Lab7 多周期CPU

2020201018 涂奕腾

1.控制信号设计

相较于10条指令cpu，增加了移位相关信号 `shift_sa`，`shift`； Mdu控制信号 `Mdu_op`，`Mdu_start`； 非对齐内存访问填充信号 `memsign` 和位数控制信号 `mem_bit`

```
typedef struct packed{
    logic regdst; //控制写入rd/rt
    logic jump;
    logic branch;
    logic memread;
    logic mem2reg;
    logic [3:0] ALU_op; //用于获取ALU操作或分支控制
    logic imm2reg;
    logic jumpreg;
    logic writera; //写31号寄存器(ra)
    logic memwrite;
    logic ALU_src;
    logic regwrite;
    logic immsign; //imm是否需要按照符号位填充
    logic shift;
    logic shift_sa; //移位的位数是否为sa字段
    mdu_operation_t Mdu_op;
    logic Mdu_start;
    logic [1:0] mem_bit; //内存读取位数
    logic memsign; //非对齐内存访问是否需要填充高位
    logic overflow; //(addi)加法是否溢出
}Controller;
```

2. 模块设计

2.1. ALU

`module ALU( pc, A, B, overflow, ctrl, C, zero );`
用于进行加、减、位运算，运算的具体类型由 `control` 决定， `overflow == 1` 时产生溢出会报错。对应的处理如下：

ctrl	option	ctrl	option
0000	A & B	0001	A | B
0011	A ^ B	0100	~(A | B)
0010	A + B (可能溢出)	0110	A - B (可能溢出)
0101	A < B	0111	$signed(A) < $signed(B)
1000	A << B	1001	A >> B
1011	($signed(A)) >>> B		

2.2. ALUControl

`module ALUControl( ALU_op, func, ctrl );`
根据 `ALU_op` 信号和R型指令的func段解析出ALU控制信号 `ctrl` ，下面只给出部分示例：

```
module ALUControl( ALU_op, func, ctrl );
    input wire[3:0] ALU_op;
    input wire[5:0] func;
    output wire[3:0] ctrl;
    reg [3:0] res;
    assign ctrl = res;

    always_comb begin
        if(ALU_op == 0) res = 4'b0010; //add in alu
        else if (ALU_op == 2'b01) res = 4'b0110; //sub in alu
        else if (ALU_op == 2'b10) begin
            case(func)
                6'b000000: res = 4'b1000; //SLL
                6'b000010: res = 4'b1001; //SRL
                //......
            endcase
        end
        else if (ALU_op & 4'b1000) begin
            case (ALU_op[2:0])
                3'b001: res = 4'b0000; //ANDI
                3'b010: res = 4'b0001; //ORI
                //......
            endcase
        end
    end
endmodule
```

2.3. BranchUnit

`module BranchUnit( in1, in2, ALU_op, out );`
根据 `in1` ， `in2` 两个输入值以及 `ALU_op` 信号判断是否走向分支。

```
module BranchUnit( in1, in2, ALU_op, out );
    input wire[31:0] in1;
    input wire[31:0] in2;
    input wire[3:0] ALU_op;
    output wire out;
    reg tmp;
    assign out = tmp;
    always_comb begin
        case(ALU_op)
            4'b0001: tmp = (in1 == in2);
            4'b0010: tmp = (in1 != in2);
            4'b0011: tmp = ($signed(in1) <= 0);
            4'b0100: tmp = ($signed(in1) > 0);
            4'b0101: tmp = ($signed(in1) >= 0);
            4'b0110: tmp = ($signed(in1) < 0);
        endcase
    end
endmodule
```

2.4. ControllerUnit

`module ControllerUnit( input Instruction instr, output Controller ctrl );`
用于解析指令，使用联合体以便于不同类型指令的独立操作，默认使用R型指令进行解析。下面给出部分示例：

```
//utils.sv
typedef struct packed {
    logic [5:0] op;
    logic [4:0] rs;
    logic [4:0] rt;
    logic [4:0] rd;
    logic [4:0] padding;
    logic [5:0] func;
}R_form;

typedef struct packed {
    logic [5:0] op;
    logic [4:0] rs;
    logic [4:0] rt;
    logic [15:0] imm;
}I_form;

typedef struct packed {
    logic [5:0] op;
    logic [25:0] addr;
}J_form;

typedef union packed {
    R_form R;
    I_form I;
    J_form J;
}Instruction;

//ControllerUnit.sv
module ControllerUnit( input Instruction instr, output Controller ctrl );
    always_comb begin
        ctrl = 0;
        if(instr != 0) begin
            case(instr.R.op)
                6'b001000: begin //ADDI
                    ctrl.overflow = 1;
                    ctrl.ALU_src = 1;
                    ctrl.regwrite = 1;
                    ctrl.immsign = 1;
                end
                //......
                default : begin
                    $display("illegal opcode: %b", instr.R.op);
                    $finish;
                end
            endcase
        end
    end
endmodule
```

