2022 春季学期

计算机组成原理

MIPS 多周期 CPU

设

计

报

告

学院：信息科学与工程学院

组员：陈昊（计一） 吴国维（计二）

一、项目简述

设计语言：Verilog 硬件描述语言
仿真环境：Vivado 2019.2
频率：25MHZ
实现指令： 11 条 MIPS 指令
CPI：3-5

二、设计原理

多周期 CPU 将整个 CPU 的执行过程分成几个阶段，每个阶段用一个时钟去完成。在多周期 CPU 中，一个指令周期可以由数个时钟周期组成，不同的指令可以占用不同的时钟周期数，从而提高了时间片的利用率以及 CPU 的主频。

多周期 CPU 在处理指令时，一般需要经过以下几个步骤：

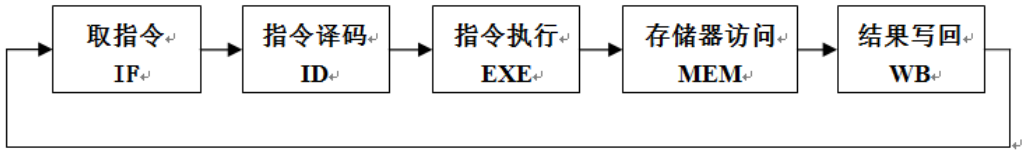
(1) **取指令**：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，同时，PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 PC，当然得到的“地址”需要做些变换才送入 PC。

(2) **指令译码**：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

(3) **指令执行**：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(4) **存储器访问**：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

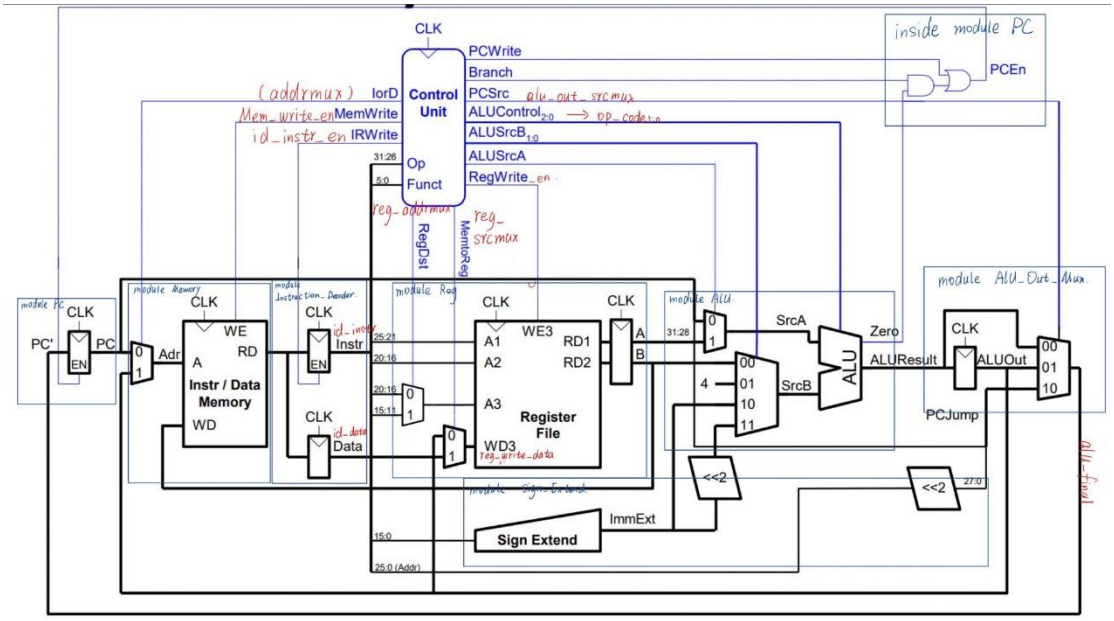
(5) **结果写回**：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。



