计算机系统结构课程实验

总结报告

题目：静态流水线设计与性能分析

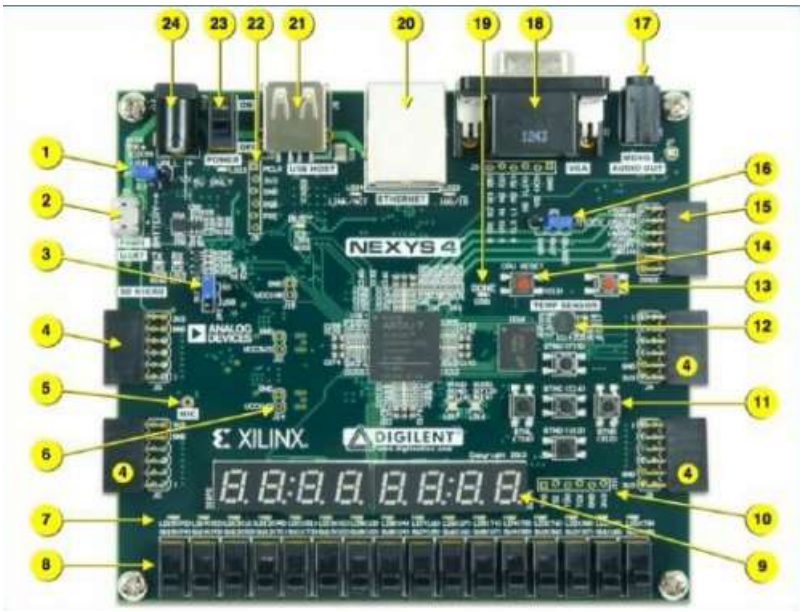
1652270 冯舜

老师：陆有军

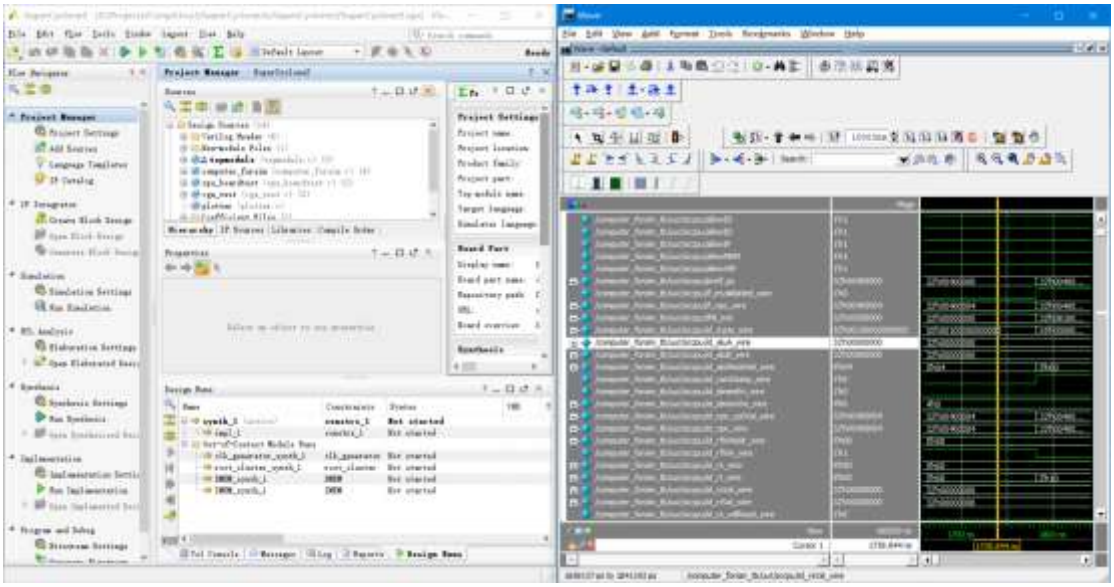
2018/11/18

一　实验环境部署与硬件配置

实验所用硬件为 Windows PC 以及 Xilinx Nexys 4 DDR 开发板。



软件包括开发板配套的 Vivado 2016（ISE 集成开发环境，综合、实现、比特流生成软件）以及业界领先仿真软件 ModelSim。



实验环境部署的过程为：打开 Vivado，新建一个空项目，硬件选择为 xc7a100tcsg324-1，不添加任何源代码，确认。

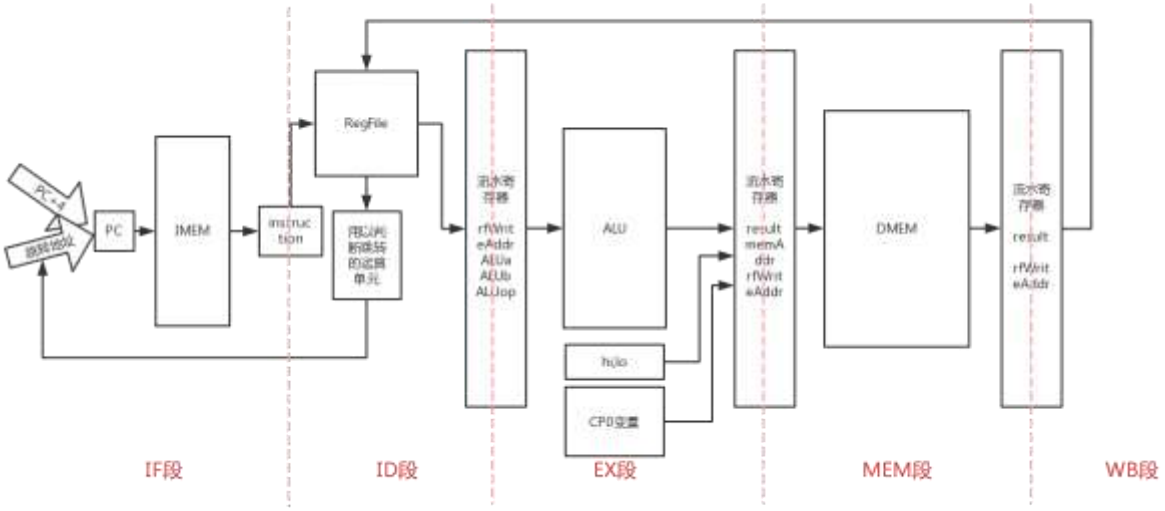
二　实验目的与目标

本次实验的目的是实现静态流水线模型，了解静态流水线的实现原理。

本实验的目标是通过编写 Verilog HDL 程序实现静态流水线 CPU，运行一个编写好的汇编程序，验证给出的数学模型。

三　实验的总体结构和部件

1.　静态流水线的总体结构



2.　部件及其解释说明

(1)　IF（取指令）段逻辑

```
always @(posedge clk) begin
    if (reset == `ENABLE) begin
        ifid_inst <= 0;
        preif_pc <= initInstAddr;
    end
    else
    if (cpuRunning) begin

        if (allowIF) begin
            if (if_invalidated_wire) begin
                ifid_inst <= NullInstruction;
            end else begin
                ifid_inst <= inst;
            end
            preif_pc <= if_npc_wire;
        end else if (bubbleIF) begin
            ifid_inst <= NullInstruction;
        end

    end
end
```