2.5. DataMemory

```
module DataMemory( fd, reset, clock, pcValue, address, writeEnabled, writeInput, ctrl, readResult);
```

内存总大小为8kb，使用 `ctrl.mem_bit` 控制内存访问位数， `ctrl.memsign` 表示非对齐内存访问时是否填充。此外还进行了异常处理。

```
module DataMemory( fd, reset, clock, pcValue, address,
    writeEnabled, writeInput, ctrl, readResult);
    input integer fd;
    input wire reset, clock, writeEnabled;
    input wire [31:0] pcValue, address, writeInput;
    input Controller ctrl;
    output wire[31:0] readResult;
    reg [31:0] data[2047:0], res;
    integer i;
    assign readResult = res;

    always_comb begin
        if(ctrl.memread == 1) begin
            case(ctrl.mem_bit)
                2'b00: begin
                    if (address[1:0] != 2'b00) begin
                        $display("@%h: *%h SignalException AddressError",
                             pcValue, address);
                        $finish;
                    end
                    res = data[address[12:2]];
                end
                2'b01: res = {{24{ctrl.memsign &
                        data[address[12:2]][8*address[1:0]+7 +: 1]}},
                        data[address[12:2]][8*address[1:0] +: 8]};
                2'b10: begin
                    if (address[0] != 1'b0) begin
                        $display("@%h: *%h SignalException AddressError",
                            pcValue, address);
                        $finish;
                    end
                    res = {{16{ctrl.memsign &
                        data[address[12:2]][16*address[1]+15 +: 1]}},
                        data[address[12:2]][16*address[1] +: 16]};
                end
            endcase
        end
    end

    always_ff @(posedge clock ) begin
        if(reset == 1) begin
            for (i = 0; i< 2048; i = i + 1)
                data[i] <= 0;
        end
        else if (writeEnabled) begin
                case(ctrl.mem_bit)
                    2'b00: begin
                        if (address[1:0] != 2'b00) begin
                            $display("@%h: *%h SignalException AddressError",
                                pcValue, address);
                            $finish;
                        end
                        data[address[12:2]] = writeInput;
                    end
                    2'b01: data[address[12:2]][8*address[1:0] +: 8]
                        = writeInput[7:0];
                    2'b10: begin
                        if (address[0] != 1'b0) begin
                            $display("@%h: *%h SignalException AddressError",
                                pcValue, address);
                            $finish;
                        end
                        data[address[12:2]][16*address[1] +: 16]
                            = writeInput[15:0];
                    end
                endcase
                $display("@%h: *%h <= %h", pcValue,
                    {address[31:2], 2'b00}, data[address[12:2]]);
                $fdisplay(fd, "@%h: *%h <= %h", pcValue,
                    {address[31:2], 2'b00}, data[address[12:2]]);
            end
    end
endmodule
```