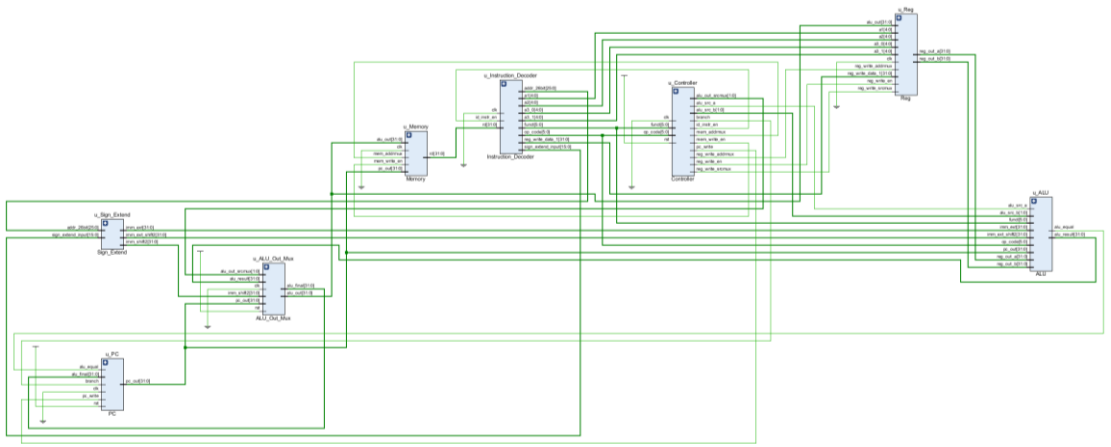
三、设计内容

1、总体结构图

结构图：在教学 PPT 的结构基础上进行修改。修改用红色部分注明，蓝色框内为设计的各个模块。



Vivado 中生成的 schematic 图：



包含模块：Controller，PC，Instruction_Decoder，Memory，Reg，ALU，ALU_Out_Mux，Sign_Extend.

本 CPU 采用冯诺依曼架构，指令与数据共享同一个存储器。

2、各模块具体功能分析

（1）、PC

PC 模块中有一个 31:0 的 reg 存储器，用来存储当前指令（在状态机状态不为 s0 时为下一条指令）的地址。接受一个 31:0 的来自 alu 的输入作为下一条指令的地址。同时接受各种控制信号的输入以决定 PC 的值是否改变。

```
module PC(
    input pc_write,
    input [31:0] alu_final,
    input branch,
    input alu_equal,
    input clk,
    input rst,
    output reg [31:0] pc_out
);
```

(2)、Instruction_Decoder

Instruction_Decoder 模块有两个作用：一是存储当前指令的值（32 位），二是对当前指令解码，产生各种解码信号，传到 CPU 的各个其他模块以便使用。

```
module Instruction_Decoder(
    input clk,
    input id_instr_en,
    input [31:0] rd,
    output [4:0] a1,
    output [4:0] a2,
    output [4:0] a3_0,
    output [4:0] a3_1,
    output [31:0] reg_write_data_1,
    output [5:0] funct,
    output [4:0] shamt,
    output [5:0] op_code,
    output [15:0] sign_extend_input,
    output [25:0] addr_26bit
);
```

(3)、Memory

Memory 模块是 memory 的存储模块，可以存储 128 个字节的数据。使用 reg 进行存储。接受来自 PC 和 ALU 的地址信号和来自 controller 的控制信号、来自 register 的数据信号，输出一个 32 位的数据。

本 CPU 的指令和数据共享同一个 memory。

```
module Memory(
    input [31:0] pc_out,
    input [31:0] alu_out,
    input mem_addrmux,
    input clk,
    input mem_write_en,
    input [31:0] wd,
    output [31:0] rd
);
```

、

(4)、Controller

Controller 模块接受 opcode 和 funct 输入，结合当前状态输出控制信号至 CPU 各模块。内部为状态机。该状态机共有 13 个状态，其中状态 0 为取值，状态 1 为译码（每个指令都需经过这两步），状态 15 为 rst 触发后的初始状态。

