实验 3：多周期 MIPS 处理器

实验时间 2023.5.6

1 实验目的

设计多周期 MIPS 处理器，包括

完成多周期 MIPS 处理器的设计

在 Vivado 软件上进行仿真

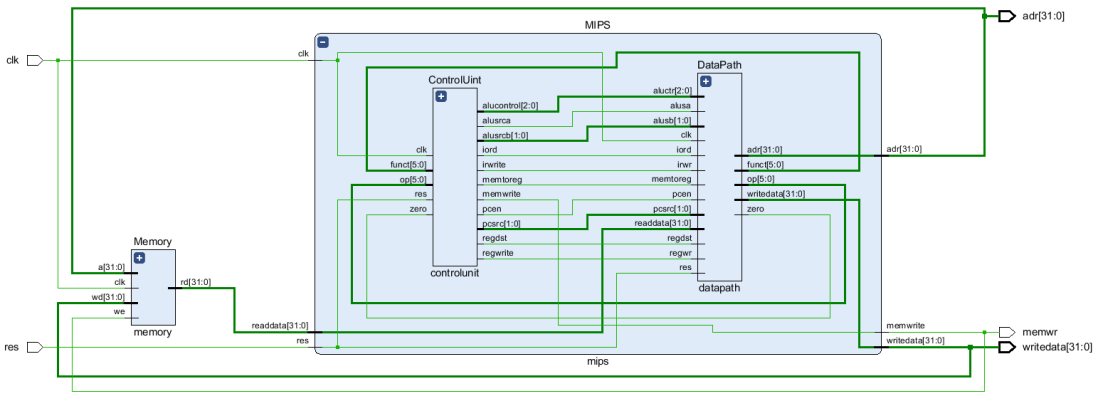
编写 MIPS 代码验证多周期 MIPS 处理器

2 实验过程

2.1 设计多周期 MIPS 处理器

实验结果

最后设计完成后的 RTL 图如下：



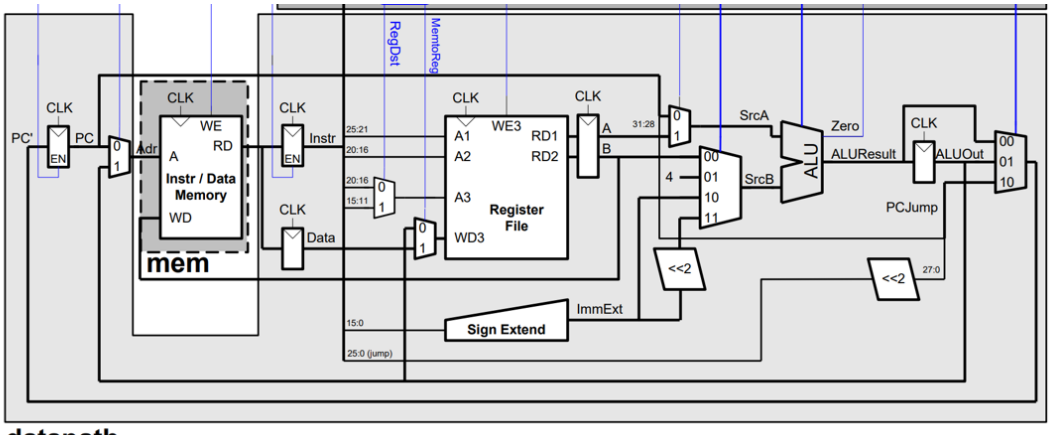
采用自顶向下的设计方法，对各个模块经行封装。

最后的整体文件架构如下：

- Design Sources (1)
  - **mips_top** (mips_top.sv) (2)
    - Memory : memory (memory.sv)
    - MIPS : mips (mips.sv) (2)
      - ControlUint : controlunit (controlunit.sv) (2)
        - MainDec : maindec (maindec.sv)
        - ALUDec : aludec (aludec.sv)
      - DataPath : datapath (datapath.sv) (15)
        - PcReg : flopenr (flopenr.sv)
        - AdrMux : mux2 (mux.sv)
        - IR : flopenr (flopenr.sv)
        - DR : flopenr (flopenr.sv)
        - A3Mux : mux2 (mux.sv)
        - Wd3Mux : mux2 (mux.sv)
        - RegFile : regfile (regfile.sv)
        - Rd1Reg : flopenr (flopenr.sv)
        - Rd2Reg : flopenr (flopenr.sv)
        - SignExtend : signextend (signextend.sv)
        - AMux : mux2 (mux.sv)
        - BMux : mux4 (mux.sv)
        - ALU : alu (alu.sv)
        - ALUoutR : flopenr (flopenr.sv)
        - PC_nextMux : mux4 (mux.sv)

具体过程

首先完成数据通路的设计。数据通路按照所给 ppt 的资料，结合《数字设计》书上的相应章节，按照下图进行数据通路的设计：



datapath

相比单周期 MIPS 处理器，多周期将 imem 和 dmem 整合在一起，需要有写入口和读出口，由写使能控制。除此之外，对 ALU 的利用率也提高了，

但随之而来的就是 ALU 两个来源的复杂化，其中 srcB 有 4 个来源。最明显的就是关于触发器的变化，需要带写使能、复位信号，并且数量多了很多。

首先实现带写使能的触发器，需要一个位宽参数 x：

```
module flopenr#(parameter x=8)
```

在时钟上升沿，若写使能有效，就写入。在复位信号上升沿就将输出置 0：

```
        if(res)q<=0;
        else if(en)q<=d;
```

接下来实现二选一、四选一的模块，同样需要位宽参数 W：

```
module mux4 #(parameter W=8)
module mux2 #(parameter W=8)
```