IF 段的逻辑实现是一段时序逻辑，其主要功能包括更新 PC，以及根据 PC 的值将指令内容取出，存入流水寄存器。

(2) ID（指令译码）段逻辑

```
always @ (*)
begin
    aluModeSel = 0;  // AND
    condJump = `DISABLE;
    rfWe = `ENABLE;
    dmemWe = 0;
    rfWAddr = 0;
    aluA = rsVal;
    aluB = rtVal;
    npc = pcPlus4;
    dmemEn = `DISABLE;
    rs_willRead = `ENABLE;
    rt_willRead = `ENABLE;
    validInstruction = `DISABLE;
    syscallFuncCode = 0;
    if(op == 6'b000000 && func == 6'b100000) begin
        validInstruction = `ENABLE;
        iAdd = 1;
        aluModeSel = ALU_SADD;
        rfWAddr = rd;
        //wd = alur
    end else iAdd = 0;
    if(op == 6'b000000 && func == 6'b100001) begin
        validInstruction = `ENABLE;
        iAddu = 1;
        aluModeSel = ALU_UADD;
        rfWAddr = rd;
        //wd = alur
    end else iAddu = 0;
    if(op == 6'b000000 && func == 6'b100010) begin
        validInstruction = `ENABLE;
        iSub = 1;                    以下其他指令省略
```

组合逻辑部分

```
always @(posedge clk) begin
    if (reset == `ENABLE) begin
        idex_itype <= 0;
        idex_rfWAddr <= 0;
        idex_imm <= 0;
        idex_aluModeSel <= 0;
        idex_rs <= 0;
        idex_rt <= 0;
        idex_rd <= 0;
        idex_aluA <= 0;
        idex_aluB <= 0;
        //idex_npc <= 32'hFFFFFFFF;
        idex_rfWe <= `DISABLE;
        idex_dmemEn <= `DISABLE;
        idex_dmemWe <= 0;
        idex_rsVal <= 0;
        idex_rtVal <= 0;
        idex_condJump <= `DISABLE;
    end
    else
    if (cpuRunning) begin

        if (allowID) begin
            idex_itype <= id_itype_wire;
            idex_rfWAddr <= id_rfWAddr_wire;
            idex_imm <= id_imm_wire;
            idex_aluModeSel <= id_aluModeSel_wire;
            idex_rs <= id_rs_wire;
            idex_rt <= id_rt_wire;
            idex_rd <= id_rd_wire;
            idex_aluA <= id_aluA_wire;
            idex_aluB <= id_aluB_wire;
            //idex_npc <= id_npc_wire
            idex_rfWe <= id_rfWe_wire;
            idex_dmemEn <= id_dmemEn_wire;
            idex_dmemWe <= id_dmemWe_wire;
            idex_rsVal <= id_rsVal_wire;
            idex_rtVal <= id_rtVal_wire;
            idex_condJump <= id_condJump_wire;
        end else if (bubbleID) begin
            idex_itype <= 0;
            idex_rfWAddr <= 0;
            idex_imm <= 0;
            idex_aluModeSel <= 0;
            idex_rs <= 0;
            idex_rt <= 0;
            idex_rd <= 0;
            idex_aluA <= 0;
            idex_aluB <= 0;
            //idex_npc <= id_npc_wire
            idex_rfWe <= `DISABLE;
            idex_dmemEn <= `DISABLE;
            idex_dmemWe <= 0;
            idex_rsVal <= 0;
            idex_rtVal <= 0;
            idex_condJump <= `DISABLE;
        end
```

时序逻辑部分

ID 段的实现方式是一段组合逻辑加一段时序逻辑。它的功能较丰富，包括对指令进行译码、取出指令涉及到需要读的寄存器值放置到流水寄存器、预先判断是否满足跳转条件以便控制 IF 段的 PC 等。

(3) EX（执行）段逻辑

```
always @* begin
    ex_extra_result_calculated = 0;
    ex_bytePos_calculated = 0;
    ex_result_calculated = 0;
    ex_dmemWe_calculated = ex_dmemWe_wire_orig;
    ex_dmemIn_calculated = ex_rtVal_wire;

    ex_rfWe_calculated = ex_rfWe_wire_orig;
    if (`E(IAdd) || `E(IAddu) || `E(ISub) || `E(ISubu) || `E(IAnd) || `E(IOr) || `E(IXor) |
    | `E(INor) || `E(ISlt) || `E(ISltu) || `E(ISll) || `E(ISrl) || `E(ISra) || `E(ISllv) ||
     `E(ISrlv) || `E(ISrav) || /*ijr*/ `E(IAddi) || `E(IAddiu) || `E(IAndi) || `E(IOri) ||
    `E(IXori) ||/*ilw,isw,ibeq,ibne*/ `E(ISlti) || `E(ISltiu) || `E(ILui) ||/*ij*/ `E(IJal)
     || `E(IDiv) || `E(IDivu) || `E(IMul) || `E(IMult) || `E(IMultu) ||/*ibgez*/ `E(IJalr)
    ||/*ilb,ilh,ilbu,ilhu,isb,ish,ibreak,isyscall,ieret*/ `E(IMfhi) || `E(IMflo) || `E
    (IMthi) || `E(IMtlo) || `E(IMfc0) || `E(IMtc0) || `E(IClz) /*iteq*/) begin
        if (`E(ILui))
            ex_result_calculated = {ex_imm_wire, 16'h0};
        else if (`E(IMfhi))
            ex_result_calculated = hi;
        else if (`E(IMflo))
            ex_result_calculated = lo;
        else if (`E(IMtc0))
            ex_result_calculated = cp0RData;
        else begin
            ex_result_calculated = aluR;
            ex_extra_result_calculated = aluRX;
        end

        if (`E(IAdd) || `E(ISub)) begin
            ex_rfWe_calculated = ex_rfWe_wire_orig & ~aluOverflow;
        end
    end

    if (`E(ILw) || `E(ISw) || `E(ILb) || `E(ILh) || `E(ILbu) || `E(ILhu) || `E(ISb) || `E
    (ISh)) begin
        ex_result_calculated = aluR;
    end

    if (`E(ISw)) begin
        ex_dmemWe_calculated = ex_dmemWe_wire_orig & 4'hf;
        ex_dmemIn_calculated = ex_rtVal_wire;
    end

    if (`E(ISb)) begin
        ex_bytePos_calculated = { 6'h0, aluR[1:0] ^ {2{BigEndianCPU}} };
        ex_dmemIn_calculated = ex_rtVal_wire << (ex_bytePos_calculated << 3);
        ex_dmemWe_calculated = ex_dmemWe_wire_orig & ((4'h1) << ex_bytePos_calculated);
    end

    if (`E(ISh)) begin
        ex_bytePos_calculated = {6'b000, aluR[1] ^ BigEndianCPU, 1'b0};
        ex_dmemIn_calculated = ex_rtVal_wire << (ex_bytePos_calculated[1] ? 16 : 0);
        ex_dmemWe_calculated = ex_dmemWe_wire_orig & ((4'b0011) << ex_bytePos_calculated);
    end
end
```

