计算机系统结构实验Lab06:
类MIPS多周期流水处理器实现

常烁晨 521021910369

2023.4.15

摘要

lab06 的实验目标是完成完整的多周期类MIPS处理器，本实验比 lab05 更加综合，所设计并完成的 CPU 可以流水化执行指令，因此所需要设计的部分更加复杂，数据通路、控制信号通路更加完整。

本实验完成了简单的类 MIPS 多周期流水化处理器，这一处理器支持 MIPS 的全部 31 条指令。同时该处理器设计时便支持 Stall、Forwarding、predict-not-taken 功能，使得处理器具有较好的性能。

由于本实验的数据与控制信号复杂繁多，且实验指导书上没有给出命名示例，因此我采用了自己的命名规则来完成数据和控制通路的信号传递与运算。

目录

摘要 1

1、实验目的 3

2、源代码实现 4

2.1 Top 4

2.2 Ctr – ALUCtr – ALU 相关拓展 16

3、仿真验证 21

3.1 multi_cycle_cpu_tb 21

3.2 仿真验证（一） 22

3.3 仿真验证（二） 24

4、致谢 26

1、实验目的

（1）在 lab05 的基础上，设计 CPU 的五段流水线，使其变成多周期可流水的处理器。

（2）在（1）的基础上设计支持 stall 功能的流水线，这要求检测流水线中的数据竞争、冒险，引入停顿功能，使得出现竞争的指令运行中断一个周期。

（3）在（2）的基础上增加 Forwarding 前向通路，减少数据竞争导致流水线的停顿延时。

（4）在（3）的基础上，增加 predict-not-taken 机制，减少控制竞争导致流水线的停顿延时。

（5）【选做】将 lab05 中的 16 条 MIPS 指令扩展至 31 条。

2、源代码实现

2.1 Top

在本实验中，由于指令流水化，因此取消了 PC 模块的设计，取指功能直接在 Top 模块中实现。为了实现五段流水线的 CPU 功能，Top 文件将单周期分割为 IF、ID、EX、MA、WB，五个部分，五部分的每两部分之间插入一部分寄存器，用于数据和控制信号的传递。

首先是 Top 文件的顶层模块初始化。此处与 lab05 相同，引入 clk 和 reset 两个信号。

```
`timescale 1ns / 1ps
module Top(
    input clk,
    input reset
    );
```

接下来是 IF 段的信号设计。在取指阶段，定义寄存器 Pc 和 信号 Inst，分别存储指令地址和 32 位二进制指令。通过指令存储器模块，将 Pc 所对应的指令加载到 Inst 中。

```
IF----------------------------------------------------------------//
    reg [31 : 0] Pc;
    wire [31 : 0] Inst;

    InstMemory inst_memory(.address(Pc), .inst(Inst));
IF----------------------------------------------------------------//
```

在 IF 和 ID 之间插入寄存器，用于分割周期，形成流水线结构。此部分寄存器需要保存 IF 段的信号。

```
//          IF to ID
reg [31 : 0] Pc_IfId;
reg [31 : 0] Inst_IfId;
```