接下来实现 mem 部分。发现 mem 部分和单周期 MIPS 处理器的 dmem 部分相似，首先将容量改为 64*64。但是还需要读入指令，就在 dmem 基础上增加激励信号，读出指令即可。本次实验有两个指令文件 memfile.dat 和 memfile_moreinstra.dat，其中后者扩展了 andi、ori、beq 指令。

```
logic [63:0] RAM[63:0];
    assign rd=RAM[a[31:2]];

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

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

接下来是寄存器组、符号扩展、ALU 模块。这一部分和单周期 m 的寄存器组、符号扩展、ALU 模块实现的功能完全一样，对于控制信号的要求也一样，就直接使用单周期 m 的寄存器组即可，就不再展示。

其中，符号扩展模块在控制信号 zeroext 的影响下，为 0 进行符号扩展，为 1 进行零扩展，以支持 andi 与 ori 指令。

将 mem 独立出，其它的模块按照要求相互连接。

在数据通路中有 6 个寄存器，它们是：

pc 寄存器，由 PCen 控制，存放下一条 pc；

指令寄存器，由 IRWrite 控制，存放读出的指令；

数据寄存器，写使始终为 1，存放从 mem 里取出的数据；

rd1 与 rd2，写使能始终为 1，存放从寄存器组中读出的值；

ALU 寄存器，存放 ALU 运算的结果，写使能始终为 1。

在数据通路中有 4 个二选一复用器：

地址复用器，由 IorD 选择地址来自 PC 还是运算的结果；

寄存器地址复用器，由 regDst 选择写入的寄存器是 instruction 的哪一部分；

寄存器写入内容复用器，由 MemtoReg 选择写入的数据来自内存读出的数据还是 ALU 运算的结果；

ALU 来源 A 复用器，由 ALUSrcA 选择 ALU 来源 A 是来自 PC 还是寄存器读出的信息。

在数据通路中还有 2 个四选一复用器：

ALU 来源 B 复用器，由 ALUSrcB 选择来源 B 是来自寄存器，常数 4，符号扩展的立即数还是符号扩展并左移 2 位的立即数。

PC 更新复用器，由 PCSrc 选择下一条 PC 是 ALU 的结果还是被存在 ALU 结果寄存器里的值，还是由 instr 部分和 pc 一起拼出的值（用于 jump）。

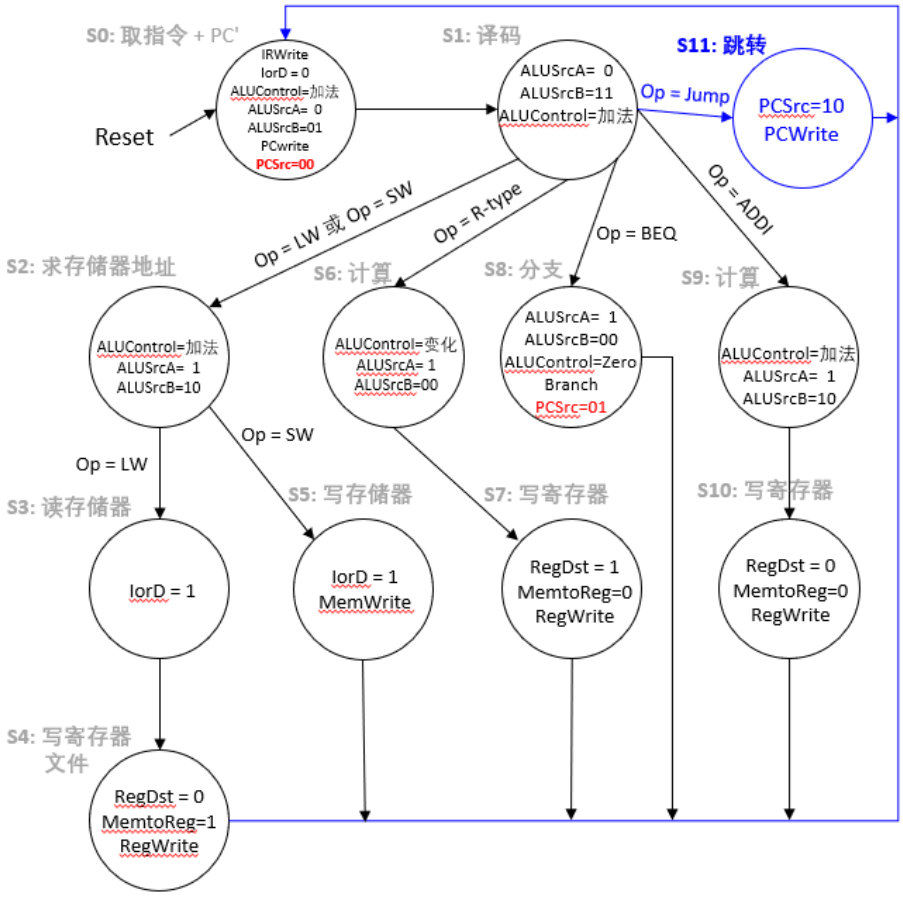
下面是数据通路的主要连接方式：

