组成原理课程第 六 次实报告

实验名称：多周期 CPU

学号： 2310420 姓名： 王晶 班次： 周二上午三四节

实验目的

1.在单周期 CPU 实验完成的提前下，理解多周期的概念。

2.熟悉并掌握多周期 CPU 的原理和设计。

3.进一步提升运用 verilog 语言进行电路设计的能力。

4.为后续实现流水线 cpu 的课程设计打下基础

实验内容

1、多周期 CPU 实验使用同步 IP 核构造 data_ram 和 inst_rom，原始 source_code 中的同名.v 文件和 ngc 文件不要导入到项目。

2、多周期 CPU 运行的指令在 inst_rom 中，这里面的指令须导入 coe 文件。

3、请把 ALU 实验中添加的三个运算，自行定义类似 MIPS 指令格式的指令，把对应的指令和功能增加到多周期 CPU 中，并自行在 coe 文件中添加指令，然后进行运行验证（仿真波形验证或实验箱验证即可）。

4、实验报告中可以不放原理图，关于验证结果的图片（仿真图片或实验箱图片）需要仔细介绍图中的信息和对指令验证的情况

实验原理图

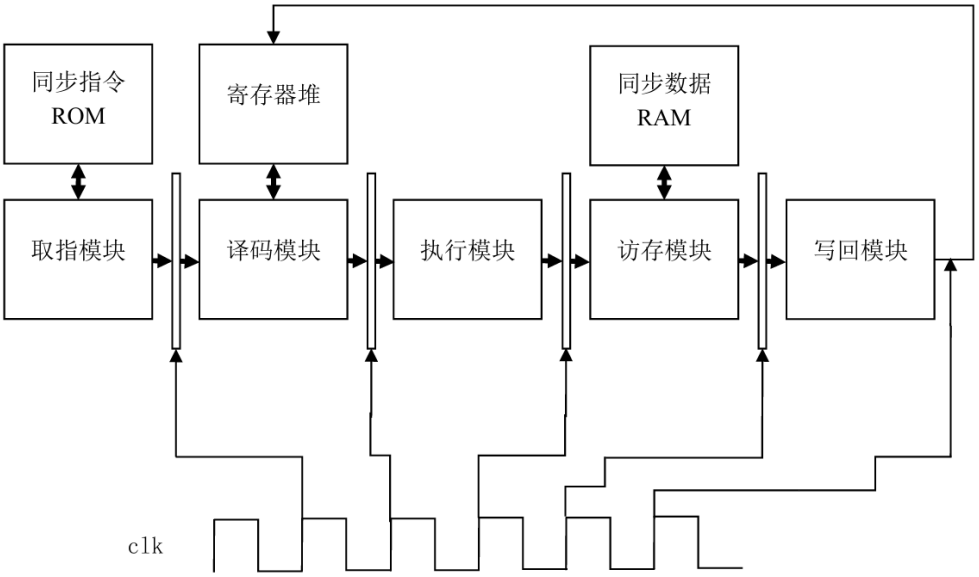


图 8.1 多周期 CPU 的大致框图

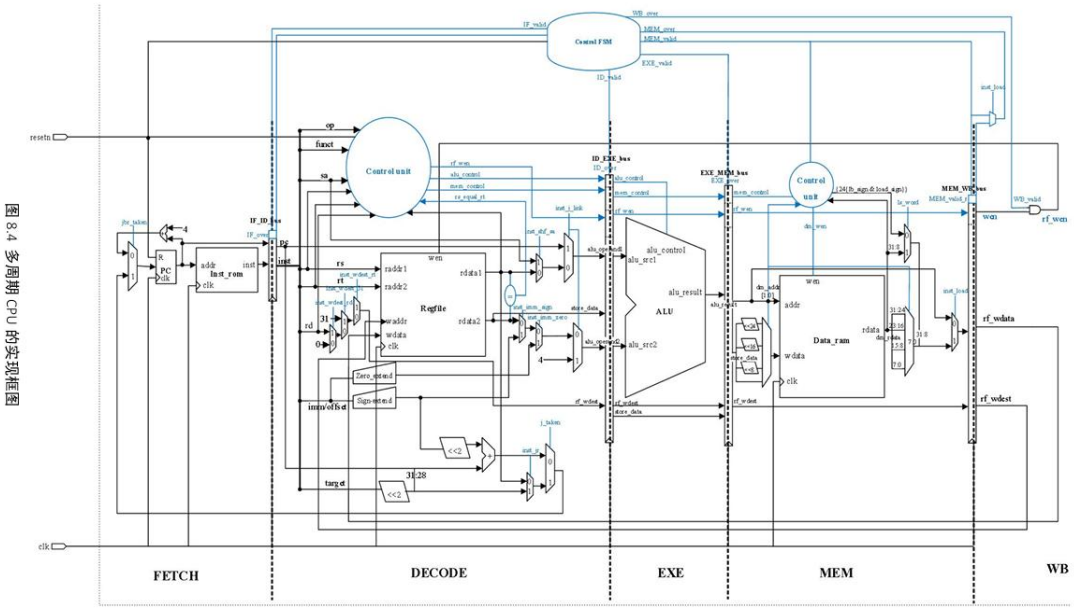


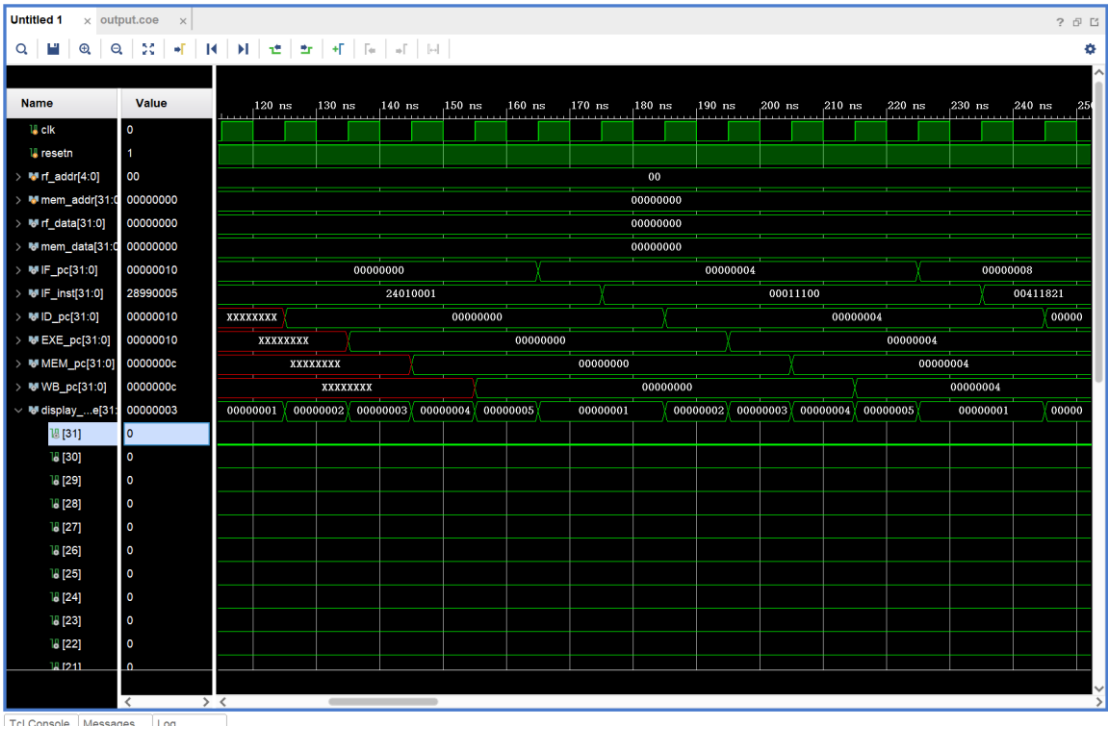
图 8.4 多周期 CPU 的实现框图

实验步骤

（1）修改代码的 bug

需要把 data_ram 的端口改为 true dual port

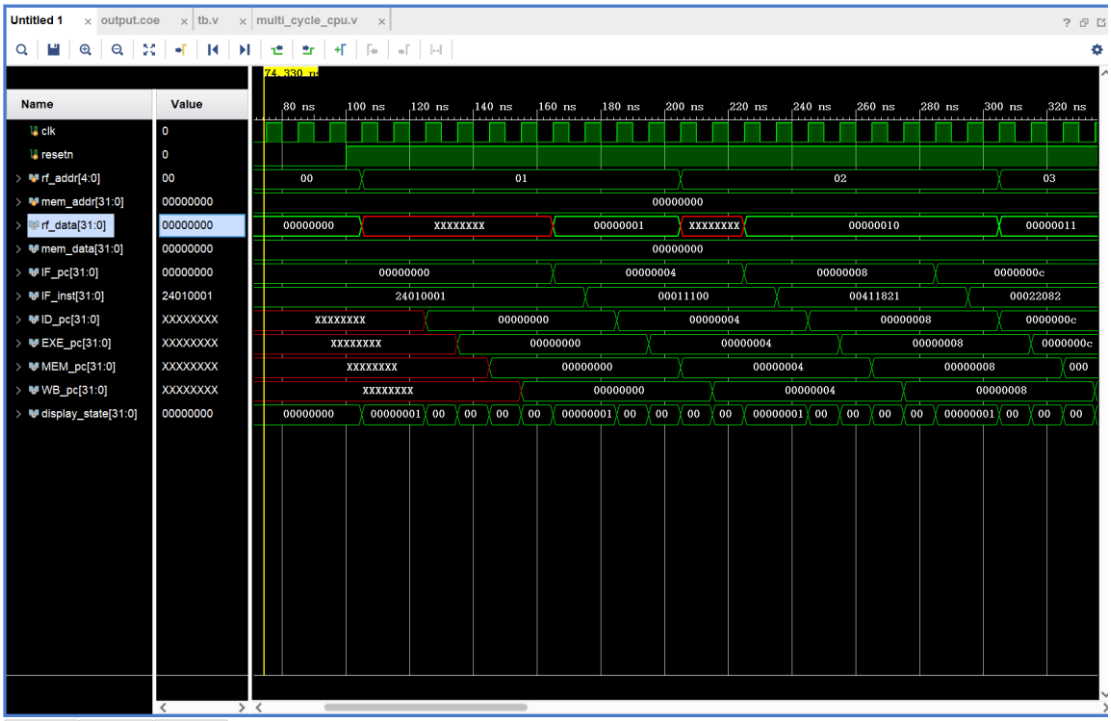
同时仿真时发现，rf_addr 和 rf_data 在任何周期均为 0，如图，可能是因为仿真时没有给其激励信号