下面是 ID 段的信号与数据通路设计。ID 是译码段，根据上一阶段所得到的指令，中央控制器 Ctr 要依据指令各个位置的高低电平，产生不同的控制信号，传递给后面的部件（包括 Ctr、Register、SignExt 和 MuxReg，这几个功能部件与 lab05 中同名的部件功能相同）。

```
ID------------------------------------------------------------------//
    wire [12 : 0] InstCtr;              wire [3 : 0] AluOp;
    wire [31 : 0] ReadData1;        wire [31 : 0] ReadData2;
    wire [4 : 0] WriteReg;              wire [31 : 0] WriteData;
    wire RegWrite;              wire [31 : 0] ExtRes;       wire [4 : 0] RegDst;

    Ctr main_controller(.opCode(Inst_IfId[31 : 26]), .funct(Inst_IfId[5 :
                                0]), .nop(Inst_IfId == 0),
                        .jump(InstCtr[12]), .jrSign(InstCtr[11]), .extSign(InstCtr[10]),
                        .regDst(InstCtr[9]), .jalSign(InstCtr[8]), .aluOp(AluOp),
                        .aluSrc(InstCtr[7]), .luiSign(InstCtr[6]), .beqSign(InstCtr[5]),
                        .bneSign(InstCtr[4]), .memWrite(InstCtr[3]), .memRead(InstCtr[2]),
                        .memToReg(InstCtr[1]), .regWrite(InstCtr[0]));
```

```
    Registers registers(.readReg1(Inst_IfId[25 : 21]), .readReg2(Inst_IfId[20 : 16]),
                        .writeReg(WriteReg), .writeData(WriteData),
                        .regWrite(RegWrite), .clk(clk), .reset(reset),
                        .jalSign(InstCtr[8]), .jalData(Pc_IfId + 4),
                        .readData1(ReadData1), .readData2(ReadData2));


    SignExt signext(.extSign(InstCtr[10]), .inst(Inst_IfId[15 : 0]), .data(ExtRes));

    MuxReg rt_rd_selector(
                        .select(InstCtr[9]),
                        .input1(Inst_IfId[15 : 11]),
                        .input2(Inst_IfId[20 : 16]),
                        .out(RegDst)
                        );
 ID----------------------------------------------------------------------//
```

ID阶段的数据通路比较复杂。首先根据 IF-ID阶段的寄存器，将上一个周期所获的二进制指令（Inst_IfId）加载到相关的模块中，比如 Ctr 产生的相关控制信号保存在 InstCtr 中，指令的高六位保存在 Ctr 模块的 opCode 中，低六位保存在 Ctr 模块的 funct 中，由此 Ctr 可以产生各种控制信号，这部分控制信号与 lab05 中的控制信号功能一致。

Ctr 模块产生控制信号后，对应的信号控制 Registers、SignExt 和 MuxReg 中，相关模块接收控制信号，产生的输出信号与指令的其他部分一起，作为寄存器文件的输入信号，确定寄存器文件将要输出的数据情况。这些数据将传递到下一周期，由 ALU 进行运算。

从 ID 到 EX 段所需要传递的寄存器内容如下。

```
//            ID to EX
reg [3 : 0] AluOp_IdEx;
reg [7 : 0] InstCtr_IdEx;
reg [31 : 0] ExtRes_IdEx;
reg [4 : 0] InstRs_IdEx;
reg [4 : 0] InstRt_IdEx;
reg [31 : 0] ReadData1_IdEx;
reg [31 : 0] ReadData2_IdEx;
reg [5 : 0] Funct_IdEx;
reg [4 : 0] Shamt_IdEx;
reg [4 : 0] RegDst_IdEx;
reg [31 : 0] Pc_IdEx;
```

首先将 Ctr 产生的四位 AluOp 保存到寄存器中，同时要保存 Ctr 产生的其他控制信号、符号扩展模块产生的 31 位立即数、寄存器的输出结果、其他 ALUCtr 所需要的控制信号（ funct、shamt ）、以及 Pc 的值。

下面是 EX 段流水线的设计。

```
EX-------------------------------------------------------------------//
    wire [3 : 0] AluCtrOut;
    wire ShamtSignal;
    ALUCtr alu_controller(
                        .aluOp(AluOp_IdEx),
                        .funct(Funct_IdEx),
                        .aluCtrOut(AluCtrOut),
                        .shamtSign(ShamtSignal)
                        );
```

EX 段产生的主要信号就是由 ALUCtr 产生的控制 ALU 运算功能的四位 AluCtrOut 信号，以及 shamt 信号。

在 EX 段，通过前向通路的设计，可以减少数据冒险。前向通路只需要把前面一段流水线访存的内容提前传递给下一段流水线的 ALU ，而不是在前一段流水线中写回 Registers 后，下一条指令等待三个周期后，再从 ID 段读取寄存器的值。添加了前向通路后，load-use 竞争就不需要流水线的停顿。

```
//--------- forward --------------//
wire [31 : 0] InputA_Forward;
wire [31 : 0] InputB_Forward;
//----------- forward ------------//
```