```
    flopenr#(32) PcReg(clk,res,pcen,pc_next,pc);
    mux2#(32) AdrMux(iord,pc,aluout,adr);
    flopenr#(32) IR(clk,res,irwr,readdata,instr);
    flopenr#(32) DR(clk,res,1'b1,readdata,data);

    assign op=instr[31:26];
    assign funct=instr[5:0];
    mux2#(5) A3Mux(regdst,instr[20:16],instr[15:11],wa3);
    mux2#(32) Wd3Mux(memtoreg,aluout,data,wd3);

    regfile
RegFile(clk,regwr,instr[25:21],instr[20:16],wa3,wd3,rd1,rd2);
    flopenr#(32) Rd1Reg(clk,res,1'b1,rd1,A);
    flopenr#(32) Rd2Reg(clk,res,1'b1,rd2,writedata);
//    assign writedata=B;

    signextend SignExtend(instr[15:0],zeroext,signimm);
    mux2#(32) AMux(alusa,pc,A,srcA);
    mux4#(32) BMux(alusb,writedata,32'h4,signimm,signimm<<2,srcB);
    alu ALU(aluctr,srcA,srcB,aluresult,zero);

    flopenr#(32) ALUoutR(clk,res,1'b1,aluresult,aluout);
    mux4#(32)
PC_nextMux(pcsrc,aluresult,aluout,{pc[31:28],instr[25:0],2'b00},32'b0,p
c_next);
```

至此，数据通路已被设计完成。接下来设计控制器。

控制器的 ALUDecoder 与单周期 MIPS 处理器完全一致，就不再展示，主要展示 MainDec 的部分。

主译码器采用了有限状态机的设计方法，根据不同指令的不同阶段，设定为不同的步骤，每条指令都从 ifetch 阶段、decode 阶段开始，在 decode 阶段后根据译码的结果转到各自的独特状态进一步执行，将每一条指令分成 3-5 步执行。其状态图如下：



首先将各个状态编码：