EX 段组合逻辑

```
always @(posedge clk) begin
    if (reset == `ENABLE) begin
        exmem_itype <= 0;
        exmem_rfWAddr <= 0;
        exmem_rfWe <= `DISABLE;
        exmem_condJump <= `DISABLE;
        exmem_result <= 0;
        exmem_bytePos <= 0;

        hi <= 0;
        lo <= 0;
    end
    else
    if (cpuRunning) begin

        if (allowEX) begin
            exmem_itype <= ex_itype_wire;
            exmem_rfWAddr <= ex_rfWAddr_wire;
            exmem_rfWe <= ex_rfWe_wire;
            exmem_condJump <= ex_condJump_wire;
            exmem_result <= ex_result_wire;
            exmem_bytePos <= ex_bytePos_wire;

            if (`E(IDiv) || `E(IDivu) || `E(IMult) || `E(IMultu)) begin
                hi <= ex_extra_result_wire;
                lo <= ex_result_wire;
            end else if (`E(IMthi)) begin
                hi <= ex_result_wire;
            end else if (`E(IMtlo)) begin
                lo <= ex_result_wire;
            end

            // 另外，cp0 的寄存器也会更新 (MTC0)

        end else if (bubbleEX) begin
            exmem_itype <= 0;
            exmem_rfWAddr <= 0;
            exmem_rfWe <= `DISABLE;
            exmem_condJump <= `DISABLE;
            exmem_result <= 0;
            exmem_bytePos <= 0;
        end

    end
end
```

EX 段时序逻辑

EX 段的实现也是一段组合逻辑加一段时序逻辑。对于计算类指令，EX 调用 ALU 获得结果；对于访存指令，EX 调用 ALU 计算出内存地址。结果统一存放于流水寄存器中。此外，EX 段还对 HI、LO 寄存器、CP0 协处理器的内部寄存器进行取值和更新。

(4) MEM（仿存）段逻辑

```
always @* begin
    mem_result_calculated = mem_result_wire_orig;
    if (`M(ILw)) begin
        mem_result_calculated = mem_dmemOut_wire;
    end
    if ( `M(ILb)) begin
        mem_result_calculated = extend8SOut;
    end
    if (`M(ILh)) begin
        mem_result_calculated = extend16S_2Out;
    end
    if ( `M(ILbu) ) begin
        mem_result_calculated = extend8UOut;
    end
    if ( `M(ILhu)) begin
        mem_result_calculated = extend16U_forLBUOut;
    end
end

always @(posedge clk) begin
    if (reset == `ENABLE) begin
        memwb_itype <= 0;
        memwb_rfWAddr <= 0;
        memwb_rfWe <= `DISABLE;
        memwb_result <= 0;
    end
    else
    if (cpuRunning) begin

        if (allowMEM) begin
            memwb_itype <= mem_itype_wire;
            memwb_rfWAddr <= mem_rfWAddr_wire;
            memwb_rfWe <= mem_rfWe_wire;
            memwb_result <= mem_result_wire;
        end else if (bubbleMEM) begin
            memwb_itype <= 0;
            memwb_rfWAddr <= 0;
            memwb_rfWe <= `DISABLE;
            memwb_result <= 0;
        end

    end
end
```

MEM 段组合逻辑与时序逻辑

MEM 段的实现也是组合逻辑加时序逻辑。主要功能是将 DMEM 数据存储器送出的数据存入流水寄存器中。

(5) WB（写回）段逻辑

```
// Writing Back

wire [54:0] wb_itype_wire = memwb_itype;
wire [4:0] wb_rfWAddr_wire = memwb_rfWAddr;
wire wb_rfWe_wire = memwb_rfWe;
wire [31:0] wb_result_wire = memwb_result;

assign rfWAddr = wb_rfWAddr_wire;
assign rfWData = wb_result_wire;
assign rfWe = wb_rfWe_wire;
```