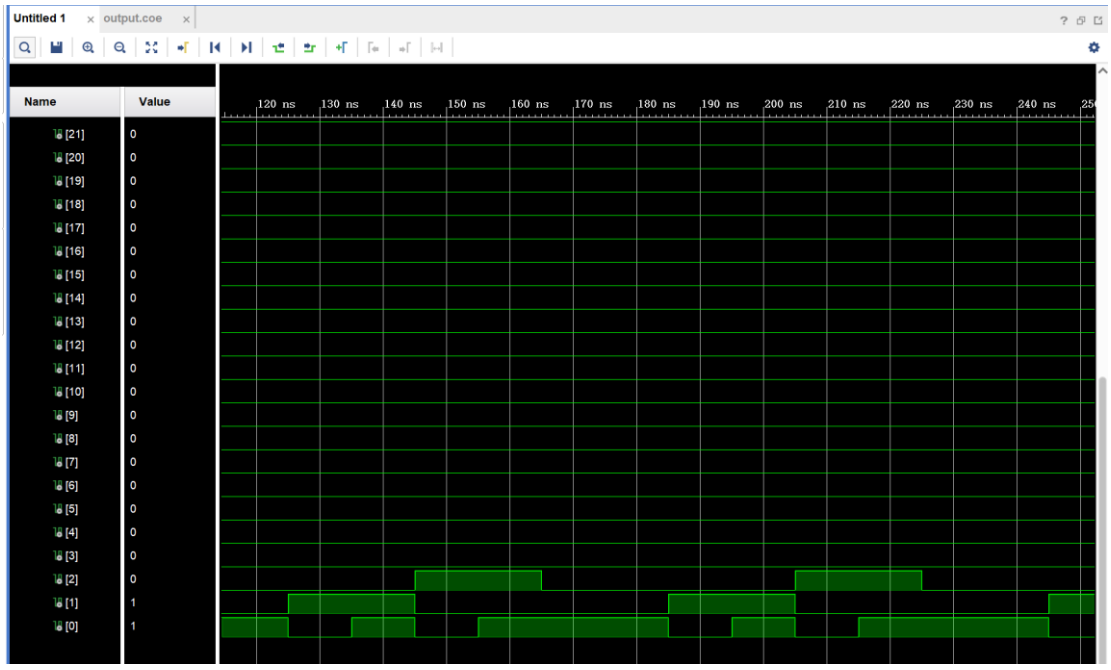


我们修改 tb.v 的代码，给其加上变化

```
reg [31:0] counter;
    always @(posedge clk) begin
        if (!resetn) begin
            counter <= 0;
            rf_addr <= 0;
        end else begin
            if (counter % 10 == 0) begin
                rf_addr <= rf_addr + 1;
            end
            counter <= counter + 1;
        end
    end
```

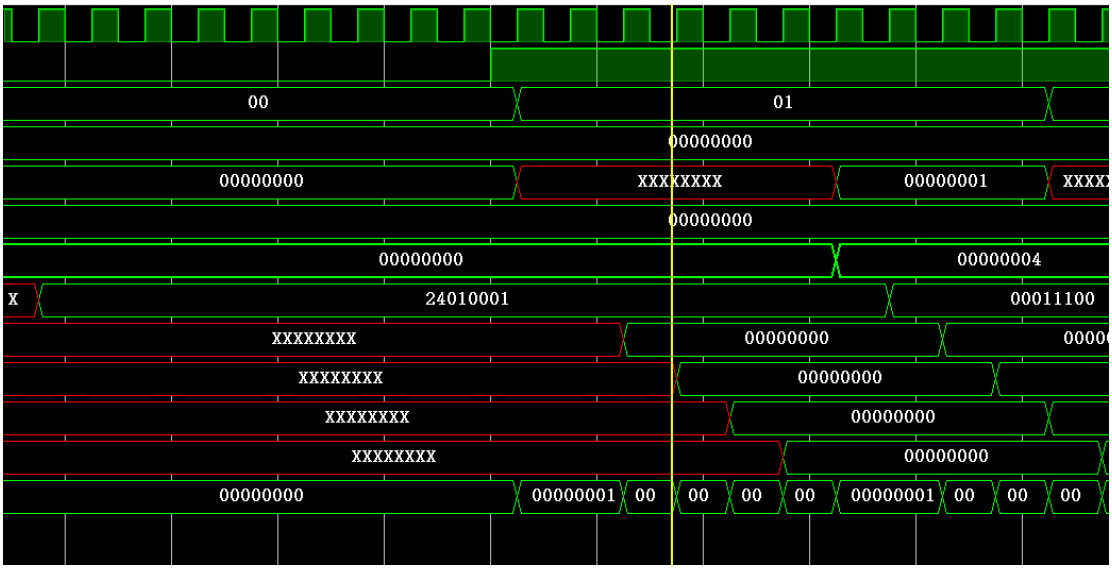
如下图，可以看到此时 rf_data 和 rf_addr 的值每个指令执行后出现变化，我们修改正确