2.6. ForwardingUnit(数据旁路)

`module ForwardingUnit( IF_ID, ID_EX, EX_MEM, MEM_WB, next_ID_EX, rs_val_in_EX, rt_val_in_EX, rs_val_in_ID, rt_val_in_ID );`
设计了到EX和ID的旁路:

```
module ForwardingUnit( IF_ID, ID_EX, EX_MEM, MEM_WB, next_ID_EX, rs_val_in_EX,
    rt_val_in_EX, rs_val_in_ID, rt_val_in_ID );
    input PipelineReg IF_ID, ID_EX, EX_MEM, MEM_WB, next_ID_EX;
    output reg[1:0] rs_val_in_EX, rt_val_in_EX, rs_val_in_ID, rt_val_in_ID;
    always_comb begin
    //EX
        if(EX_MEM.ctrl.regwrite == 1 && EX_MEM.rd !=0 &&
            EX_MEM.rd == ID_EX.rs)  rs_val_in_EX = 2'b10;
        else if(MEM_WB.ctrl.regwrite == 1 && MEM_WB.rd != 0 &&
            MEM_WB.rd == ID_EX.rs) rs_val_in_EX = 2'b01;
        else rs_val_in_EX = 2'b00;
        if(EX_MEM.ctrl.regwrite == 1 && EX_MEM.rd !=0 &&
            EX_MEM.rd == ID_EX.rt)  rt_val_in_EX = 2'b10;
        else if(MEM_WB.ctrl.regwrite == 1 && MEM_WB.rd != 0 &&
            MEM_WB.rd == ID_EX.rt) rt_val_in_EX = 2'b01;
        else rt_val_in_EX = 2'b00;
    //ID
        if(ID_EX.ctrl.regwrite == 1 && ID_EX.rd !=0 &&
            ID_EX.rd == next_ID_EX.rs)  rs_val_in_ID = 2'b11;
        else if(EX_MEM.ctrl.regwrite == 1 && EX_MEM.rd != 0 &&
            EX_MEM.rd == next_ID_EX.rs) rs_val_in_ID = 2'b10;
        else if(MEM_WB.ctrl.regwrite == 1 && MEM_WB.rd != 0 &&
            MEM_WB.rd == next_ID_EX.rs) rs_val_in_ID = 2'b01;
        else rs_val_in_ID = 2'b00;
        if(ID_EX.ctrl.regwrite == 1 && ID_EX.rd !=0 &&
            ID_EX.rd == next_ID_EX.rt)  rt_val_in_ID = 2'b11;
        else if(EX_MEM.ctrl.regwrite == 1 && EX_MEM.rd != 0 &&
            EX_MEM.rd == next_ID_EX.rt) rt_val_in_ID = 2'b10;
        else if(MEM_WB.ctrl.regwrite == 1 && MEM_WB.rd != 0 &&
            MEM_WB.rd == next_ID_EX.rt) rt_val_in_ID = 2'b01;
        else rt_val_in_ID = 2'b00;
    end
endmodule
```

2.7. GeneralPurposeRegisters

`module GeneralPurposeRegisters( fd, reset, clock, pcValue, rs, rt, rd, writeEnabled, writeInput, regval1, regval2 );`
通用寄存器的读写。

```
module GeneralPurposeRegisters( fd, reset, clock, pcValue, rs, rt, rd,
    writeEnabled, writeInput, regval1, regval2 );
    input integer fd;
    input reset, clock, writeEnabled;
    input [4:0] rs,rt,rd;
    input [31:0] pcValue, writeInput;
    output [31:0] regval1, regval2;
    reg[31:0] regdata[31:0];
    assign regval1 = regdata[rs];
    assign regval2 = regdata[rt];
    integer i;

    always_ff @ (posedge clock) begin
        if (reset) begin
            for ( i = 0; i < 32; i = i + 1 )
                regdata[i] <= 0;
        end
        if (writeEnabled) begin
            if (rd != 0) regdata[rd] <= writeInput;
            $display("@%h: $%d <= %h", pcValue, rd, writeInput);
            $fdisplay(fd, "@%h: $%d <= %h", pcValue, rd, writeInput);
        end
    end
endmodule
```

2.8. HazardDetectionUnit(阻塞)

`module HazardDetectionUnit( Mdubusy, IF_ID, ID_EX, EX_MEM, MEM_WB, next_ID_EX, flush, Mduflush );`
分支、跳转与上一条指令以及加载指令与下一条指令出现寄存器重用时设置 `flush` 信号来阻塞一个周期，而加载指令需要对于下一条指令为分支和跳转时的寄存器重用额外阻塞一个周期。对于MDU的阻塞，MDU已经处于busy状态时若还需要使用MDU进行运算，则需阻塞一个周期：

```
module HazardDetectionUnit( Mdubusy, IF_ID, ID_EX, EX_MEM, MEM_WB,
    next_ID_EX, flush, Mduflush );
    input wire Mdubusy;
    input PipelineReg IF_ID, ID_EX, EX_MEM, MEM_WB, next_ID_EX;
    output reg flush, Mduflush;

    always_comb begin
        if(ID_EX.ctrl.memread == 1 && (ID_EX.rd == next_ID_EX.rs ||
            ID_EX.rd == next_ID_EX.rt)) flush = 1;
        else if (EX_MEM.ctrl.memread == 1 && (EX_MEM.rd == next_ID_EX.rs ||
            EX_MEM.rd == next_ID_EX.rt) && (next_ID_EX.ctrl.branch == 1 ||
            next_ID_EX.ctrl.jumpreg == 1)) flush = 1;
        else if(ID_EX.ctrl.regwrite == 1 && ID_EX.rd != 0 &&
            (ID_EX.rd == next_ID_EX.rs || ID_EX.rd == next_ID_EX.rt) &&
            (next_ID_EX.ctrl.branch == 1 || next_ID_EX.ctrl.jumpreg == 1))
            flush = 1;
        else flush = 0;

        if ((Mdubusy &  ID_EX.ctrl.Mdu_start) == 1 ) Mduflush = 1;
        else Mduflush = 0;
    end
endmodule
```