此处需要添加 ALU 的两个输入数据信号，而前向通路的数据起点在进行内存访问时开始。下面要进行多路选择器，选择 ALU 的输入数据，最终实例化 ALU 模块。

```
wire [31 : 0] InputA;
wire [31 : 0] InputA_Temp;
wire [31 : 0] InputB;
Mux rt_ext_selector(
                .select(InstCtr_IdEx[7]),
                .input1(ExtRes_IdEx),
                .input2(InputB_Forward),
                .out(InputB)
                );
Mux rs_shamt_selector(.select(ShamtSignal),
                  .input1({{27{0}}, Shamt_IdEx}),
                  .input2(InputA_Forward),
                  .out(InputA_Temp));
```

```
    Mux lui_selector(
                .select(InstCtr_IdEx[6]),
                .input1(32'h00000010),
                .input2(InputA_Temp),
                .out(InputA)
                );

    wire zero;
    wire Overflow;
    wire [31 : 0] AluRes;
    ALU alu(
          .inputA(InputA),
          .inputB(InputB),
          .aluCtrOut(AluCtrOut),
          .zero(zero),
          .overflow(Overflow),
          .aluRes(AluRes)
          );

    wire [31 : 0] BranchDst = Pc_IdEx + 4 + (ExtRes_IdEx << 2);
 EX-------------------------------------------------------------------//
```

首先根据前向通路 InputB_Forward 和拓展后的立即数，通过多路选择器选择一路，作为 ALU 的一个输入。之后通过两个多路选择器，在 shamt位运算数据、前向通路 InputA_Forward、立即数之间选择一个，作为 ALU 的另一个输入。

确定了 ALU 的输入信号、控制信息后，将 ALU 模块实例化，并设置零位和溢出位。最后加入 Branch 跳转地址的信号，用于接下来更新流水线 Pc。

从 EX 到 MA 段的寄存器设置如下。

```
//          EX to MA
reg [3 : 0] InstCtr_ExMa;
reg [31 : 0] AluRes_ExMa;
reg [31 : 0] ReadData2_ExMa;
reg [4 : 0] RegDst_ExMa;
```

需要向后传递的包括 Ctr 产生的剩余控制信号、上一阶段 ALU 的运算输出结果、寄存器地址以及读内存数据。

MA 段需要进行内存的读取或者写入。主要需要对数据存储器模块进行实例化，利用 EX 和 MA 之间的段寄存器来传递数据和控制信号。

```
MA-----------------------------------------------------------------//
    wire [31 : 0] MemReadData;
    wire [31 : 0] MemData;
    DataMemory data_memory(
                        .clk(clk),
                        .address(AluRes_ExMa),
                        .writeData(ReadData2_ExMa),
                        .memWrite(InstCtr_ExMa[3]),
                        .memRead(InstCtr_ExMa[2]),
                        .readData(MemReadData)
                        );
    Mux reg_mem_selector(
                    .select(InstCtr_ExMa[1]),
                    .input1(MemReadData),
                    .input2(AluRes_ExMa),
                    .out(MemData)
                    );
MA-----------------------------------------------------------------//
```

在 MA 阶段需要增加一个选择器，用于选择经过 MA 段后继续向后传递的数据。如果指令是访存操作，那么向后传递的数据来自内存，否则该数据应该是 ALU 的计算结果。

下面介绍 MA 和 WB 之间的段寄存器。

```
//          MA to WB
reg InstCtr_MaWb;
reg [31 : 0] MemData_MaWb;
reg [4 : 0] RegDst_MaWb;
```

首先上文提到的 MA 段向后传递的数据需要被写入段寄存器，此外是否需要写回寄存器相关的控制信号、写回寄存器的编号都需要进行传递。

最后是 WB 段的数据和控制信号通路。

```
WB----------------------------------------------------------------//
  assign WriteReg = RegDst_MaWb;
  assign WriteData = MemData_MaWb;
  assign RegWrite = InstCtr_MaWb;
WB----------------------------------------------------------------//
```