WB 段逻辑

WB 段逻辑非常简单，是将寄存器堆的写有关值赋予前一段流出的值，是组合逻辑。当然，时钟上升沿到来时，在寄存器堆内部会做写入更新。

四　实验仿真过程

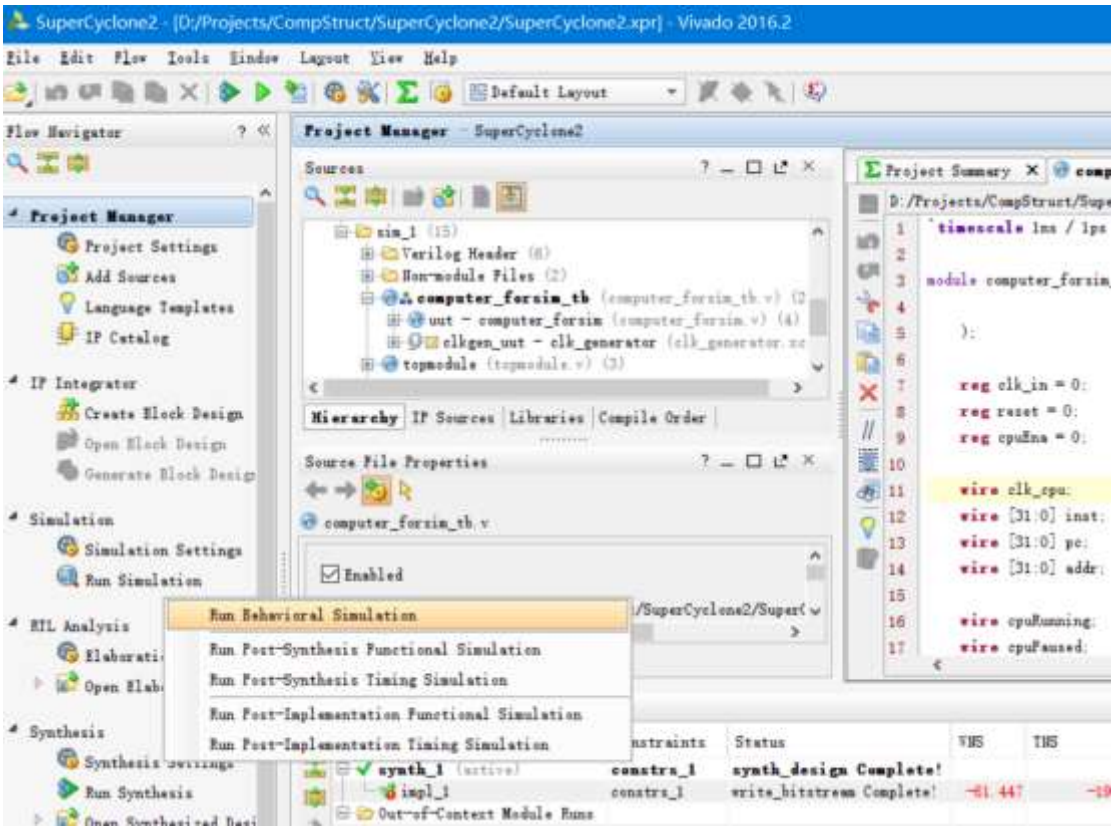
1. 静态流水线仿真过程

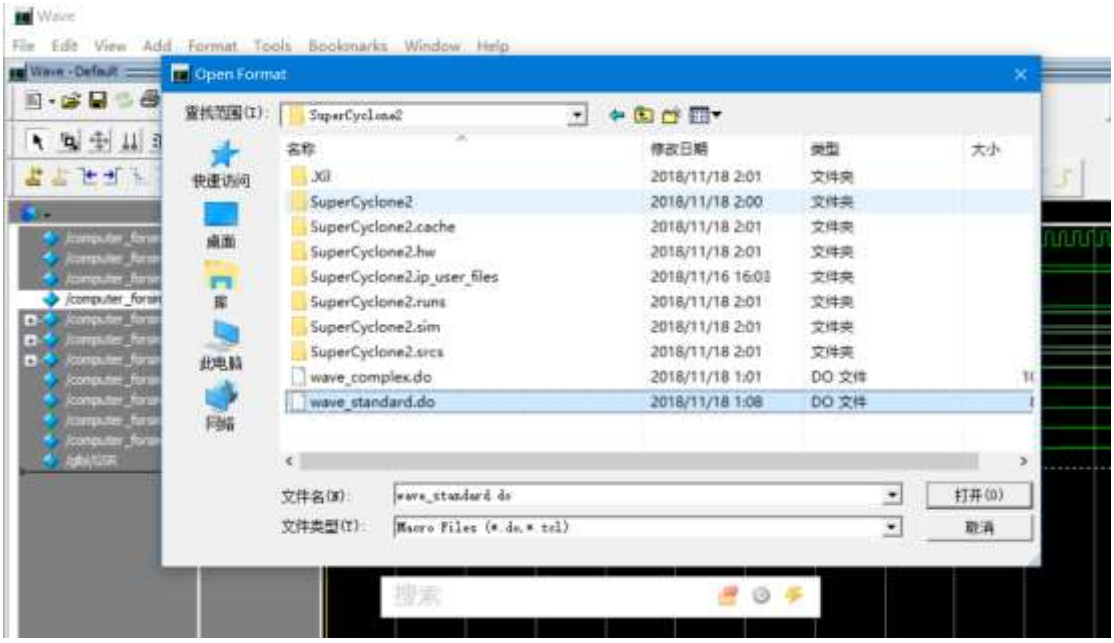
写仿真顶层文件：

```
`timescale 1ns / 1ps

module computer_forsim_tb(

    );

    reg clk_in = 0;
    reg reset = 0;
    reg cpuEna = 0;

    wire clk_cpu;
    wire [31:0] inst;
    wire [31:0] pc;
    wire [31:0] addr;

    wire cpuRunning;
    wire cpuPaused;

    always begin
        #5 clk_in = ~clk_in;
    end

    initial begin
        #3 reset = 1;
        #5 reset = 0;
        cpuEna = 1;
    end
```

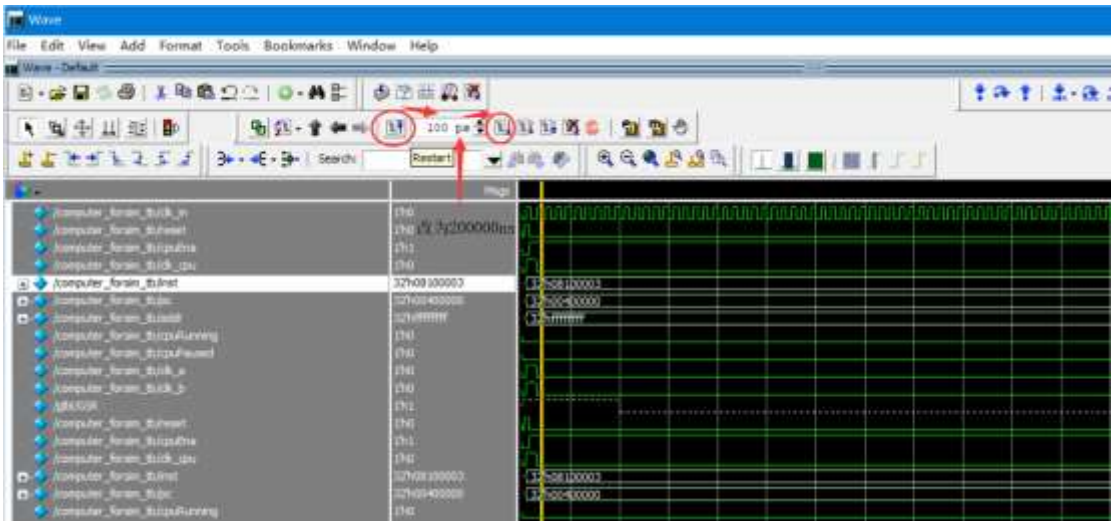
```
    computer_forsim uut(
        clk_in,
        reset,
        cpuEna,
        clk_cpu,
        inst,
        pc,
        addr,
        cpuRunning,
        cpuPaused
    );

    wire clk_a;
    wire clk_b;
    clk_generator clkgen_uut(
        .clk_100MHz(clk_in),
        .clk_vga(clk_a),
        .clk_cpu(clk_b)
    );

endmodule
```