```
localparam fetch=4'b0000,decode=4'b0001,memadr=4'b0010,memrd=4'b0011,
           memwb=4'b0100,memwr=4'b0101,rtypeex=4'b0110,rtypewb=4'b01
11,
           bex=4'b1000,addiex=4'b1001,iwb=4'b1010,jex=4'b1011,
           subiex=4'b1100,andiex=4'b1101,oriex=4'b1110;
```

接着将不同的指令编码：

```
localparam lw=6'b100011,sw=6'b101011,rtype=6'b000000,beq=6'b000100,
           addi=6'b001000,j=6'b000010,andi=6'b001100,ori=6'b001101,
           slti=6'b001010,bne=6'b000101,subi=6'b001001;
```

然后根据上面的状态图，由当前状态和指令的 op 码，跳转到相应的下一状态。这一部分代码比较长，而且比较容易理解，就不展示了，可以在实验结论中的链接里下载打开来查看。

最后是根据当前的状态，给相应的控制信号赋值并输出，用一个 case-endcase 代码块即可解决。注意这是实现了扩展的控制信号，所以 ALUOp 是 3 位，不是 ppt 上的 2 位，一共有 16 位控制信号。

```
always_comb
       case(state)
           fetch:  controls=16'b1010000000100000;
           decode: controls=16'b0000000001100000;
           memadr: controls=16'b0000100001000000;
           memrd:  controls=16'b0000001000000000;
```

```
            memwb:  controls=16'b0001000100000000;
            memwr:  controls=16'b0100001000000000;
            rtypeex:controls=16'b0000100000000010;
            rtypewb:controls=16'b0001000010000000;
            bex:    controls=16'b0000110000001001;
            addiex: controls=16'b0000100001000000;
            subiex: controls=16'b0000100001000001;
            andiex: controls=16'b0000100001000100;
            oriex:  controls=16'b0000100001000101;
            iwb:    controls=16'b0001000000000000;
            jex:    controls=16'b1000000000010000;
            default:controls=16'bxxxx;
        endcase
```

接下来是对于 subi、andi、ori、bne 进行扩展。这些 I 型指令是除了在 exe 阶段的 ALUOp 与 addi 不一样，最后写回的部分控制信号与 addi 完全一样。就根据相应的要求添加状态即可。但是为了保证 andi 与 ori 是进行零扩展，就需要增加控制信号 zeroext，检测到 andi 与 ori 时就为 1。

对于 beq 指令，本实验的解决办法和 ppt 上的解决办法不一样。发现它们操作码除了最低位不一样，其它都一样。所以就将原来的 beq 指令的 ex 阶段改为 bex 阶段，beq 和 bne 共享这一阶段，在控制器输出 PCEn 控制信号时需要将 zero 标志与 op 码的最低位经行比较，两者不等就说明需要跳转，当 branch 为真时，就可以将 PCEn 置为真。同时当 PCWrite 为真时，PCEn 也为真。于是 pcen 可以这样写：

```
assign pcen=branch&(zero!=op[0])|pcwrite;
```

至此，已完成控制器的设计。

接下来封装各个部分。首先将控制器和数据路径封装在 MIPS 模块里：

```
    controlunit
ControlUint(clk,res,op,funct,zero,memwrite,irwrite,regwrite,alusrca,
            iord,memtoreg,regdst,pcen, zeroext,
            alusrcb,pcsrc,alucontrol);
    datapath DataPath(clk,res,iord,irwrite, zeroext,
             pcen,alusrca,regwrite,regdst,memtoreg,
            pcsrc,alusrcb, ,readdata,adr, writedata,funct,op,zero);
```

接下来将 mips 模块和 memory 模块封装在一起，组成最终的顶层文件