2.9. MultiplicationDivisionUnit

由下发文件提供

2.10. PipelineRegister

`module PipelineRegister( reset, clock, in, out );`
用于进行流水线间寄存器的更新，即用下一条指令的流水线寄存器信息更新当前指令的流水线寄存器信息。将 `reset` 信号设置为1时清空寄存器。

```
//utils.sv
typedef struct packed {
    Instruction instr;
    Controller ctrl;
    logic [3:0] ALU_sig;
    logic ALU_zero;
    logic [31:0] pc;
    logic [31:0] rs_val, rt_val, rd_val;
    logic [31:0] imm_val;
    logic [31:0] ALU_input;
    logic [4:0] rs, rt, rd;
}PipelineReg;

//PipelineRegister.sv
module PipelineRegister( reset, clock, in, out );
    input wire reset, clock;
    input PipelineReg in;
    output PipelineReg out;
    reg [$bits(PipelineReg)-1:0] tmp;
    assign out = tmp;

    always_ff @ (posedge clock) begin
        if (reset == 1) tmp <= 0;
        else tmp <= in;
    end
endmodule
```

2.11. ProgramCounter

`module ProgramCounter( reset, clock, jumpInput, pcValue );`
在reset为0时更新pc为输入值：

```
module ProgramCounter( reset, clock, jumpInput, pcValue );
    input wire reset, clock;
    input wire [31:0] jumpInput;
    output wire[31:0] pcValue;
    reg [31:0] pc;
    assign pcValue = pc;

    always_ff @( posedge clock ) begin
        if (reset) pc <= 'h00003000;
        else pc <= jumpInput;
    end
endmodule
```

3. 流水线设计思路概述

- IF
  使用PC进行取指，阻塞时pc保持不变
- ID
  使用CU对指令进行解析，使用GPR进行寄存器读写，根据信号内容计算ALU的操作数和option以及分支/跳转地址

- EX
  使用ALU/MDU进行计算
- MEM
  读写内存
- WB
  写回寄存器实际上放到了ID阶段，这里只需要判断是否遇到 `syscall` 即可

4.冲突处理

4.1. 数据旁路

由于在EX需要处理寄存器重用的ALU操作(即旁路数据)，在ID需要处理寄存器重用的跳转和分支，当前指令所需要的某个寄存器值依赖于上一条指令，但上一条指令还没有给出结果时，需要使用数据旁路。其中ID需要来自 ID_EX、EX_MEM、MEM_WB三个流水线寄存器的数据，而EX需要EX_MEM、MEM_WB两级流水线寄存器的数据。通过 `ForwardingUnit` (见2.6.)实现了到ID和EX的数据旁路，通过 `rs_val_in_ID` 、 `rt_val_in_ID` 、 `rs_val_in_EX` 、 `rt_val_in_EX` 信号确定传入ID/EX阶段rs/rt寄存器的值。以rs为例，ID和EX阶段的处理分别如下：

```
//ID-------
case (rs_val_in_ID)
    2'b00: next_ID_EX.rs_val = rsval; //来自GPR读取结果
    2'b01: next_ID_EX.rs_val = MEM_WB.rd_val; //来自上一周期的MEM_WB
    2'b10: next_ID_EX.rs_val = EX_MEM.rd_val; //来自上一周期的EX_MEM
    2'b11: next_ID_EX.rs_val = ID_EX.rd_val; //来自上一周期的ID_EX
endcase
//EX-------
case (rs_val_in_EX)
    2'b00: next_EX_MEM.rs_val = ID_EX.rs_val; //来自ID_EX的结果
    2'b01: next_EX_MEM.rs_val = MEM_WB.rd_val; //来自上一周期的MEM_WB
    2'b10: next_EX_MEM.rs_val = EX_MEM.rd_val; //来自上一周期的EX_MEM
endcase
```