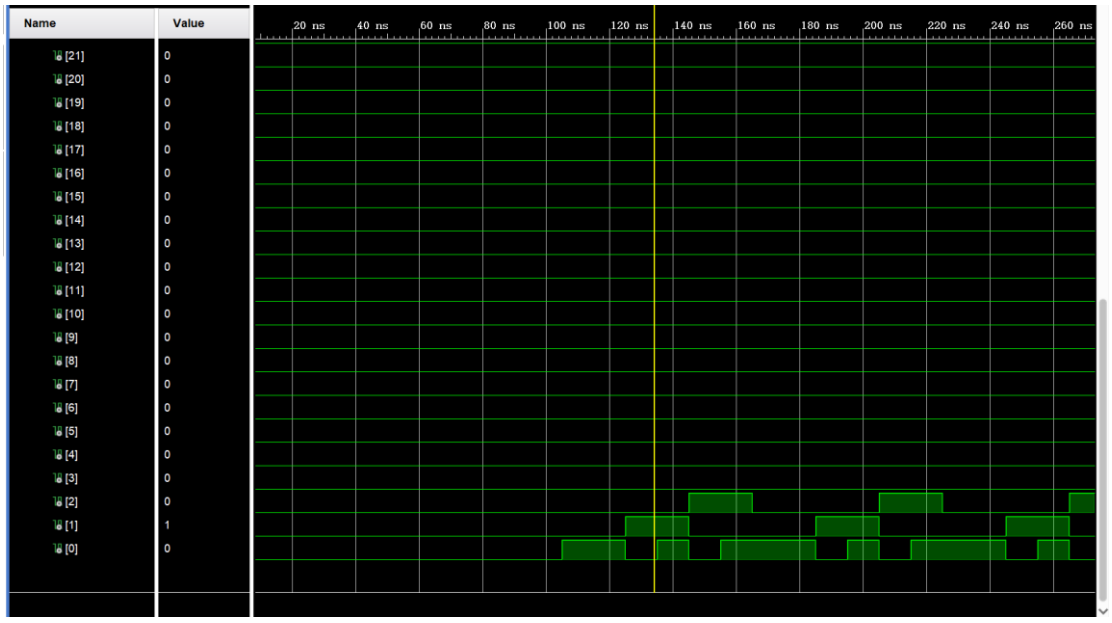




（2）分析指令执行过程

2.1 指令一





2.1.1 取指阶段（IF）

从 inst_rom 取指，将其存入 IF_ID_bus，准备送往 ID 级。inst_addr = PC

inst_rom 返回指令 24010001，IF_ID_bus = {PC, inst_rom_out} → {0x00000000, 24010001}，IF_pc = PC → 0x00000000，IF_inst = 24010001

IF_over = 1（指令已获取完成）

2.1.2 译码阶段（ID）

解析指令格式，获取操作数，并生成 ID_EXE_bus 供下阶段执行。rs = 0（源寄存器）rt = 1（目标寄存器）immediate = 0x0001（立即数）rs_value = 0（因为 $0 恒为 0）rt_value = 无需读取 jbr_not_link = 0（非跳转指令）

ID_EXE_bus = {rs, rt, immediate, rs_value} → {0, 1, 0x0001, 0}

ID_pc = PC → 0x00000000，ID_over = 1（译码完成）

2.1.3 执行阶段（EXE）

进行 ALU 计算，将 $0 + 1 存入 $1。ALU_result = rs_value + immediate = 0 + 0x0001 = 0x0001，EXE_MEM_bus = {rt, ALU_result} → {1, 0x0001}