```
    logic [31:0] readdata;
    memory Memory(clk,memwr,adr,writedata,readdata);
    mips MIPS(clk,res,readdata,memwr,adr,writedata);
```

2.2 仿真

实验结果

仿真后运行实例代码的结果如图，控制台的消息显示成功：

datapath.sv | memory.sv | Untitled 1* | testbench.sv

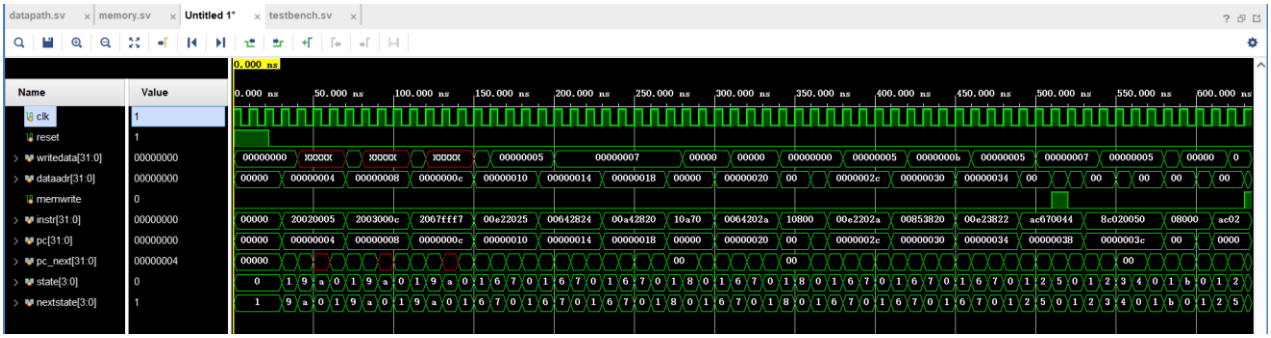
D:/my_project/COS/2/MIPS/MIPS.srcs/sim_1/new/testbench.sv

```
13          begin
14              reset<=1;#22;reset<=0;
15          end
16
17      always
18      begin
19          clk<=1;#5;clk<=0;#5;
20      end
21
22      always@(negedge clk)
23      begin
24          if(memwrite)
25          begin
26              if(dataadr===84&writedata===7)
27              begin
28                  $display("success");
29                  $stop;
30              end
```

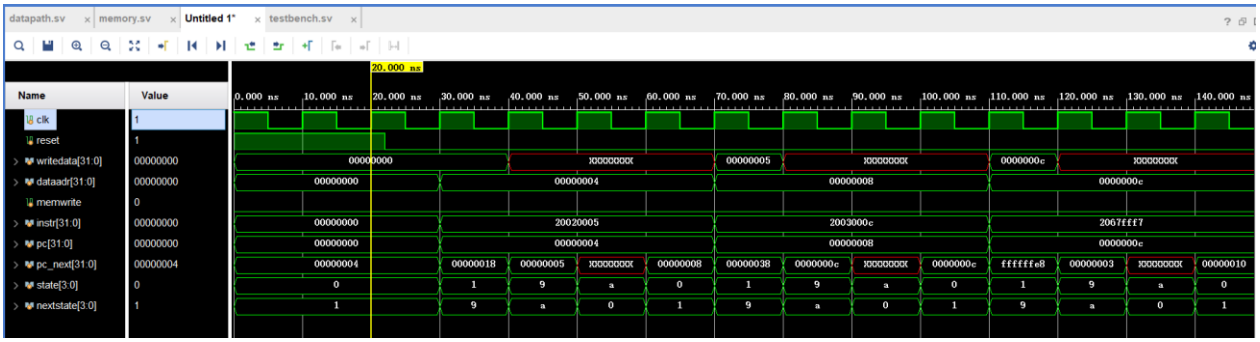
Tcl Console | Messages | Log