下面介绍流水线的补充功能，包括 Stall、Forwarding、predict-not-taken 的功能。

首先介绍 Pc 的更新方式。这里运用了predict-not-taken 等方式提高流水线的性能。由于 CPU 支持多条转移指令，因此需要多个选择器，选出下一次加载到流水线中的指令地址。

选择器的判断条件是在 ID 阶段完成的 Ctr 控制信号，因此在 ID 阶段就可以获取下一条指令的地址。

```
wire [31 : 0] Pc_Jump;
wire [31 : 0] Pc_Jr;
wire [31 : 0] Pc_Beq;
wire [31 : 0] Pc_Bne;
wire BneSign = InstCtr_IdEx[4] & (~ zero);
wire BeqSign = InstCtr_IdEx[5] & zero;
Mux Jump_Select(
            .select(InstCtr[12]),
            .input1(((Pc_IfId + 4) & 32'hf0000000) + (Inst_IfId [25 : 0] << 2)),
            .input2(Pc + 4),
            .out(Pc_Jump));
Mux Jr_Select(
           .select(InstCtr[11]),
           .input1(ReadData1),
           .input2(Pc_Jump),
           .out(Pc_Jr)
           );
Mux Beq_Select(
            .select(BeqSign),
            .input1(BranchDst),
            .input2(Pc_Jr),
            .out(Pc_Beq)
            );
Mux Bne_Select(
            .select(BneSign),
            .input1(BranchDst),
            .input2(Pc_Beq),
            .out(Pc_Bne)
            );
wire Branch = BeqSign | BneSign;
```

从上面可以看出，转移指令共有四种，用四个选择器选择后得到的最终输出就是下一条需要加载的地址。此处 Branch 信号代表是否跳转。

下面介绍前向通路。

```
wire [31 : 0] InputA_Forward_Temp;
wire [31 : 0] InputB_Forward_Temp;
Mux InputA_Forward_Select1(
                .select(InstCtr_MaWb & (RegDst_MaWb == InstRs_IdEx)),
                .input1(MemData_MaWb),
                .input2(ReadData1_IdEx),
                .out(InputA_Forward_Temp)
                );
Mux InputA_Forward_Select2(
                .select(InstCtr_ExMa[0] & (RegDst_ExMa == InstRs_IdEx)),
                .input1(AluRes_ExMa),
                .input2(InputA_Forward_Temp),
                .out(InputA_Forward)
                );
Mux InputB_Forward_Select1(
                .select(InstCtr_MaWb & (RegDst_MaWb == InstRt_IdEx)),
                .input1(MemData_MaWb),
                .input2(ReadData2_IdEx),
                .out(InputB_Forward_Temp)
                );
Mux InputB_Forward_Select2(
                .select(InstCtr_ExMa[0] & (RegDst_ExMa == InstRt_IdEx)),
                .input1(AluRes_ExMa),
                .input2(InputB_Forward_Temp),
                .out(InputB_Forward)
                );
```

InputA 和 InputB 的情况几乎一致。首先判断下一条指令所读取的寄存器结果是否与上一条指令即将更新的寄存器相同，前向通路的值与 ID 段读寄存器的值之间选出 Temp ，紧接着 Temp 再与之前的 ALU 计算结果，根据指令的种类选出最终读入下一条指令 ALU 的数据。

通过前向通路，可以解除 load-use 的数据竞争冒险，提高流水线性能。

最后介绍暂停。

```
wire Stall = InstCtr_IdEx[2] &
    ((InstRt_IdEx == Inst_IfId [25 : 21]) | (InstRt_IdEx == Inst_IfId [20 : 16]));
```

设置 Stall 信号，当出现上一条指令的目标寄存器和下一条指令的源寄存器相同时，设置 Stall 信号，在后面的处理中与前面提到的 Branch 信号一样，适时暂停流水线。