Controller 模块会根据不同的状态输出不同的控制信号。

```
module Controller (
        input clk,
        input rst,
        input [5:0] op_code,
        input [5:0] funct,
        output reg pc_write,
        output reg branch,
        output reg reg_write_en,
        output reg reg_write_srcmux,
        output reg reg_write_addrmux,
        output reg mem_write_en,
        output reg mem_addrmux,
        output reg alu_src_a,
        output reg [1:0] alu_src_b,
        output reg [2:0] alu_ctrl,
        output reg [1:0] alu_out_srcmux,
        output reg id_instr_en,
        output reg [3:0] current_state
    );
```

(5)、Reg

Reg 模块内部有 32 个 32 位寄存器，用于 CPU 计算前后的临时结果储存。接受 4 组地址输入和两个 32 位数据输入，通过两个数据选择器信号输入选择正确的地址输入，可同时输出两个 32 位的数据。

```
module Reg(
        input clk,
        input [4:0] a1,
        input [4:0] a2,
        input [4:0] a3_0,
        input [4:0] a3_1,
        input [31:0] alu_out,
        input [31:0] reg_write_data_1,
        input reg_write_en,
        input reg_write_srcmux,
        input reg_write_addrmux,
        output reg [31:0] reg_out_a,
        output reg [31:0] reg_out_b
    );
```

(6)、Sign_Extend

Sign_Extend 模块用于拓展立即数，接受两种不同的立即数输入，将其拓展（并移位）为三种不同的 32 位立即数。

```
module Sign_Extend(
    input [15:0] sign_extend_input,
    input [25:0] addr_26bit,
    output [31:0] imm_ext,
    output [31:0] imm_ext_shift2,
    output [31:0] imm_shift2    // for J type
);
```

(7)、ALU

算数逻辑计算单元。接受来自 reg、Sign_Extend 的操作数，根据控制信号选择对应的计算模式，输出计算结果（overflow、equal 和 32 位的 result）。

```
module ALU(
    input [5:0] op_code,
    input alu_src_a,    //controller signals
    input [1:0] alu_src_b,
    input [31:0] pc_out,
    input [31:0] reg_out_a,
    input [31:0] reg_out_b,
    input [31:0] imm_ext,
    input [31:0] imm_ext_shift2,
    input [5:0] funct,
    output reg alu_equal,
    output reg [31:0] alu_result,
    output reg alu_overflow
);
```

(8)、ALU_Out_Mux

该模块用于处理 ALU 的计算结果，根据控制信号，将计算结果按照指定的时间传

输到各个模块（PC、Memory）中。同时在 J 型指令中，该模块将 PC 的低 28 位用立即数替代，并送回 PC 的输入信号。

```
module ALU_Out_Mux(
    input clk,
    input rst,
    input [31:0] alu_result,
    input [31:0] pc_out,
    input [1:0] alu_out_srcmux,
    input [31:0] imm_shift2,
    output reg [31:0] alu_out,
    output reg [31:0] alu_final
);
```