打开 Vivado 工程，使用 Vivado 的“行为级仿真”功能，调用 ModelSim 进行仿真。

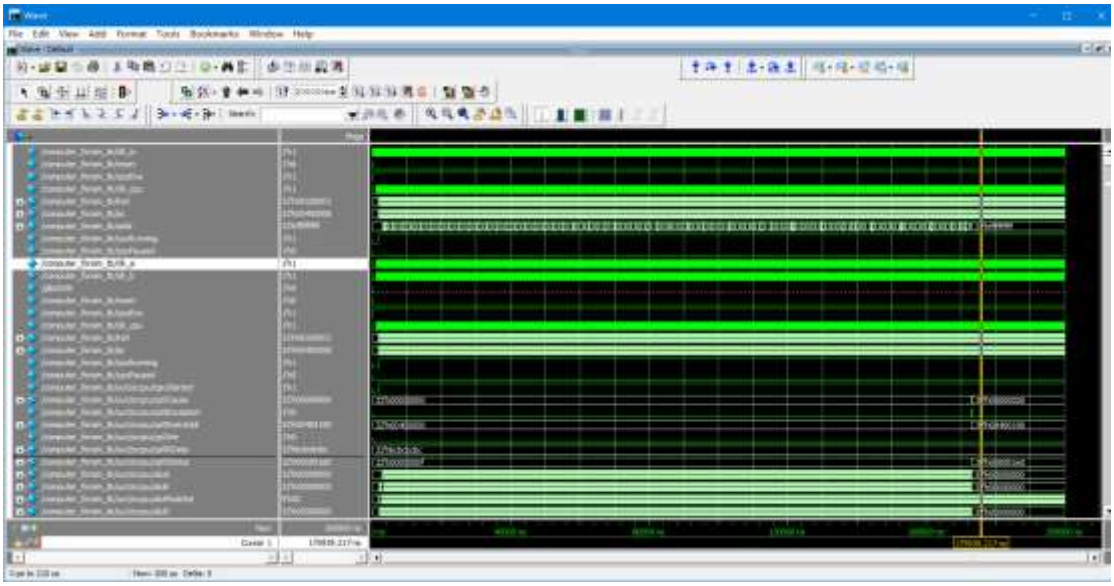




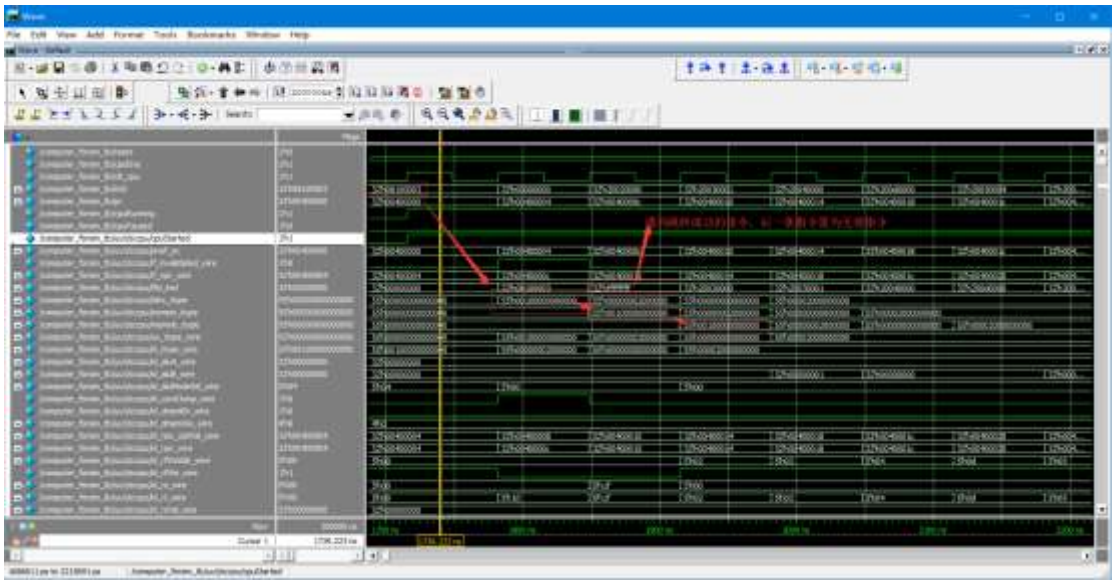
在波形表中，加载工程文件夹根目录下的 wave_standard.do。



点击“Restart”，修改仿真时间为 200000ns，点击“Run”，可以看到从程序开始运行到停止运行整个过程的仿真波形。



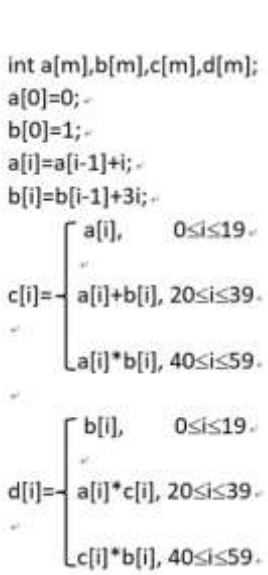
2. 波形图和寄存器波形的意义



- clk_cpu：CPU 的时钟
- cpuRunning：CPU 是否在运行
- pc：当前流入 CPU 的指令地址（PC 寄存器）
- inst：当前流入 CPU 的指令字
- ifid_inst：当前待译码的指令字
- *_itype：对应两个段之间的解析后的指令字（55 比特长的独热码，每个比特对应一种指令）。ID 之后每个段都有一个波形，总共四个波形，与 ifid_inst 结合可以看到指令流过 CPU 的过程。
- id_*：ID 段有关的信号内容
- idex_*：ID 段和 EX 段之间的流水寄存器
- ex_*：EX 段有关的信号内容
- exmem_*：EX 段与 MEM 段之间的流水寄存器
- mem_*：MEM 段有关的信号内容
- memwb_*：MEM 段与 WB 段之间的流水寄存器