EXE_pc = PC → 0x00000000，EXE_over = 1（执行完成）

2.1.4 访存阶段（MEM）

addi 不涉及访存，直接传递数据。MEM_WB_bus = EXE_MEM_bus = {1, 0x0001}

MEM_pc = PC → 0x00000000，MEM_over = 1（访存完成）

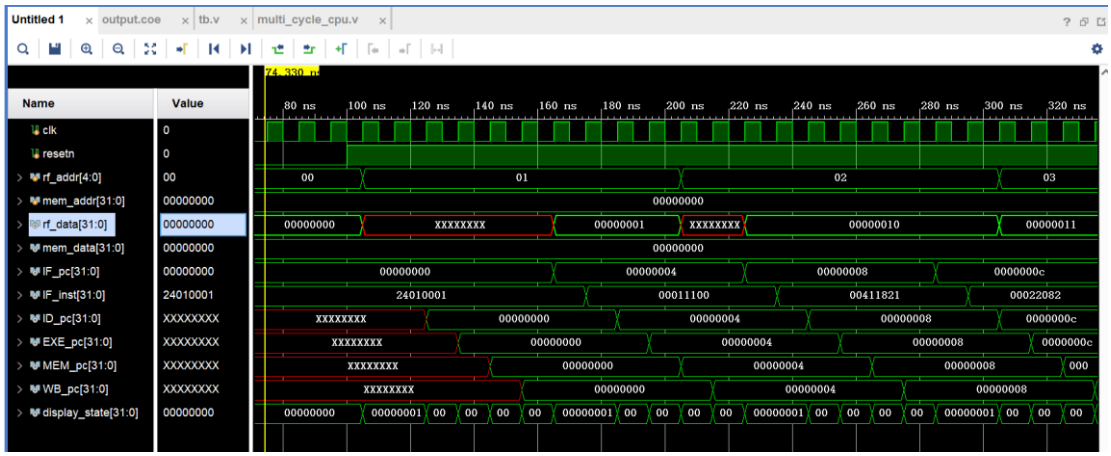
2.1.5 写回阶段（WB）

将 0x0001 写入寄存器 $1。rf_wen = 1（写入使能）rf_wdest = 1（目标寄存器）rf_wdata = MEM_WB_bus[1] = 0x0001，WB_pc = PC → 0x00000000，WB_over = 1（写回完成）

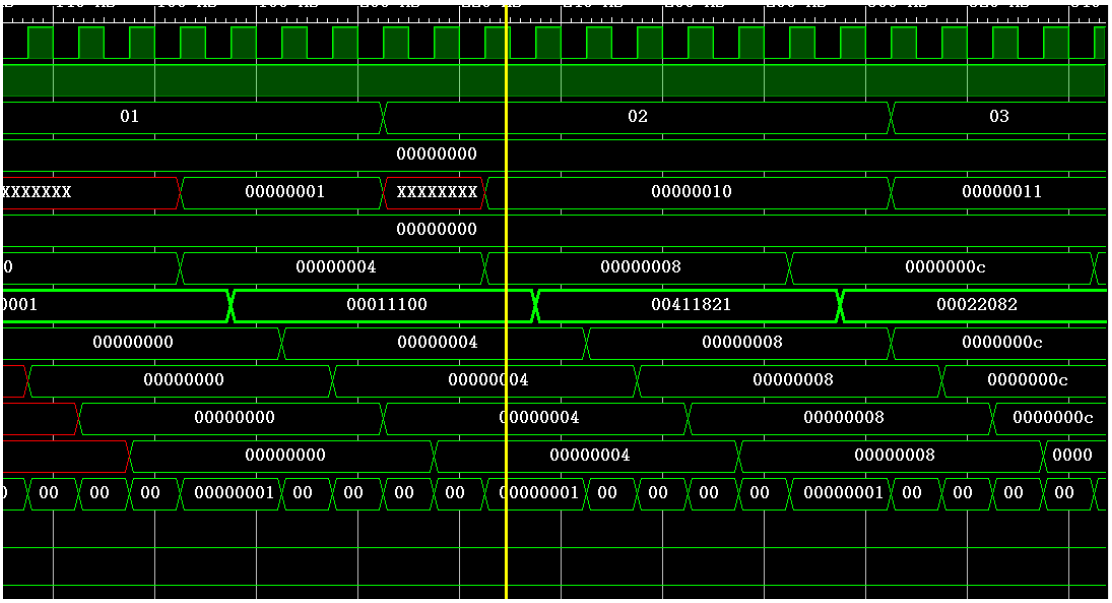
2.1.6 最终结果

寄存器 $1 被赋值 0x0001

如图，可以看到 rf_addr 为 1 时，rf_data 为 00000001



2.2 指令二



2.2.1 取指阶段（IF）

从 inst_rom 取指，将指令存入 IF_ID_bus，准备送往 ID 级。inst_addr = PC （假设 PC = 0x00000004）inst_rom 返回指令 00011100，IF_ID_bus = {PC, inst_rom_out} → {0x00000004, 00011100}，IF_pc = PC → 0x00000004，IF_inst = 00011100，IF_over = 1（取指完成）

2.2.2 译码阶段（ID）

解析指令格式，获取操作数，并生成 ID_EXE_bus 供下阶段执行。rs = 1（源寄存器）rd = 2（目标寄存器）shamt = 4（移位位数）funct = sll（操作类型），rs_value = regfile[$1] = 0x00000001，ID_EXE_bus = {rs, rd, shamt, rs_value} → {1, 2, 4, 0x00000001}，ID_pc = PC → 0x00000004，ID_over = 1（译码完成）

2.2.3 执行阶段（EXE）

进行 ALU 计算，左移 $1 的值 4 位，并存入 $2。ALU_result = rs_value << shamt = 0x00000001 << 4 = 0x00000010，EXE_MEM_bus = {rd, ALU_result} → {2, 0x00000010}，EXE_pc = PC → 0x00000004，EXE_over = 1（执行完成）