四、指令实现情况

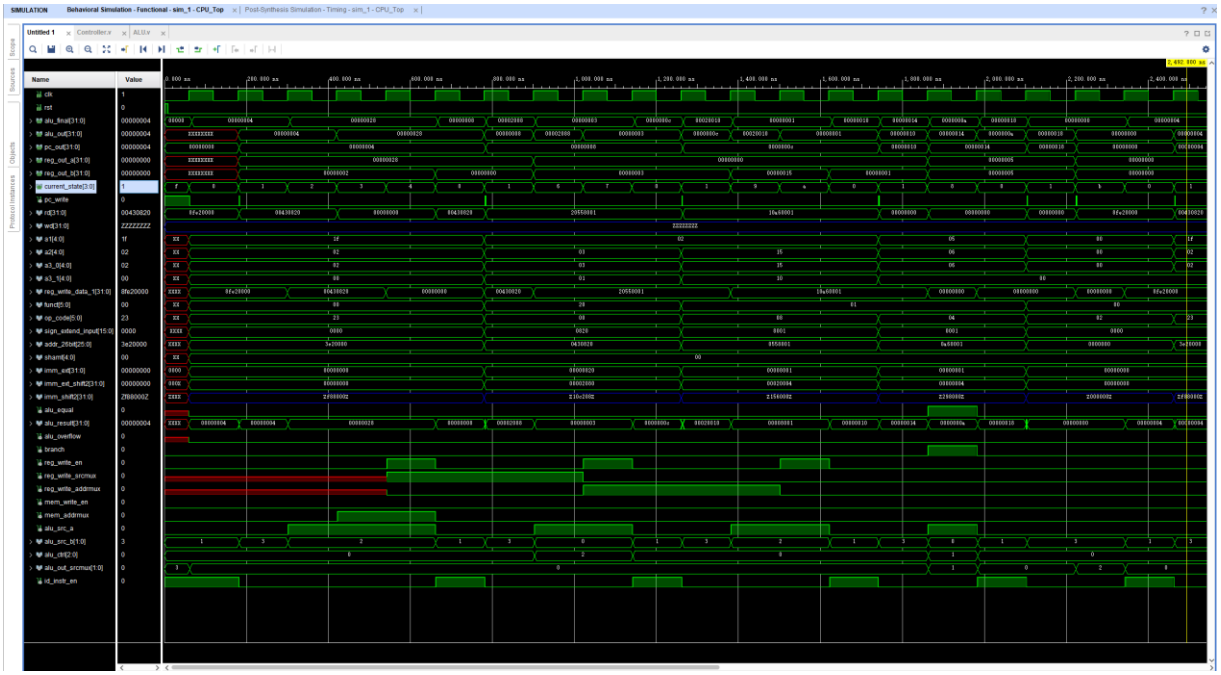
本设计覆盖了 R 型（6 条）、I 型计算类、I 型取数类、I 型存数类、I 型条件判断类、J 型，共 11 条指令。

R 型	Opcode	funct	shamt	描述
ADD	000000	100000	00000	rd ← rs + rt
SUB	000000	100010	00000	rd ← rs - rt
AND	000000	100100	00000	rd ← rs AND rt
OR	000000	100101	00000	rd ← rs OR rt
XOR	000000	100110	00000	rd ← rs XOR rt
NOR	000000	100111	00000	rd ← rs NOR rt
I 型	Opcode			描述
ADDI	001000			rt ← rs + immediate
LW	100011			rt ← memory[base+offset]
SW	101011			memory[base+offset] ← rt
BEQ	000100			if (rs = rt) then branch
J 型	Opcode			
J	000010			