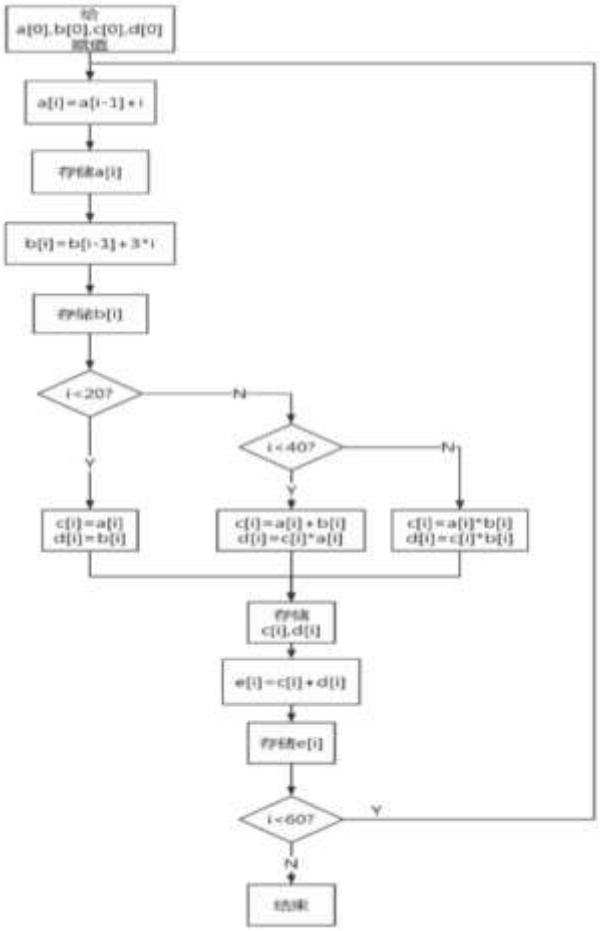
- wb_*：WB 段有关的信号内容
- blk_mem_gen_v8_3_3_inst/memory：DMEM 的内容

五　实验验算数学模型及算法程序




```
int a[m],b[m],c[m],d[m];
    a[i] = a[i-1]+i;
    b[i] = b[i-1]+3*i;
    m = 60;
    a[0] = 0;
    b[0] = 1;
    if(0<=i<=19) {
        c[i] = a[i];
        d[i] = b[i];
    }
    if(20<=i<=39) {
        c[i] = a[i] + b[i];
        d[i] = c[i] * a[i];
    }
    if(40<=i<=59) {
        c[i] = a[i] * b[i];
        d[i] = c[i] * b[i];
    }
```

给出的数学模型

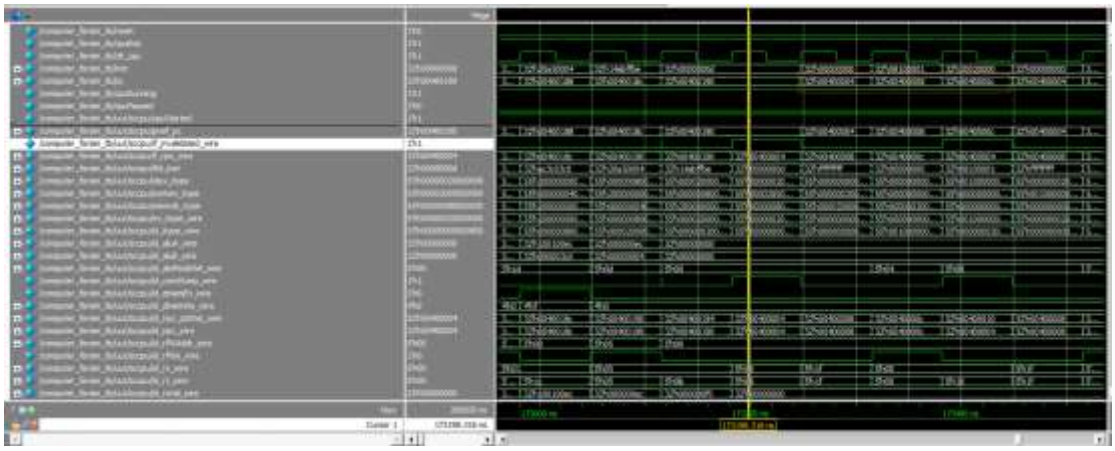


数学模型对应的算法流程图

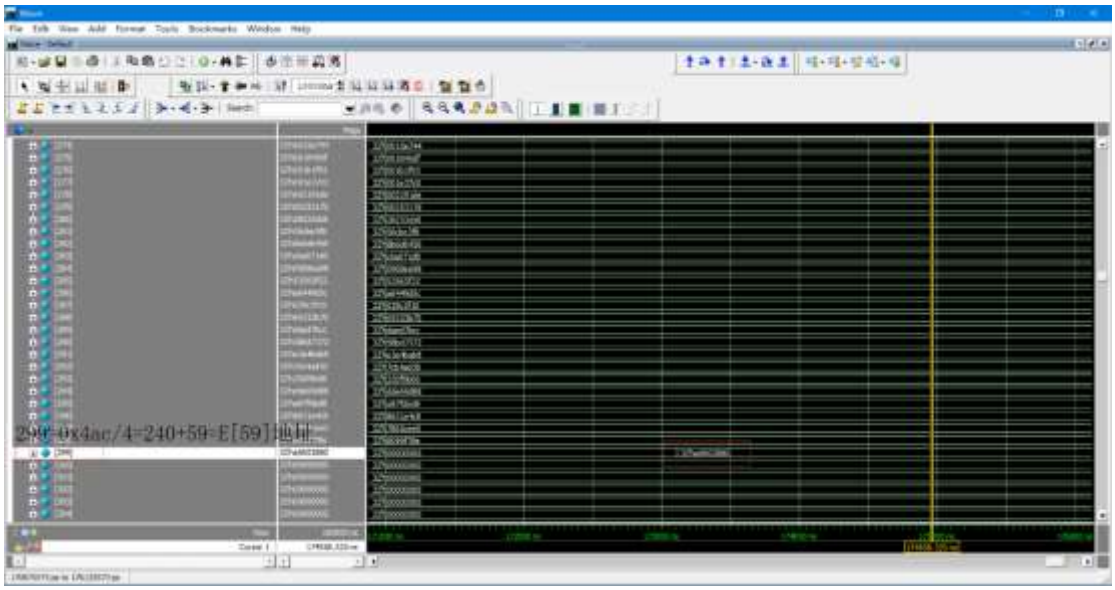
```
.data
A:.space 240
B:.space 240
C:.space 240
D:.space 240
E:.space 240

.text
j main
exc:
nop
j exc

main:
addi $2,$0,0 #a[i]
addi $3,$0,1 #b[i]
addi $4,$0,0 #c[i]
addi $13,$0,0 #d[i]
addi $5,$0,4 #counter
addi $6,$0,0 #a[i-1]
addi $7,$0,1 #b[i-1]
addi $10,$0,0 #flag for i<20 || i<40
addi $11,$0,240 #sum counts
addi $14,$0,3
addi $30,$0,0

# 把 0 1 0 0 ($2,...,$13) 分别存入 A B C D
lui $27,0x0000
addu $27,$27,$0
sw $2,A($27)
lui $27,0x0000
addu $27,$27,$0
sw $3,B($27)
lui $27,0x0000
addu $27,$27,$0
sw $2,C($27)
lui $27,0x0000
addu $27,$27,$0
sw $3,D($27)

# 循环
loop:
## $5(4) 除以 4 (=i) 存入 $12
srl $12,$5,2
```

