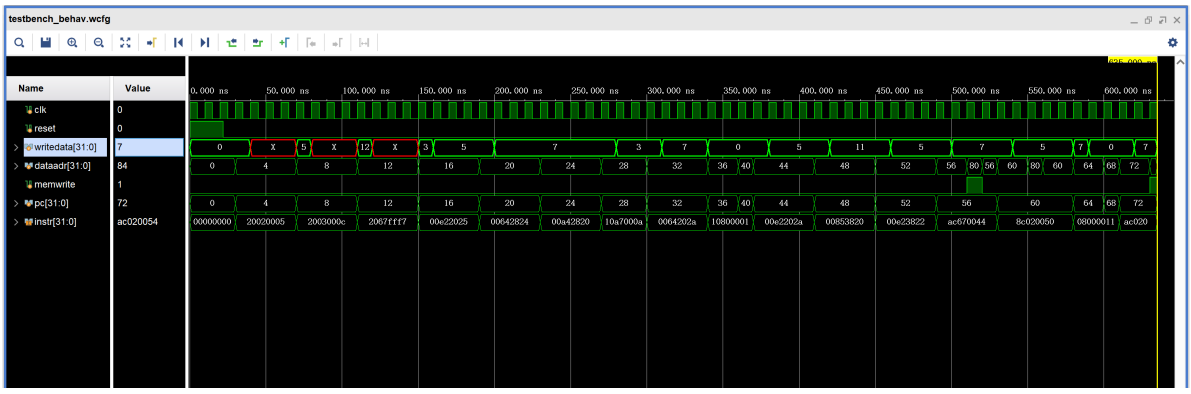
2.仿真

根据 FSM，各指令执行周期如下：

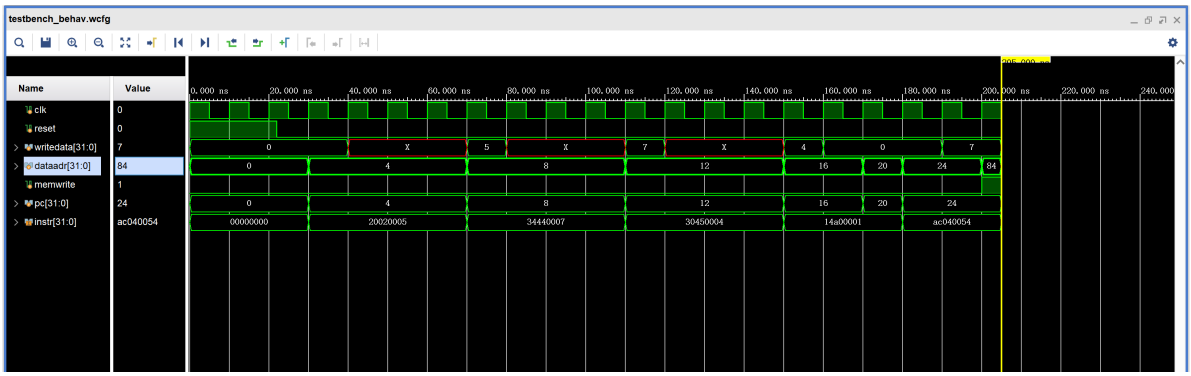
指令	执行周期
lw	5
sw	4
R 型	4
I 型	4
分支	3
跳转	3

基本指令仿真测试

用给出的仿真代码和 memfile.dat 文件进行测试。最后 writedata 值为 7，dataadr 值为 84，仿真波形图与参考资料相同。说明 `lw`、`sw`、`add`、`sub`、`and`、`or`、`slt`、`beq`、`addi`、`j` 指令正常执行



用 2.3 IO 接口设计给出的测试代码和 memfileExt.dat 文件进行测试。最后 writedata 值为 7，dataadr 值为 84，仿真波形图与参考资料相同。说明 `andi`、`ori`、`bne` 指令正常执行



3.验证

I/O 接口的实现方法与单周期处理器基本相同。不同点在于多周期处理器将指令存储器和数据存储器合并成一个存储器，因此各模块要进行相应的改动。I/O 接口的实现原理已经在单周期处理器报告中给出，在此仅作简略叙述。为了模拟存储器映像的 I/O 接口结构，我们扩充了存储器空间，将 `0x80 - 0xFF` 作为 I/O 接口空间。根据这种方案，我们可以：