最后介绍流水线的推进过程。由于前面介绍各个阶段输入输出信号以及段与段之间段寄存器时已经详细介绍了数据和控制信号的传递过程，因此这里不再赘述。

```
initial Pc <= 0;
always @(reset)
begin
    if (reset)
    begin
        Pc <= 0; Inst_IfId <= 0; Pc_IfId <= 0; AluOp_IdEx <= 0;
        InstCtr_IdEx <= 0; ExtRes_IdEx <= 0; InstRs_IdEx <= 0;
        InstRt_IdEx <= 0; ReadData1_IdEx <= 0; ReadData2_IdEx <= 0;
        Funct_IdEx <= 0; Shamt_IdEx <= 0; RegDst_IdEx <= 0;
        InstCtr_ExMa <= 0; AluRes_ExMa <= 0; ReadData2_ExMa <= 0;
        RegDst_ExMa <= 0; InstCtr_MaWb <= 0; MemData_MaWb <= 0;
        RegDst_MaWb <= 0;
    end
```

```
always @(posedge clk)
begin
    // MA - WB
    InstCtr_MaWb <= InstCtr_ExMa [0];
    MemData_MaWb <= MemData; RegDst_MaWb <= RegDst_ExMa;
    // EX - MA
    InstCtr_ExMa <= InstCtr_IdEx [3 : 0]; AluRes_ExMa <= AluRes;
    ReadData2_ExMa <= InputB_Forward; RegDst_ExMa <= RegDst_IdEx;
    // ID - EX
    if (Stall || Branch) begin
        Pc_IdEx <= Pc_IfId; AluOp_IdEx <= 4'hf; InstCtr_IdEx <= 0;
        ExtRes_IdEx <= 0; InstRs_IdEx <= 0; InstRt_IdEx <= 0;
        ReadData1_IdEx <= 0; ReadData2_IdEx <= 0;
        Funct_IdEx <= 0; Shamt_IdEx <= 0; RegDst_IdEx <= 0;
    end else begin
        Pc_IdEx <= Pc_IfId; AluOp_IdEx <= AluOp;
        InstCtr_IdEx <= InstCtr [7 : 0]; ExtRes_IdEx <= ExtRes;
        InstRs_IdEx <= Inst_IfId [25 : 21]; InstRt_IdEx <= Inst_IfId [20 : 16];
        ReadData1_IdEx <= ReadData1; ReadData2_IdEx <= ReadData2;
        Funct_IdEx <= Inst_IfId [5 : 0]; Shamt_IdEx <= Inst_IfId [10 : 6];
        RegDst_IdEx <= RegDst;                end
    // IF - ID
    if (! Stall)        Pc <= Pc_Bne;
    if (Branch || InstCtr[12] || InstCtr[11]) begin
        Inst_IfId <= 0; Pc_IfId <= 0;
    end else if (!Stall) begin
        Inst_IfId <= Inst; Pc_IfId <= Pc;             end
end

endmodule
```

可以看到，上面在 ID - EX 段，如果出现跳转/暂停情况，则清空流水线。在 IF - ID 段，在不需要进行暂停时，读取下面一条指令，并加载到流水线中。这是 Stall 的原理。

2.2 Ctr - ALUCtr - ALU 相关拓展

由于本实验完成了拓展，实现了支持全部的 31 条 MIPS 指令的流水线 CPU ，因此需要更多的控制信号，ALU 需要支持更多种类的运算。以下介绍 Ctr - ALUCtr - ALU 的拓展。