数学模型对应的代码

六　实验验算程序仿真过程



如图，在约 173200ns，PC 陷入了 0x0040004-0x00400008-0x0040000c 的循环（0x0040000c 对应无效化的延迟槽），证明程序终止。在第 299 个 DMEM 字（299=240+59）查看 E[59]的内容：



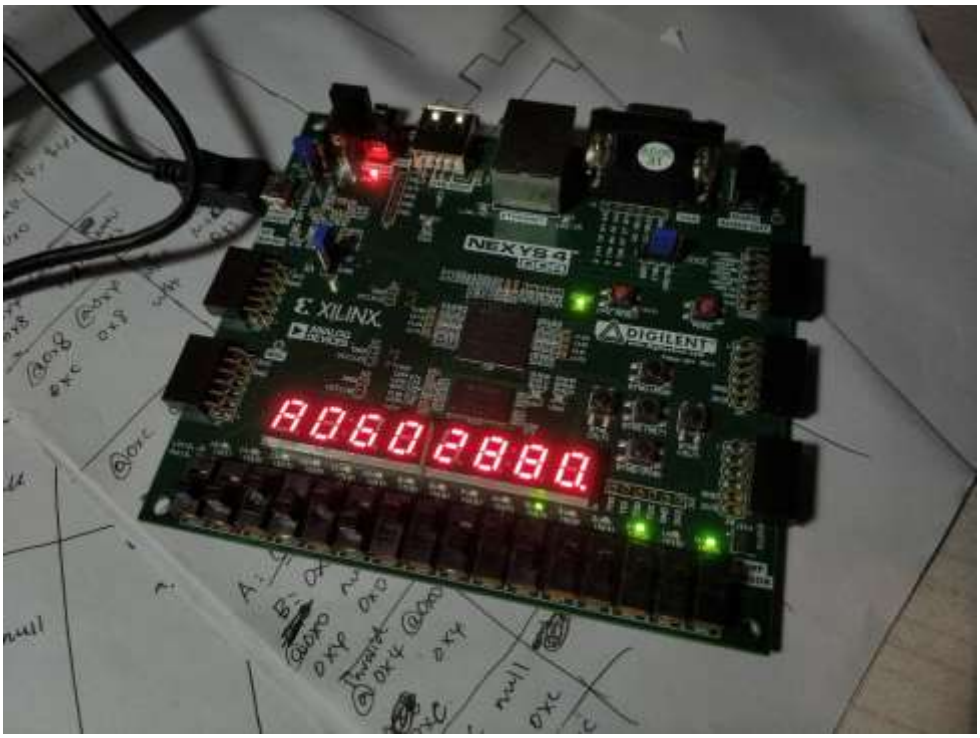
是 A0602880，运行结果正确，数学模型得到验证。

七　实验验算程序下板测试过程与实现

编写可以输出 DMEM 内容的顶层模块 topmodule.v（见工程中文件），将工程综合、实现、生成比特流后下载到开发板进行运行。可以直接下载预编译好的 SuperCyclone.bit 文件。

将十五个开关从左到右依此拨为“关关开关开关开开，关开关关关关开”（左 8 位 00101011 为 E[59]所在字的序号 0x4ac/4 的低 8 位，右侧的开关表示在调试输出模式下输出内存序号为 0x1XXXXXXXX（X 表示左侧开关对应的低八位）的字的值），按下

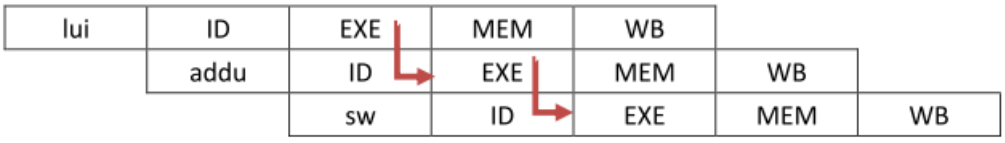
红色的 CPU_RESET 按钮重置 CPU，再按下“中”键。程序应该立刻跑完终止，七段数码管显示 E[59]的值 A0602880。验证了数学模型，CPU 制作正确。



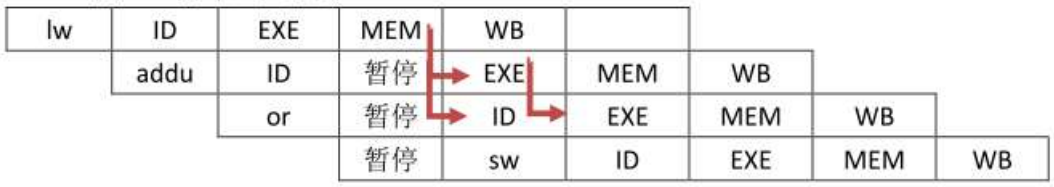
八　流水线的性能指标定性分析

1.　数据相关的情况

当距离较近的指令（相邻、隔一条指令、隔两条指令）存在对同一寄存器先写后读的情况时，称为数据相关。数据相关采用相关作用路径的方式解决，即能够在 EX 段或 MEM 段确定的该寄存器写入的结果，通过一条旁路导入需要读该寄存器的地方，用多路选择器最终选定该寄存器的读出值。