```
success
$stop called at time : 635 ns : File "D:/my_project/COS/2/MIPS/MIPS.srcs/sim_1/new/testbench.sv" Line 29
relaunch_sim: Time (s): cpu = 00:00:01 ; elapsed = 00:00:07 . Memory (MB): peak = 783.648 ; gain = 0.000
```

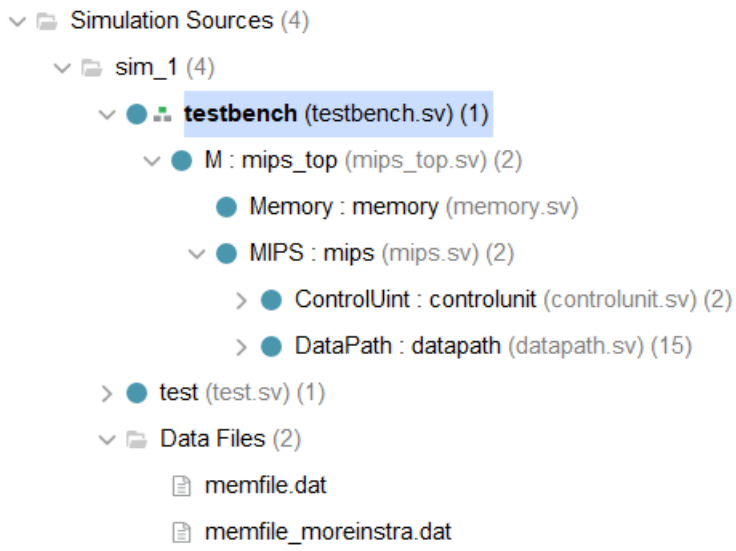
仿真的波形图如下：



放大后可以发现，对应 addi 的部分是 4 周期（黄色光标及其之后的 4 个周期）：



仿真文件的结构如下：



具体过程

首先使用 ppt 上的示例代码。如果正确执行，它就会在地址为 84 的位置写入数据 7。具体内容如下：

```
# Test the MIPS processor.
# add, sub, and, or, slt, addi, lw, sw, beq, j
# If successful, it should write the value 7 to address 84

#          Assembly              Description            Address    Machine
main:      addi $2, $0, 5        # initialize $2 = 5      0          20020005
           addi $3, $0, 12       # initialize $3 = 12     4          2003000c
           addi $7, $3, -9       # initialize $7 = 3      8          2067fff7
           or   $4, $7, $2       # $4 = (3 OR 5) = 7      c          00e22025
           and  $5, $3, $4       # $5 = (12 AND 7) = 4    10         00642824
           add  $5, $5, $4       # $5 = 4 + 7 = 11        14         00a42820
           beq  $5, $7, end      # shouldn't be taken     18         10a7000a
           slt  $4, $3, $4       # $4 = 12 < 7 = 0        1c         0064202a
           beq  $4, $0, around   # should be taken        20         10800001
           addi $5, $0, 0        # shouldn't happen       24         20050000
around:    slt  $4, $7, $2       # $4 = 3 < 5 = 1         28         00e2202a
           add  $7, $4, $5       # $7 = 1 + 11 = 12       2c         00853820
           sub  $7, $7, $2       # $7 = 12 - 5 = 7        30         00e23822
           sw   $7, 68($3)       # [80] = 7               34         ac670044
           lw   $2, 80($0)       # $2 = [80] = 7          38         8c020050
           j    end              # should be taken        3c         08000011
           addi $2, $0, 1        # shouldn't happen       40         20020001
end:       sw   $2, 84($0)       # write mem[84] = 7      44         ac020054
```

然后用 memory 模块里的激励信号读取一下就行。

接下来是完成仿真文件的编写。使用单周期 MIPS 处理器的仿真文件即可。