2.2.4 访存阶段（MEM）

sll 不涉及访存，直接传递数据。

MEM_WB_bus = EXE_MEM_bus = {2, 0x00000010}，MEM_pc = PC → 0x00000004

MEM_over = 1（访存完成）

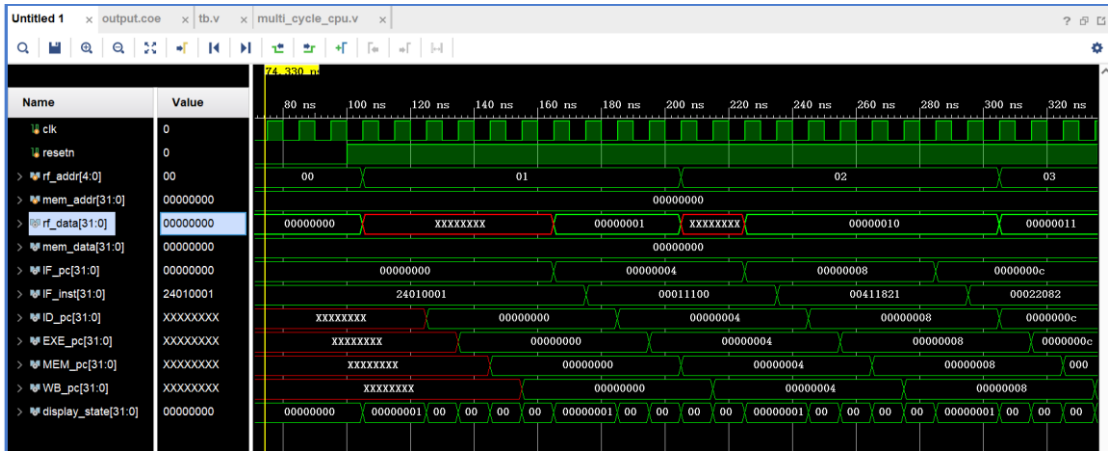
2.2.5 写回阶段（WB）

将 0x00000010 写入寄存器 $2。rf_wen = 1（写入使能）rf_wdest = 2（目标寄存器）rf_wdata = MEM_WB_bus[1] = 0x00000010，WB_pc = PC → 0x00000004，WB_over = 1（写回完成）

2.2.6 最终结果

寄存器 $2 被赋值 0x00000010

如图，可以看到 rf_addr 为 2 时，rf_data 为 000000010



（3）增加指令

我们在下面九条指令后面增加三条指令

00H	addiu $1, $0,#1	[$1] = 0000_0001H	24010001	0010_0100_0000_0001_0000_0000_0000_0001
04H	sll $2, $1,#4	[$2] = 0000_0010H	00011100	0000_0000_0000_0001_0001_0001_0000_0000
08H	addu $3, $2,$1	[$3] = 0000_0011H	00411821	0000_0000_0100_0001_0001_1000_0010_0001
0CH	srl $4, $2,#2	[$4] = 0000_0004H	00022082	0000_0000_0000_0010_0010_0000_1000_0010
10H	slti $25,$4,#5	[$25] = 0000_0001H	28990005	0010_1000_1001_1001_0000_0000_0000_0101
14H	bgez $25,#16	跳转到 54H	07210010	0000_0111_0010_0001_0000_0000_0001_0000
18H	subu $5, $3,$4	[$5] = 0000_000DH	00642823	0000_0000_0110_0100_0010_1000_0010_0011
1CH	sw $5, #20($0)	Mem[0000_0014H] = 0000_000DH	AC050014	1010_1100_0000_0101_0000_0000_0001_0100
20H	nor $6, $5,$2	[$6] = FFFF_FFE2H	00A23027	0000_0000_1010_0010_0011_0000_0010_0111

增加为

nxor $3,$1,$2

Sgt $3,$1,$2

Sgtu $3,$1,$2

（4）代码修改

alu.v 文件修改为：

将 alu_control 控制信号改成 13 位，

```
input [14:0] alu_control, // ALU 控制信号
```

加上下面的代码，实现信号的声明\控制信号的赋值\结果信号的声明\结果信号的赋值

```
wire alu_sgt;
wire alu_sgtu;
wire alu_nxor;
assign alu_sgt=alu_control[12];
assign alu_sgtu=alu_control[13];
assign alu_nxor=alu_control[14];
wire [31:0] sgt_result;
wire [31:0] sgtu_result;
wire [31:0] nxor_result;
assign sgt_result  = ($signed(alu_src1) > $signed(alu_src2)) ?
```

```
32'd1 : 32'd0;
assign sgtu_result = (alu_src1 > alu_src2) ? 32'd1 : 32'd0;
assign nxor_result = ~(alu_src1 ^ alu_src2);
```