五、仿真测试

（若要重新运行仿真，请将 Memory.v 文件中 45 行的 readmemh 指令指向正确的"ireg.txt"路径）

行为仿真：（总图）

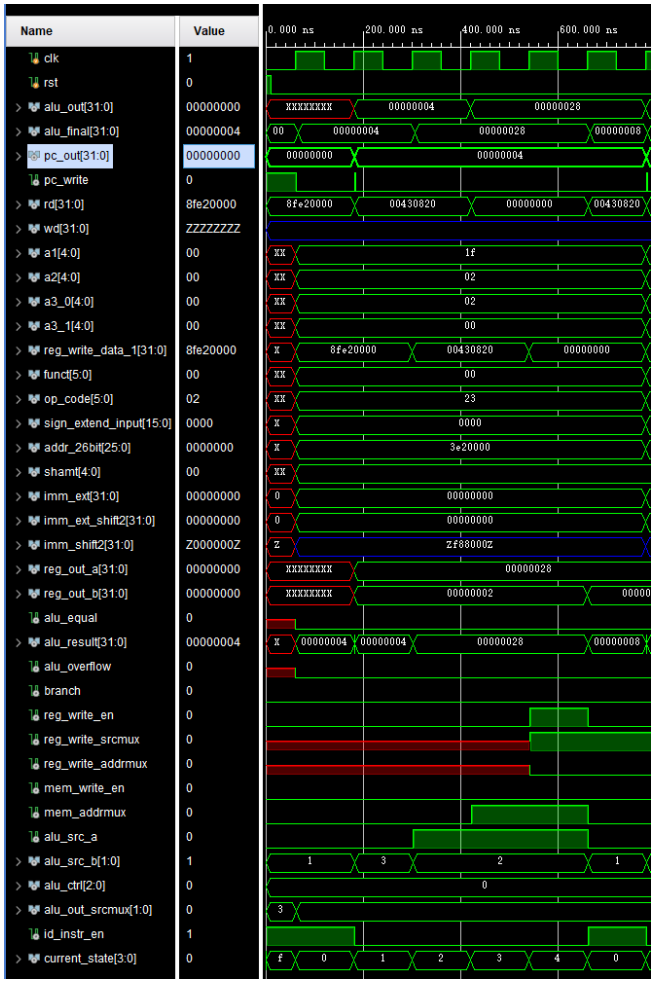


功能仿真：（总图）

根据测试，该 CPU 在 clk<20ns 时，功能仿真出现了 PC 更新滞后一个指令周期的情况。故 CPU 的最大运行频率为 25MHZ。

各指令波形图分析：

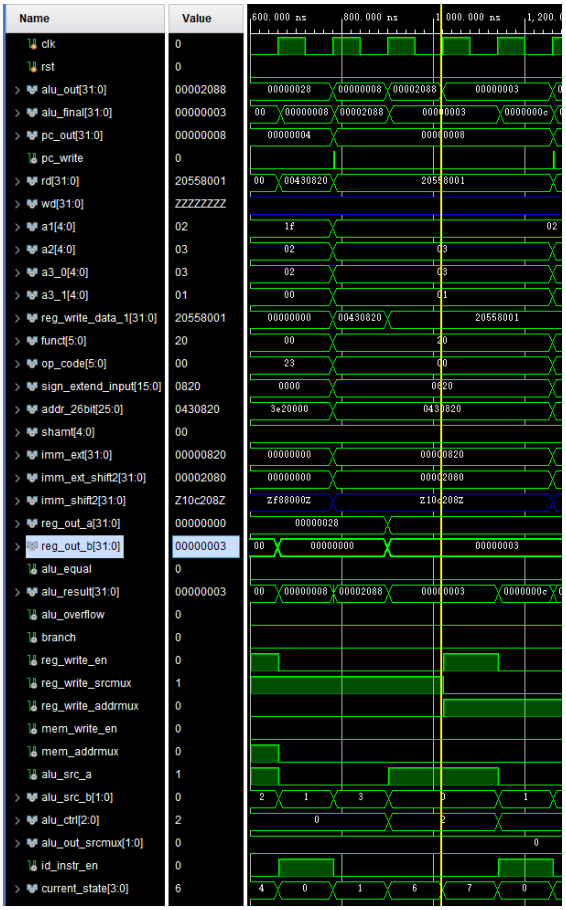
(1).LW 指令：8f e2 00 00 （I 型取数类）



状态变化：0->1->2->3->4->0

可以看到，在状态 2 时，寄存器 a1 中的值和立即数相加得出结果（alu_out）；状态 3 时使用相加得到的地址访问 memory；状态 4 时将从 memory 中取到的数写入 register 中。