```
mips_top M(clk,reset,writedata,dataadr,memwrite);

initial
    begin
        reset<=1;#22;reset<=0;
    end

always
```

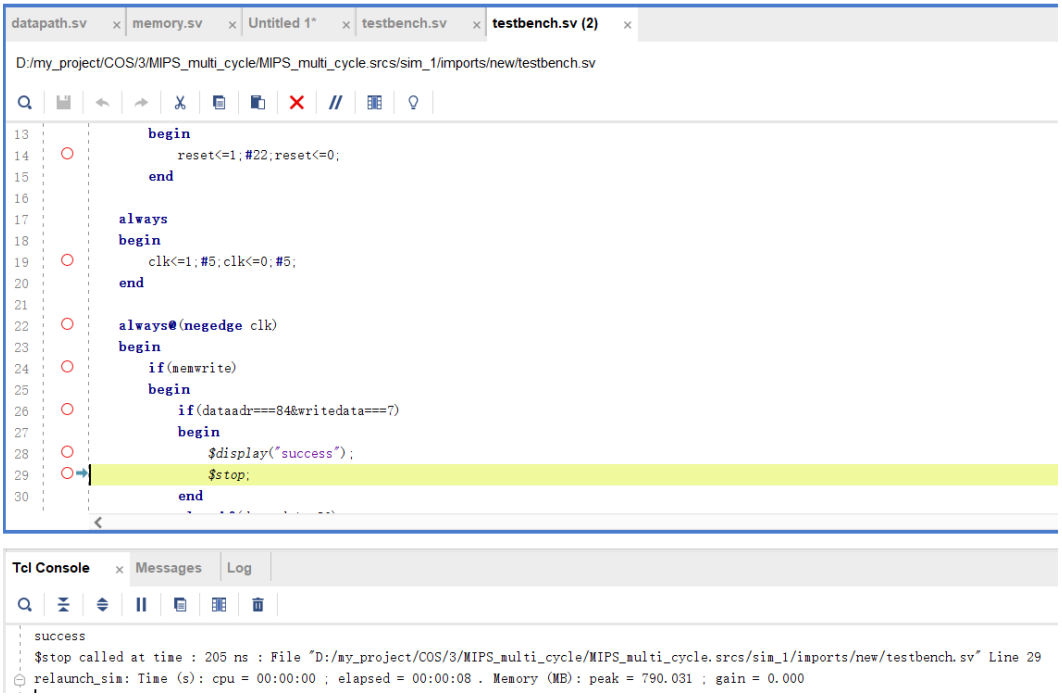
```
begin
    clk<=1;#5;clk<=0;#5;
end

always@(negedge clk)
begin
    if(memwrite)
    begin
        if(dataadr===84&writedata===7)
        begin
            $display("success");
            $stop;
        end
        else if(dataadr!==80)
        begin
            $display("fail");
            $stop;
        end
    end
end
```

至此，完成仿真文件的编写。运行仿真后得到正确的反馈。

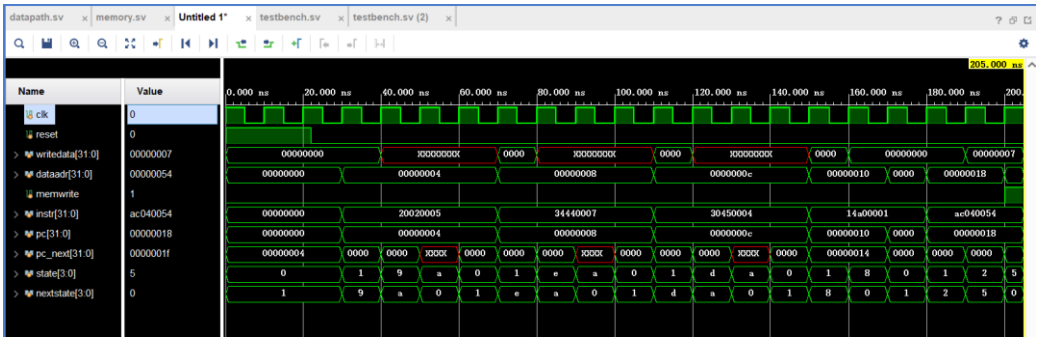
2.3 验证

实验结果

运行仿真后的结果如图，Tcl Console 得到成功的反馈：



其波形图如下：



可以发现其 i 型指令都占据了 4 个周期，符合设计过程。

具体过程

编写支持了更多指令，包括 andi、ori、bne。如果正确执行，它就会在地址为 84 的位置写入数据 7。具体内容如下：