4.2. 阻塞

通过HDU判断是否进行阻塞(见2.8.)：对于分支、跳转以及加载指令出现的寄存器重用需要阻塞一个周期，而加载需要对于接下来分支和跳转时的寄存器重用额外阻塞一个周期，阻塞时ID_EX寄存器需要清空。此外MDU的也会导致EX级阻塞，需要清空EX_MEM寄存器。因此ID_EX和EX_MEM为复位型寄存器控制系统。

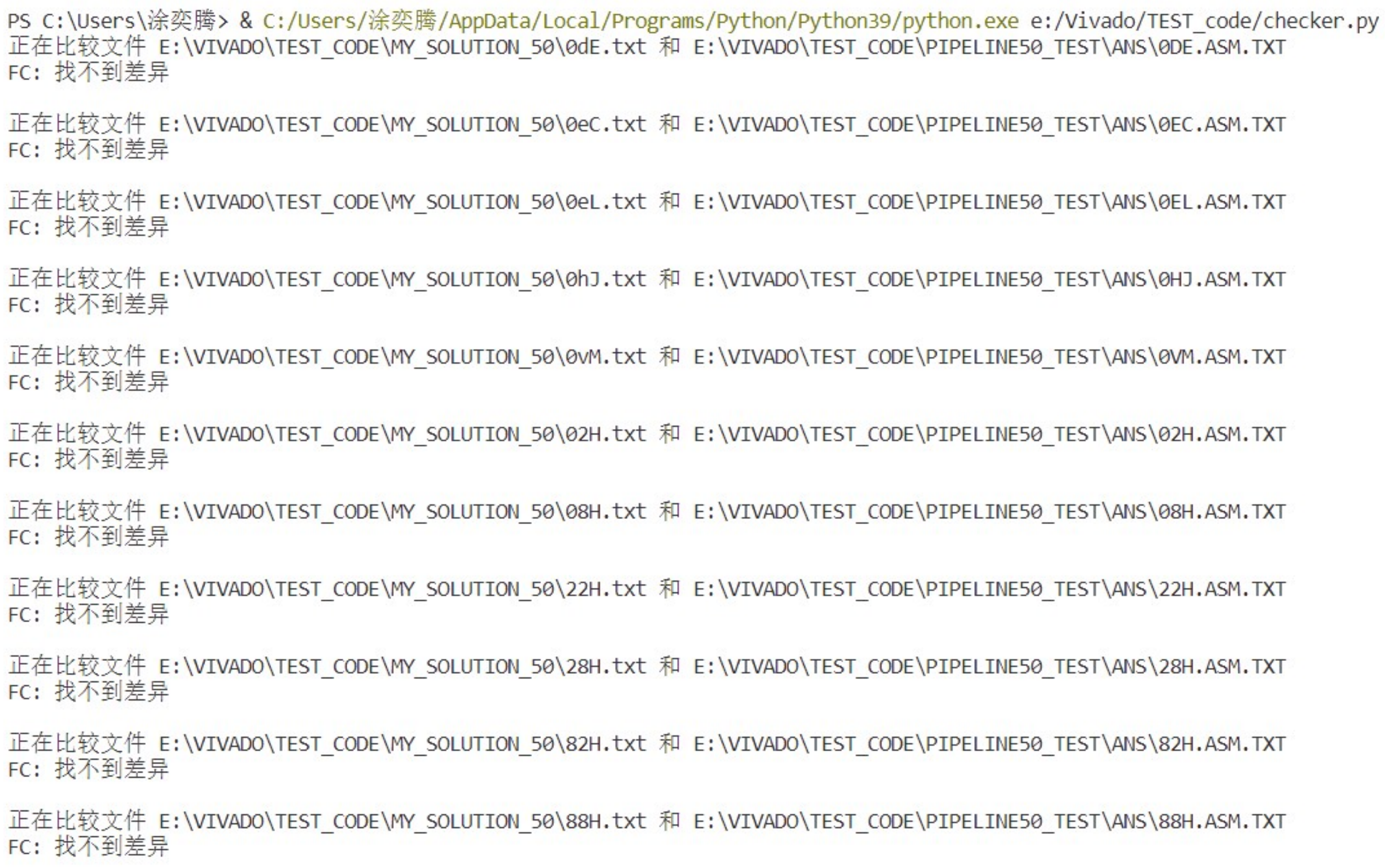
```
PipelineRegister IDEX(.reset(reset | flush), .clock(clock),
    .in(Mduflush ? ID_EX : next_ID_EX), .out(ID_EX) );
PipelineRegister EXMEM(.reset(reset | Mduflush), .clock(clock),
    .in(next_EX_MEM), .out(EX_MEM) );

//IF-------
if (flush | Mduflush) next_IF_ID.pc = IF_ID.pc;
else next_IF_ID.pc = pc;

//ID-------
if(flush | Mduflush) nextpc = next_IF_ID.pc + 4;
```