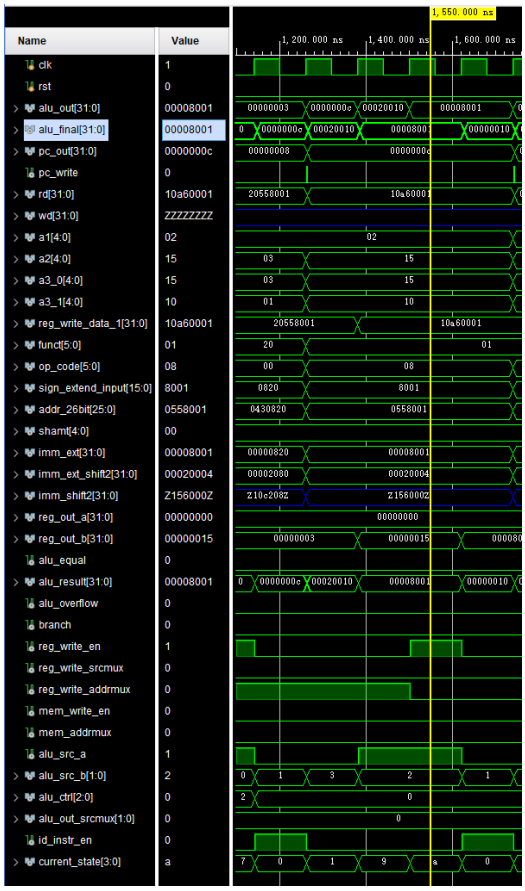
（2）.ADD 指令：00 43 08 20 （R 型）



状态变化：0->1->6->7->0

状态 1 时进行 pc+4（结果在 alu_final 中，并在本阶段写入 pc 中的寄存器）和译码操作。状态 6 时从 register 中取出操作数 1 和操作数 2（此时可以看到操作数 1，即第二个寄存器的值为 32'b0，和初始值不同，这可以验证第一个 LW 指令确实将 memory 中的值读入 register 中了），并进行 ALU 运算操作（结果为 alu_result）。状态 7 将运算结果写回 register 中。

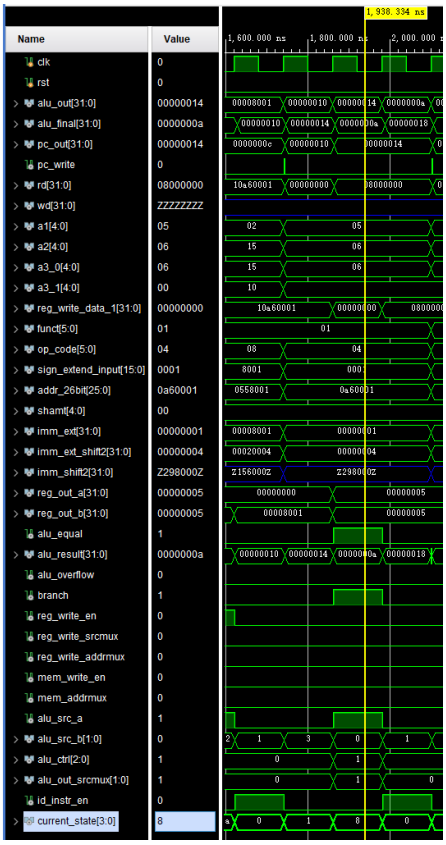
（3）.ADDI 指令：20 55 80 01 （I 型计算类）



状态变化：0->1->9->10->0

状态 1 时进行 pc+4（结果在 alu_final 中，并在本阶段写入 pc 中的寄存器）和译码操作。状态 9 将 register[a1]中的数（0x2）与立即数(0x00008001)相加，得到结果存在 alu_result 中（0x00008003）。状态 10 执行写回操作，将运算结果写回 register。