首先是 Ctr 的拓展，主要增加了空指令 nop、新增控制信号 beqSign 和 bneSign，代替 lab5 中的单一一条 branch 指令。同时 Ctr 发出的 aluOP 进一步扩展，由 3 位变成了 4 位。

```
`timescale 1ns / 1ps
module Ctr(
    input [5 : 0] opCode, input [5 : 0] funct, input nop, output reg regDst,
    output reg aluSrc, output reg memToReg, output reg regWrite,
    output reg memRead, output reg memWrite, output reg beqSign,
    output reg bneSign, output reg luiSign, output reg extSign,
    output reg jalSign, output reg jrSign, output reg [3 : 0] aluOp, output reg jump );
    always @(opCode or funct or nop)
    begin       if(nop)//all zero
          begin         regDst = 0; aluSrc = 0; regWrite = 0; memToReg = 0;
                        memRead = 0; memWrite = 0; beqSign = 0; bneSign = 0;
                        luiSign = 0; extSign = 0;  jalSign = 0;  jrSign = 0;
                        aluOp = 4'b1111;  jump = 0;             end
        else begin
```

```
        case(opCode)
            6'b000000: //R-Type:add(u)/sub(u)/and/or/xor/nor/sll(v)/srl(v)/sra(v)/
slt(u)/jr
            begin
                if(funct == 6'b001000)    //jr
                begin
                    regDst = 0; regWrite = 0; jrSign = 1; aluOp = 4'b1111;
                end
                else begin
                    regDst = 1; regWrite = 1; jrSign = 0; aluOp = 4'b1101;
                end
                aluSrc = 0; memToReg = 0; memRead = 0; memWrite = 0;
                beqSign = 0; bneSign = 0; luiSign = 0; extSign = 0;
                jalSign = 0; jump = 0;                  end
            6'b001000:     // addi
            begin
                regDst = 0; aluSrc = 1; regWrite = 1; memToReg = 0;
                memRead = 0; memWrite = 0; beqSign = 0; bneSign = 0;
                luiSign = 0; extSign = 1; jalSign = 0; jrSign = 0;
                aluOp = 4'b0000; jump = 0;                  end
            6'b001001:     // addiu
            begin
                regDst = 0; aluSrc = 1; regWrite = 1; memToReg = 0;
                memRead = 0; memWrite = 0; beqSign = 0; bneSign = 0;
                luiSign = 0; extSign = 0; jalSign = 0; jrSign = 0;
                aluOp = 4'b0001; jump = 0;
            end
```

```
6'b001100:      // andi
begin
   regDst = 0; aluSrc = 1; regWrite = 1; memToReg = 0;
   memRead = 0; memWrite = 0; beqSign = 0; bneSign = 0;
   luiSign = 0; extSign = 0; jalSign = 0; jrSign = 0;
   aluOp = 4'b0100; jump = 0;                end
6'b001101:      // ori
begin
   regDst = 0; aluSrc = 1; regWrite = 1; memToReg = 0;
   memRead = 0; memWrite = 0; beqSign = 0; bneSign = 0;
   luiSign = 0; extSign = 0; jalSign = 0; jrSign = 0;
   aluOp = 4'b0101; jump = 0;                end
6'b001110:      // xori
begin
   regDst = 0; aluSrc = 1; regWrite = 1; memToReg = 0;
   memRead = 0; memWrite = 0; beqSign = 0; bneSign = 0;
   luiSign = 0; extSign = 0; jalSign = 0; jrSign = 0;
   aluOp = 4'b0110; jump = 0; end
6'b001111:      // lui
begin
   regDst = 0; aluSrc = 1; regWrite = 1; memToReg = 0;
   memRead = 0; memWrite = 0; beqSign = 0; bneSign = 0;
   luiSign = 1; extSign = 0; jalSign = 0; jrSign = 0;
   aluOp = 4'b1010; jump = 0;                end
6'b100011:      // lw
begin
   regDst = 0; aluSrc = 1; regWrite = 1; memToReg = 1;
   memRead = 1; memWrite = 0; beqSign = 0; bneSign = 0;
   luiSign = 0; extSign = 1;  jalSign = 0; jrSign = 0;
   aluOp = 4'b0001; jump = 0;                end
```

```
6'b101011:     // sw
begin
   regDst = 0; aluSrc = 1; regWrite = 0; memToReg = 0;
   memRead = 0; memWrite = 1; beqSign = 0; bneSign = 0;
   luiSign = 0; extSign = 1; jalSign = 0; jrSign = 0;
   aluOp = 4'b0001; jump = 0;                end
6'b000100:     // beq
begin
   regDst = 0; aluSrc = 0; regWrite = 0; memToReg = 0;
   memRead = 0; memWrite = 0; beqSign = 1; bneSign = 0;
   luiSign = 0; extSign = 1; jalSign = 0; jrSign = 0;
   aluOp = 4'b0011; jump = 0;                end
6'b000101:    // bne
begin
    regDst = 0; aluSrc = 0; regWrite = 0; memToReg = 0;
    memRead = 0; memWrite = 0; beqSign = 0; bneSign = 1;
    luiSign = 0; extSign = 1; jalSign = 0;  jrSign = 0;
    aluOp = 4'b0011; jump = 0;                end
```