对于大多数情况，可将 EX 的结果导入 ID，不会产生气泡。但是，对于前一条指令为从存储器加载到寄存器的指令、后一条指令为需要读到这个寄存器的指令的情况，由于前一条指令必须到 MEM 才能产生结果，所以必须插入一条指令的“气泡”（从 ID 处产生一条空指令，ID 及之前的流水段不前进，后面的前进），再用相关作用路径。



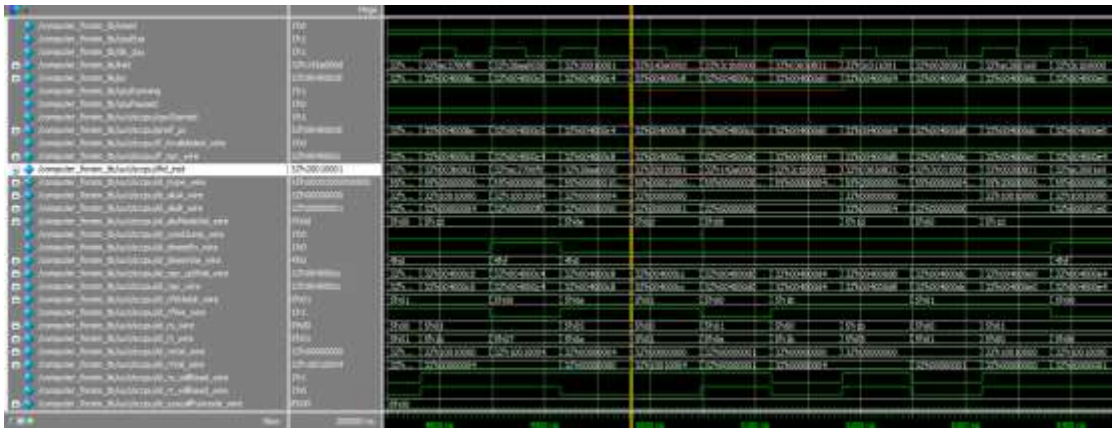

```
assign id_rsVal_wire = // 相关作用路径
    (ex_rfWAddr_wire == id_rs_wire && id_rs_wire != 0 && ex_rfWe_wire) ?
        (ex_itype_wire[ISw] | ex_itype_wire[ILbu] | ex_itype_wire[ILhu]
        | ex_itype_wire[ILb] | ex_itype_wire[ILh]) ? // 此时无法拿到内存
        中的数据
            32'h8A8A8A8A
        :
            ex_result_wire
    : (mem_rfWAddr_wire == id_rs_wire && id_rs_wire != 0 &&
    mem_rfWe_wire) ?
        mem_result_wire
    : (wb_rfWAddr_wire == id_rs_wire && id_rs_wire != 0 && wb_rfWe_wire)
     ?
        wb_result_wire
    : rfRData1;
assign id_rtVal_wire = // 相关作用路径
    (ex_rfWAddr_wire == id_rt_wire && id_rt_wire != 0 && ex_rfWe_wire) ?
        (ex_itype_wire[ISw] | ex_itype_wire[ILbu] | ex_itype_wire[ILhu]
        | ex_itype_wire[ILb] | ex_itype_wire[ILh]) ? // 此时无法拿到内存
        中的数据
            32'h8A8A8A8A
        :
            ex_result_wire
    : (mem_rfWAddr_wire == id_rt_wire && id_rt_wire != 0 &&
    mem_rfWe_wire) ?
        mem_result_wire
    : (wb_rfWAddr_wire == id_rt_wire && id_rt_wire != 0 && wb_rfWe_wire)
     ?
        wb_result_wire
    : rfRData2;
```

相关作用路径有关的 Verilog HDL 代码

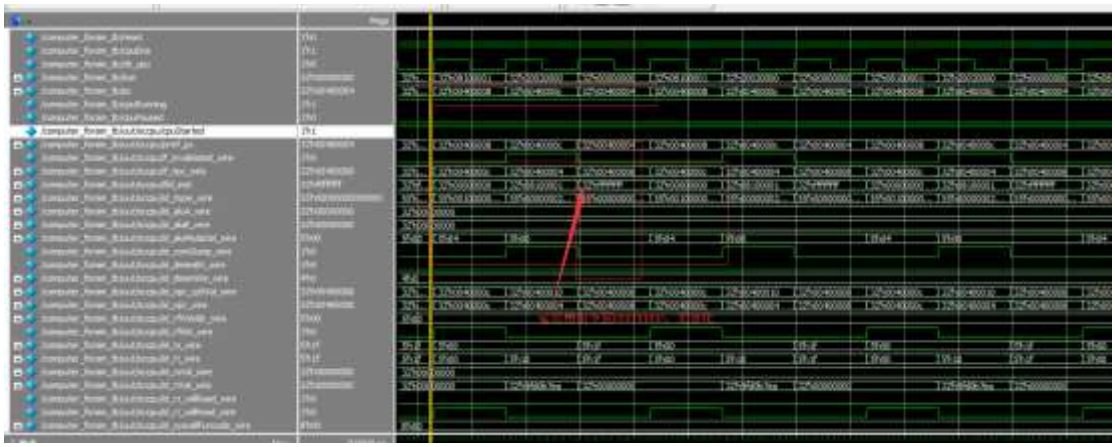
2. 控制相关的情况

对于分支或跳转指令，先默认转移失败。当 ID 段检测到转移成功时，应立即将前一条已经取指的指令无效化（此后，它作为延迟槽经过流水线）。本 CPU 的实现方式是，将已经取指的指令字改为无效指令 0xFFFFFFFF，并在之后的功能段加以识别，不执行这条指令。

转移失败的情况，流水照常运行：



转移成功的情况，延迟槽无效化，之后跳转：



3. 吞吐率

完成任务共执行了 2532 条指令，其中包括延迟槽 158 条，没有气泡（暂停）。

算上启动时间，吞吐率为：

$$\frac{2532-158}{2532+4}=0.93612\ 指令/时钟周期$$

4. 加速比

与所有指令用四倍时间单周期运行相比，加速比为

$$\frac{(2532-158)\times 4}{2532+4}=3.74448$$

5. 效率

认为指令平均使用四个功能段，效率为

$$\frac{(2532-158)\times 4}{(2532+4)\times 5}\times 100\%=74.89\%$$