1. 通过 `addr[7]` 的值即可区分存储器空间和 I/O 接口空间
2. 通过 `addr[3:2]` 的值来编码各 I/O 端口地址

存储译码器模块

将存储器、I/O接口、七段数码管三个模块封装为存储译码器模块（MemoryDecoder），对 mem 模块的空间作如下调整：

```
        logic [31:0] RAM[255:0];
```

MemoryDecoder 模块核心代码如下：

```
    //addr[7] == 1 说明是io接口空间

    //是否从 IO 读
    assign pRead    = (addr[7] == 1'b1) ? 1 : 0;

    //是否向 IO 写
    assign pWrite   = (addr[7] == 1'b1) ? writeEn : 0;

    //写入数据存储器的开关
    assign we       = (addr[7] == 1'b0) ? writeEn : 0;

    assign readData = (addr[7] == 1'b0) ? ReadData1 : ReadData2;
```

以上代码通过 addr[7] 的值实现了对存储器空间和 I/O 接口空间读写操作的选择

I/O 接口模块

I/O 接口模块的代码已经在单周期处理器的参考资料上给出，在此仅作简要分析。为了实现 CPU 查询方式 I/O 输入输出，引入状态端口 status[1:0]，将 status[0] 作为 LED 状态位，将 status[1] 作为 switch 状态 位，使得可以通过开关 BTNR 和 BTNL 控制开关数据输入和 LED 数据输出。同时，我们通过 addr[3:2] 来编码 switch 高 8 位端口、switch 低 8 位端口、LED端口、状态端口的地址，用以向不同 的 I/O 端口读写数据

7 段数码管模块

7 段数码管模块可设计为 mux7seg 和 Hex7Seg 模块。前者控制各组数码管的分时复用（an值）并传入数据，后者根据传入的数据控制数码管的显示（a2g）值。为了得到正确的仿真结果，在这一步仅需要传入最右侧数码管组的数据，不需要进行分时复用。mux7seg 模块核心代码如下：

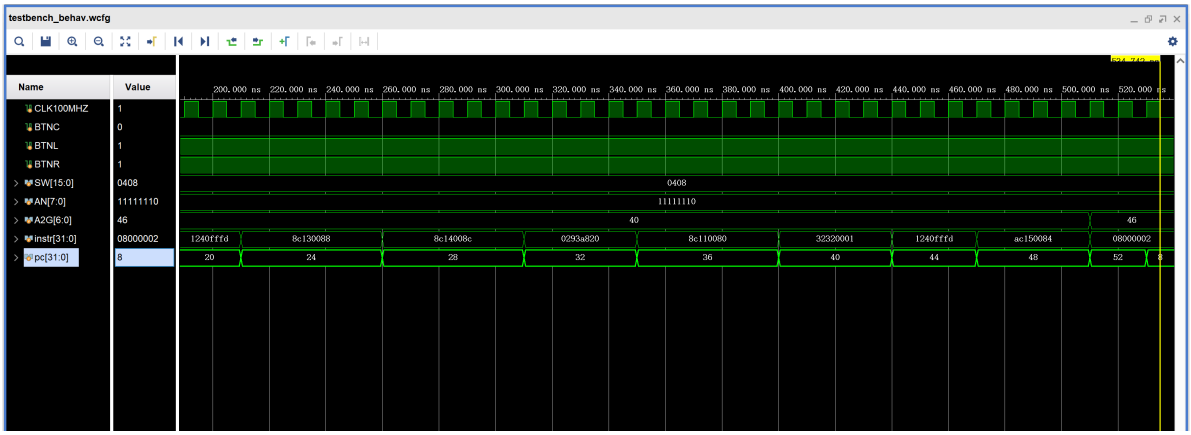
```
    //仿真用
    assign an = 8'b1111_1110;
    assign digit1 = {1'b0, digit[3:0]};
```

Hex7Seg 模块核心代码如下：

```
    //......
    always_comb
        begin
            case(data)
                'h0: a2g = 7'b1000000;
                //......
                'hF: a2g = 7'b0001110;
                'h10: a2g = 7'b1110110; //等号
                default: a2g = 7'b0000000;
            endcase
        end
```

仿真验证

用 2.3 IO 接口设计给出的测试代码和 TestIO.dat 文件进行测试。A2G 值出现了 40（显示 0）到 46（显示 C）的变化， 仿真波形图与参考资料相同，可以验证多周期 MIPS 处理器的正确性



实验感想

1. 在多周期处理器的设计中，ALU 的复用是一个贯穿始终的问题。在不同状态下复用 ALU 可以节省硬件开销
2. 通过仿真可以很方便地对实现效果进行验证，减小成本，节省时间