……(由于篇幅过长，以下略写）……

slti: beq = 0; bne = 0; lui = 0; jal = 0; aluop = 1000;

sltiu: 同上; aluop = 1001

jump: beq = 0; bne = 0; lui = 0; jal = 0; jump = 1; aluop = 1111;

jal: beq = 0; bne = 0; lui = 0; jal = 1; jump = 0; aluop = 1111;

default: beq = 0; bne = 0; lui = 0; jal = 0; jump = 0; aluop = 1111;

ALUCtr 根据拓展后的 aluOp 和 funct 的情况，给出具体的四位 aluctrout ，控制 ALU 的运算过程。

```
`timescale 1ns / 1ps
module ALUCtr( input [3 : 0] aluOp, input [5 : 0] funct, output reg [3 : 0]
aluCtrOut, output reg shamtSign );
   always @ (aluOp or funct)
   begin       shamtSign = 0;
     if (aluOp == 4'b1101)
        begin
      case (funct)
          6'b100000:     aluCtrOut = 4'b0000; // add
          6'b100001:     aluCtrOut = 4'b0001;// addu
          6'b100010:     aluCtrOut = 4'b0010; // sub
          6'b100011:     aluCtrOut = 4'b0011;// subu
          6'b100100:     aluCtrOut = 4'b0100;// and
          6'b100101:     aluCtrOut = 4'b0101;// or
          6'b100110:     aluCtrOut = 4'b0110;// xor
          6'b100111:     aluCtrOut = 4'b0111;// nor
          6'b101010:     aluCtrOut = 4'b1000;// slt
          6'b101011:     aluCtrOut = 4'b1001;// sltu
          6'b000100:     aluCtrOut = 4'b1010;//sllv
          6'b000110:     aluCtrOut = 4'b1011;// srlv
          6'b000111:     aluCtrOut = 4'b1100;// srav
          6'b001000:     aluCtrOut = 4'b1111;// jr
          6'b000000:     aluCtrOut = 4'b1010,shamtSign = 1;//sll
          6'b000010:     aluCtrOut = 4'b1011,shamtSign = 1;//srl
          6'b000011:     aluCtrOut = 4'b1100,shamtSign = 1;//sra
          default: aluCtrOut = 4'b1111; endcase end
       else
            aluCtrOut = aluOp;
```

此处为了方便，将立即数操作相关指令的 aluOp 直接作为 aluCtrOut ，控制 ALU 的运算方式。

相比 lab5 ，ALU 新增的功能如下所示。

ALU	aluctrout	function
异或（xor）	4’b0110	res = a ^ b
取反（nor）	4’b0111	res = ~(a | b)
无符号设置（sltu）	4’b0101	res = a < b
算数右移（srlu）	4’b0111	res = sign(a) >>> sign(b)