4.3. 分支冒险

存在分支延时槽，读入控制指令后下一条也是需要执行的指令，可以正常执行。在ID阶段从寄存器/旁路获取对应的数据，根据分支比较器结果确定下一条指令地址：

```
//ID-------
if(flush | Mduflush) nextpc = next_IF_ID.pc + 4;
else if(next_ID_EX.ctrl.jump) begin
    if(next_ID_EX.ctrl.jumpreg) nextpc = next_ID_EX.rs_val;
    else nextpc = {pc_4[31:28], IF_ID.instr.J.addr[25:0], 2'b00};
end
else if (next_ID_EX.ctrl.branch & gobranch) nextpc = IF_ID.pc +
    4 + {next_ID_EX.imm_val[29:0],2'b00};
else nextpc = next_IF_ID.pc + 4;
```

5. 实验结果



```
PS C:\Users\涂奕腾> & C:/Users/涂奕腾/AppData/Local/Programs/Python/Python39/python.exe e:/Vivado/TEST_code/checker.py
正在比较文件 E:\VIVADO\TEST_CODE\MY_SOLUTION_50\0dE.txt 和 E:\VIVADO\TEST_CODE\PIPELINE50_TEST\ANS\0DE.ASM.TXT
FC: 找不到差异

正在比较文件 E:\VIVADO\TEST_CODE\MY_SOLUTION_50\0eC.txt 和 E:\VIVADO\TEST_CODE\PIPELINE50_TEST\ANS\0EC.ASM.TXT
FC: 找不到差异

正在比较文件 E:\VIVADO\TEST_CODE\MY_SOLUTION_50\0eL.txt 和 E:\VIVADO\TEST_CODE\PIPELINE50_TEST\ANS\0EL.ASM.TXT
FC: 找不到差异

正在比较文件 E:\VIVADO\TEST_CODE\MY_SOLUTION_50\0hJ.txt 和 E:\VIVADO\TEST_CODE\PIPELINE50_TEST\ANS\0HJ.ASM.TXT
FC: 找不到差异

正在比较文件 E:\VIVADO\TEST_CODE\MY_SOLUTION_50\0vM.txt 和 E:\VIVADO\TEST_CODE\PIPELINE50_TEST\ANS\0VM.ASM.TXT
FC: 找不到差异

正在比较文件 E:\VIVADO\TEST_CODE\MY_SOLUTION_50\02H.txt 和 E:\VIVADO\TEST_CODE\PIPELINE50_TEST\ANS\02H.ASM.TXT
FC: 找不到差异

正在比较文件 E:\VIVADO\TEST_CODE\MY_SOLUTION_50\08H.txt 和 E:\VIVADO\TEST_CODE\PIPELINE50_TEST\ANS\08H.ASM.TXT
FC: 找不到差异

正在比较文件 E:\VIVADO\TEST_CODE\MY_SOLUTION_50\22H.txt 和 E:\VIVADO\TEST_CODE\PIPELINE50_TEST\ANS\22H.ASM.TXT
FC: 找不到差异

正在比较文件 E:\VIVADO\TEST_CODE\MY_SOLUTION_50\28H.txt 和 E:\VIVADO\TEST_CODE\PIPELINE50_TEST\ANS\28H.ASM.TXT
FC: 找不到差异

正在比较文件 E:\VIVADO\TEST_CODE\MY_SOLUTION_50\82H.txt 和 E:\VIVADO\TEST_CODE\PIPELINE50_TEST\ANS\82H.ASM.TXT
FC: 找不到差异

正在比较文件 E:\VIVADO\TEST_CODE\MY_SOLUTION_50\88H.txt 和 E:\VIVADO\TEST_CODE\PIPELINE50_TEST\ANS\88H.ASM.TXT
FC: 找不到差异
```

任务编号	时钟周期	任务编号	时钟周期
0dE	506	0eC	31534
0eL	4340	0hJ	48883
0vM	23227	02H	1040
08H	1200	22H	1095
28H	1163	82H	1271
88H	1092		