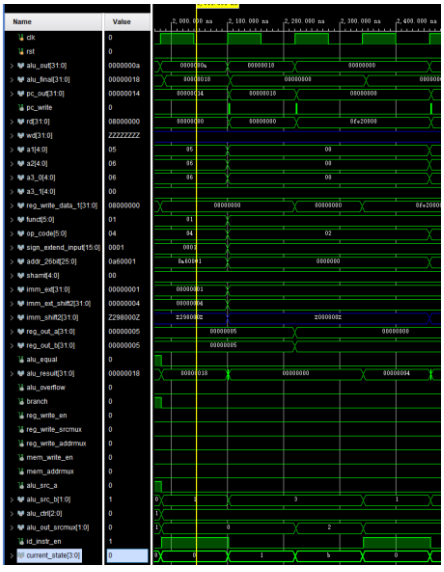
（4）.BEQ 指令： 10 a6 00 01 （I 型判断类）



状态变化：0->1->8->0

状态 1 译码与 PC+4。状态 8 时，ALU 模块判断 register[5]和 register[6]中的值是否相等（两者均为 0x5，相等），于是 ALU 输出 branch=1. Sign_Extend 模块将立即数左移两位（0x1 -> 0x4），与当前 PC 相加得到新的 PC 值。即 PC 在当前基础上+4，跳过 memory 中的 32bit 数据。

（5）. J 指令： 08 00 00 00 （J 型）



状态变化：0->1->11->0

状态 1 译码、PC+4. 状态 11 时，将 26 位立即数左移两位，变成 28 位并赋给 PC 的低

28 位。在此条指令中，立即数为 26'h000000，在该状态结束时 PC 为 32'h0。重新开始执行第一条指令，不断循环。

六、设计中遇到的问题

1、仿真时报错：”[Timing 38-12] [SDFWriter-1] No valid delays were found for the entire design. Please check the input design for valid instances and ensure that Vivado implementation ran successfully before calling write_sdf.”

在功能仿真时出现该问题。经过搜索后发现，该问题的原因是顶层模块中没有输出导致的。由于 vivado 在布线时有自动优化机制，若顶层模块没有输出，则程序判定该模块无任何功能。

解决方案：在顶层模块(CPU_Top)中添加任意输出。

2、初始化问题

在设计之初时，我试图将状态机的初始状态设为 s0，即和取指状态共用一个状态。但在仿真时我发现，由于 PC 的初始化不能在 controller 状态为“取指令”时进行（s0 状态下，pc 应在 alu 模块中被+4，但初始化时执行该操作会产生错误），故该设计是有问题的。

解决方案：加入一个新的状态“置零状态”，它的下一状态为 s0.

3、临界值的问题

该问题在单周期 CPU 的设计时并未出现，但在本次设计中尤为严重。例如：某个控制信号在 posedge clk 时发生改变（比如从 0 ->1），而某个被该信号控制的模块中，在 posedge clk 时会根据该信号的值决定某个寄存器的赋值结果。该信号的值在该时刻到底会被判定为 0 还是 1 呢？这个执行结果在行为仿真和功能仿真时可能是不一样的。

解决方案：按照信号的流动方向，对各个类似的判断语句划定层级，通过#语句手动延迟判定的时间，错开信号变化的临界值；比如让控制信号在 posedge clk 后#1 发生变化，而在 posedge clk 后#2 才进行信号值的判定。

七、总结

多周期 CPU 虽然相较于单周期 CPU 在各部分功能、结构上没有非常大的变化，但是由于数据通路的变化和单个时钟周期执行目标的改变，两个 CPU 的设计内部细节相差还是很多的。

本小组设计的 MIPS 指令集多周期 CPU 可执行 R 型（6 条）、I 型计算类、I 型取数类、I 型存数类、I 型条件判断类、J 型，共 11 条指令，运行频率为 25MHZ。由于运行频率的提高和不必要状态的时间减少，该 CPU 在大多数指令的运行时间上，相较于设计结构类似的单周期 CPU 有了明显的提高。

本组的单周期 CPU 设计为哈佛结构，而多周期则改为了冯诺依曼结构。相比于哈佛结构，冯诺依曼结构有许多优势：比如，可以轻易的通过 SW 指令，对指令本身进行修改，使得程序在执行过程中更加灵活；同时，合并两个存储器也可以优化存储资源的配置，防止浪费的情况。

通过本次对多周期 CPU 的设计，我们对 CPU 的结构有了更好的掌控，在设计过程中遇到问题时的应对措施和解决方案也更加娴熟，相较于单周期 CPU 的设计时，设计能力有了较大的提升。