```
# Test the MIPS-Ext processor
# andi, ori, bne
# if successful, it should write the value 7 to address 84
main:   addi $2, $0, 5      # initialize $2 = 5
        ori  $4, $2, 7      # 101 or  111 = 111 ($4)
        andi $5, $2, 4      # 101 and 100 = 100 ($5)
        bne  $5, $0, end    # should be taken
        addi $4, $0, 1      # shouldn't happen $4 = 1
end:    sw $4, 84($0)       # write mem[84] = 7
```

将上述的代码命名为 memfile_moreinstra.dat，用 memory 中的激励信号读取一次即可。

由于判断方式和 2.2 节的标准程序一样，就沿用其仿真文件。运行仿真后得到正确的反馈信息。

3 实验结论

本次实验通过使用 system Verilog 语言，在 vivado 软件上实现了多周期 CPU 的制作，支持 add，sub，addi，subi，and，or，andi，ori，slt，slti，sw，lw，beq，bne，j 指令，并封装为 CPU 模块，运行基础测试代码和扩展指令的测试代码均在仿真后通过测试。

本次设计的重点在于控制器部分，部分模块沿用了实验 2 的单周期 MIPS 处理器的模块。

本次实验的文件命名为 mips_multi_cycle，相关的项目链接如下：

链接：https://pan.baidu.com/s/1yhLgC92JoYlS9_t3qxvMMA
提取码：lab3

---

4 实验感想

本次实验的工作量也是比较大的，但是由于在课上已经学过了多周期 MIPS 处理器的主要设计思想，所以做起来的压力就没那么大，加上前一个实验已经完成了单周期 MIPS 处理器的设计，因此本次多周期处理器可以在单周期的基础上进行开发，就不会像当时设计单周期处理器的时候那样摸着石头过河。

不可否认的是，在完成项目的过程中也遇到了形形色色各种各样的问题，导致一直过不了仿真测试。

有一个问题是出现在 addi 指令的写回阶段，addi 只被分成了 3 段，而不是预想的 4 段。最后在波形图排查阶段，发现是主译码器的有限状态机部分，把 addi 的状态代码在定义的时候输错了，写成 beq 的状态代码了，就导致实际上 addi 执行了 beq 的操作。

虽然导入了很多上个单周期 MIPS 处理器的模块，在数据通路主要做的就是连线和封装的工作，但是还是有出错。因为我为了保证代码的简洁，就选择在实例化的时候不写底层模块的输入输出变量名，直接放入本层模块的变量名，所以会出现一些 not connect 的情况。但是幸运的是这些错误在 RTL 原理图上是一眼就可以看出来的问题。但是当时排查还认为是自己写的带写使能的触发器不能正常锁存，因为 pc 在不该变化的位置变成了 x。在对触发器做测试之后，发现尽管输入变成 x，pc 还是可以正常锁存。最后发现其实是连线错误。

由于沿用了单周期处理器的仿真文件 testbench.sv，但是上一个顶层文件和这次的顶层文件中的变量顺序不一样，导致连线对应的接口不一样。是通过波形图中出现了高阻态的 z 信号，判断有的单个值（WriteEn）与数组（WriteData）相互连接造成的。

在 2.3 部分也出现了类似的问题。由于单周期里扩展了 nop 指令，但是导入的时候没有删除 nop 指令，就导致始终无法正常运行。在删除 nop 后，得以成功运行。

总的说来这次收获还是挺大的，对于多周期 MIPS 处理器有了更深刻的认识。对于多周期 CPU 的硬件实现各种指令的方式都有了比较深入的了解，特别是控制流、数据流相关的内容，以前都是抽象的名词，但是现在是真的理解到了其中的含义，这是不动手做所实现不了的。这次试验对于理解计算机软硬件交接的层面的有着极大的帮助。

对于 verilog 设计方面，已经掌握了各种排查问题的手段，比如观察 RTL 图、添加变量看波形图、以及最低效但有用的分析代码文本。