将结果信号加入其中,以便根据不同的控制信号选择不同的运算结果,赋值给 alu_result 信号

```
assign alu_result = (alu_add|alu_sub) ? add_sub_result[31:0] :
                        alu_slt           ? slt_result :
                        alu_sltu          ? sltu_result :
                        alu_and           ? and_result :
                        alu_nor           ? nor_result :
                        alu_or            ? or_result  :
                        alu_xor           ? xor_result :
                        alu_sll           ? sll_result :
                        alu_srl           ? srl_result :
                        alu_sra           ? sra_result :
                        alu_lui           ? lui_result :
                        alu_sgt           ? sgt_result :
                        alu_sgtu           ? sgtu_result :
                        alu_nxor          ? nxor_result:
                        32'd0;
```

decode.v 的修改

```
wire inst_SGT,inst_SGTU,inst_NXOR;
assign inst_SGT  = (op == 6'b000000) && (funct == 6'b101010);  //
有符号大于
    assign inst_SGTU = (op == 6'b000000) && (funct == 6'b101011);
// 无符号大于
    assign inst_NXOR = (op == 6'b000000) && (funct == 6'b101100);
// 异或后取反
```

```
assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT  |
inst_SLTU
                        | inst_JALR | inst_AND  | inst_NOR  |
inst_OR
                        | inst_XOR  | inst_SLL  | inst_SLLV |
inst_SRA
```

```
                          | inst_SRAV | inst_SRL  |
inst_SRLV|inst_SGT|inst_SGTU|inst_NXOR;
```

```
assign alu_control = {inst_add,        // ALU 操作码，独热编码
                      inst_sub,
                      inst_slt,
                      inst_sltu,
                      inst_and,
                      inst_nor,
                      inst_or,
                      inst_xor,
                      inst_sll,
                      inst_srl,
                      inst_sra,
                      inst_lui,
                      inst_sgt,
                      inst_sgtu,
                      inst_nxor};
```

inst_rom 的写入

```
00000000001000100001100000100110,
00000010010100010001100000101010,
00000010010100010001100000101011,
```

exe.v 文件修改

```
input              EXE_valid,   // 执行级有效信号
    input      [152:0] ID_EXE_bus_r,// ID->EXE 总线
    output             EXE_over,    // EXE 模块执行完成
    output     [105:0] EXE_MEM_bus, // EXE->MEM 总线
```

multi_cycle_cpu.v 的修改

```
wire [ 63:0] IF_ID_bus;   // IF->ID 级总线
    wire [152:0] ID_EXE_bus;  // ID->EXE 级总线
    wire [105:0] EXE_MEM_bus; // EXE->MEM 级总线
    wire [ 69:0] MEM_WB_bus;  // MEM->WB 级总线

    //锁存以上总线信号
    reg [ 63:0] IF_ID_bus_r;
```

```
    reg [152:0] ID_EXE_bus_r;
    reg [105:0] EXE_MEM_bus_r;
    reg [ 69:0] MEM_WB_bus_r;
```

修改后的运行结果，如图

在这两条指令的后面依次增加

nxor $3,$1,$2

Sgt $3,$1,$2

Sgtu $3,$1,$2

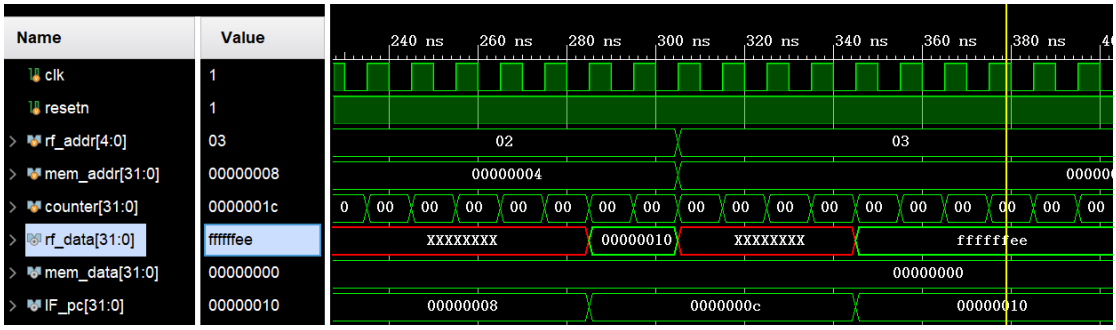
这三条指令

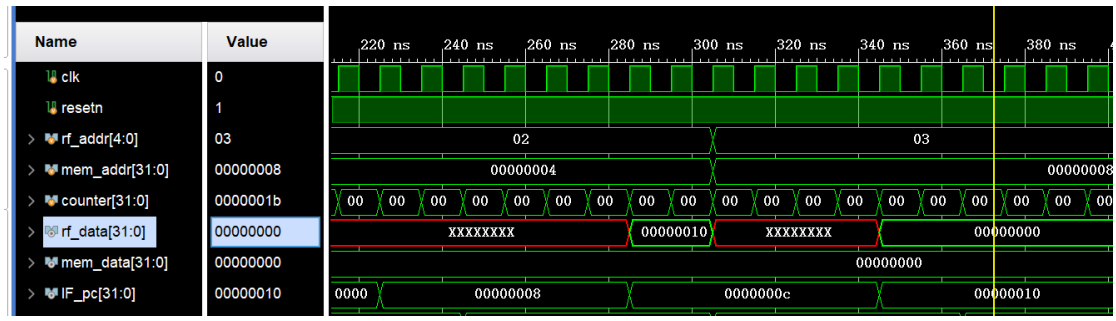
其中增加第一条指令后，$3 的值应该为 ffffffee

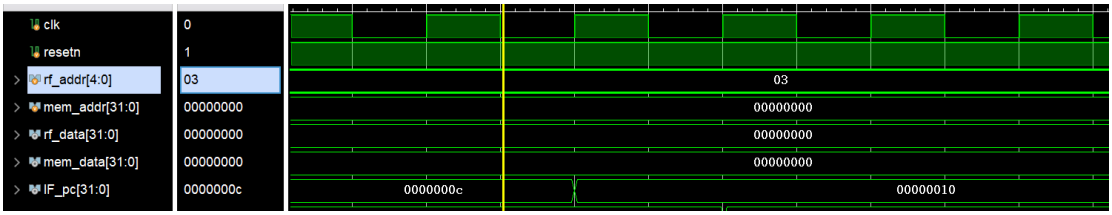
增加第二条指令后，$3 的值应该为 00000000

增加第三条指令后，$3 的值应该为 00000000，如下图仿真结果可以看出，验证正确

00H	addiu $1, $0,#1	[$1] = 0000_0001H	24010001	0010_0100_0000_0001_0000_0000_0000_0001
04H	sll $2, $1,#4	[$2] = 0000_0010H	00011100	0000_0000_0000_0001_0001_0001_0000_0000

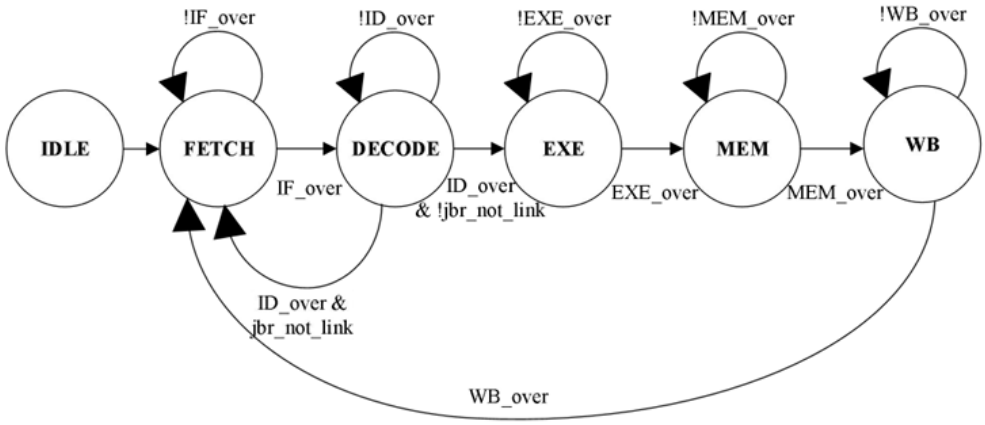






总结感想

多周期 CPU 设计在单周期 CPU 基础上，主要做两部分改进。第一部分是控制单元，增加控制电路使每一个时钟只有一个阶段的电路产生的结果有效，并锁存上一阶 段的结果用于后续阶段的运行；第二部分是数据通路，增加实现新增指令的电路。 第一部分的改进主要是增加状态机控制及增加各阶段之间的用于锁存的寄存器。 由于有 5 个阶段，状态机共有 6 个状态：空闲（IDLE）、取指（FETCH）、译码 （DECODE）、执行（EXE）、访存（MEM）、写回（WB），如下图：



空闲（IDLE）状态：CPU 在复位时，所有阶段电路都无效，CPU 等待复位结束开始 下一状态——取指。

取指（FETCH）状态：在状态机进入取指状态的同时，PC 更新为下一 PC 值。故取 指状态下，将 PC 值作为指令存储器的地址去取指令。由于同步指令存储器在下一时钟 周期返回指令，因此取指需要两个时钟周期的时间。当取指结束后锁存取指阶段产生 的结果——当前 PC 值和指令。

译码（DECODE）状态：类似于单周期 CPU 的译码阶段，主要完成指令译码、读寄 存器、判断跳转等。控制单元区分各条指令并产生用于译码、执行、访存、写回的控 制信号。当译码结束后锁存译码阶段产生的结果用于下一状态执行：

分别用于执行、访存、写回的控制信号、用于执行阶段的两个源操作数、用于访存阶段的写入内存数 据、用于写回阶段的写寄存器地址。

执行（EXE）状态：ALU 模块完成操作。当执行结束后锁存执行阶段产生的结果及 前级传递的结果：用于访存、写回的控制信号、ALU 结果、内存写入数据、寄存器写 地址。

访存（MEM）状态：完成对数据存储器的读或写，并选择出将要写回寄存器的值。当访存结束后锁存访存阶段产生的结果及前级传递的结果：用于写回的控制信号、写 回数据、写回地址。 写回（WB）状态：完成寄存器写入。 CPU 复位结束，状态机由 IDLE 进入取指状态，其后在每次上一级结束信号有效的 时进入下一状态，写回级结束后返回取指级。当然也有例外，当指令是跳转而非链接跳转指令时，在译码状态后直接返回取下一指令不需要经过执行等后续阶段。