3、仿真验证

3.1 multi_cycle_cpu_tb

此处初始化的方式与 lab5 相同，因此不再赘述。本实验我自己编写了应用额外 15 条指令的二进制机器代码，并设置了相关的数据冒险。

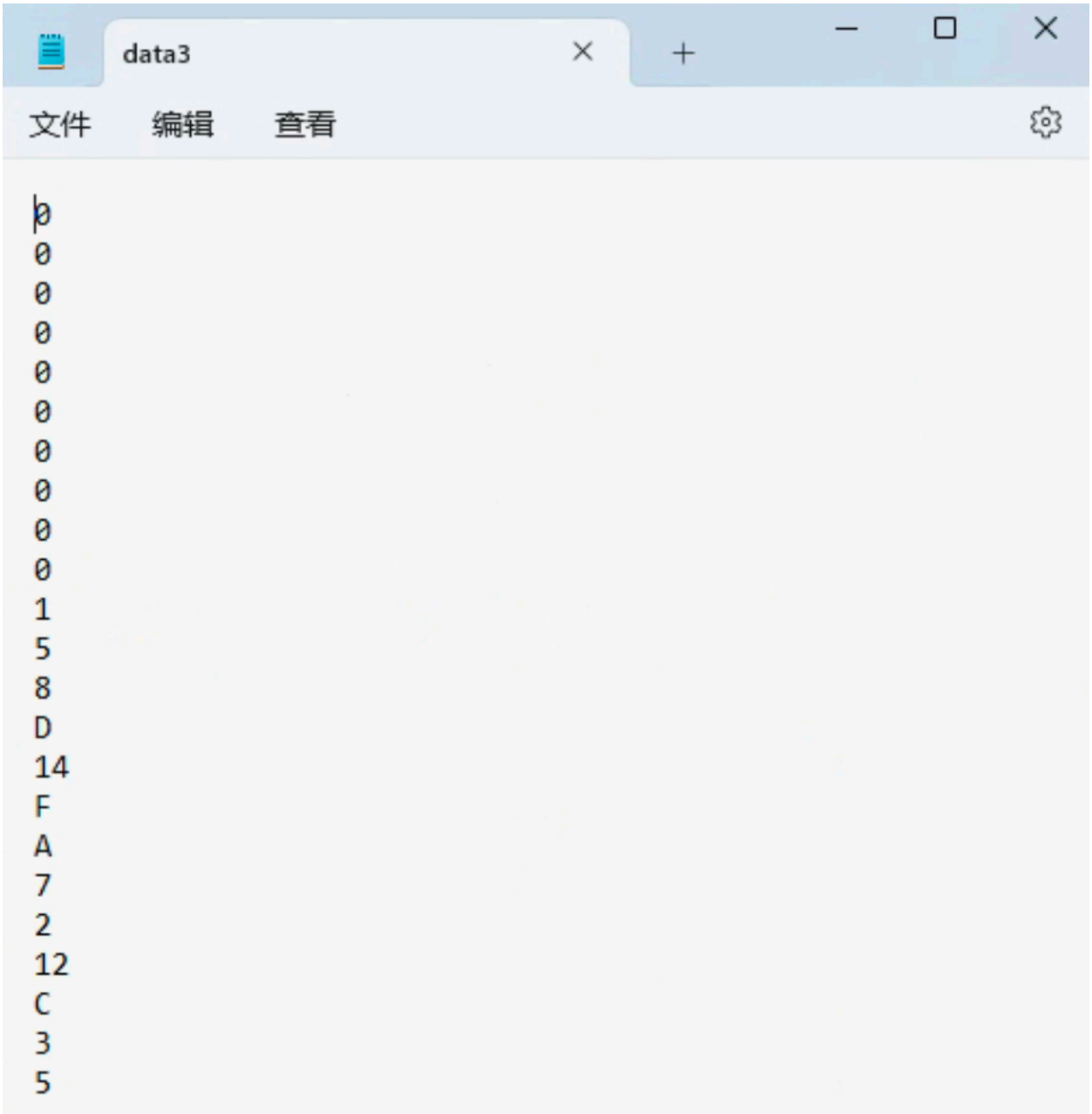
激励文件如下。

```
`timescale 1ns / 1ps
module multi_cycle_cpu_tb( );
    reg clk; reg reset;
    Top processor(.clk(clk), .reset(reset));

    initial begin
        $readmemb("D:/archlabs/lab06/inst_data3.dat",
processor.inst_memory.instFile);
        $readmemh("D:/archlabs/lab06/data3.dat",
processor.data_memory.memFile);
        reset = 1;
        clk = 0;
    end

    always #20 clk = ~clk;
```

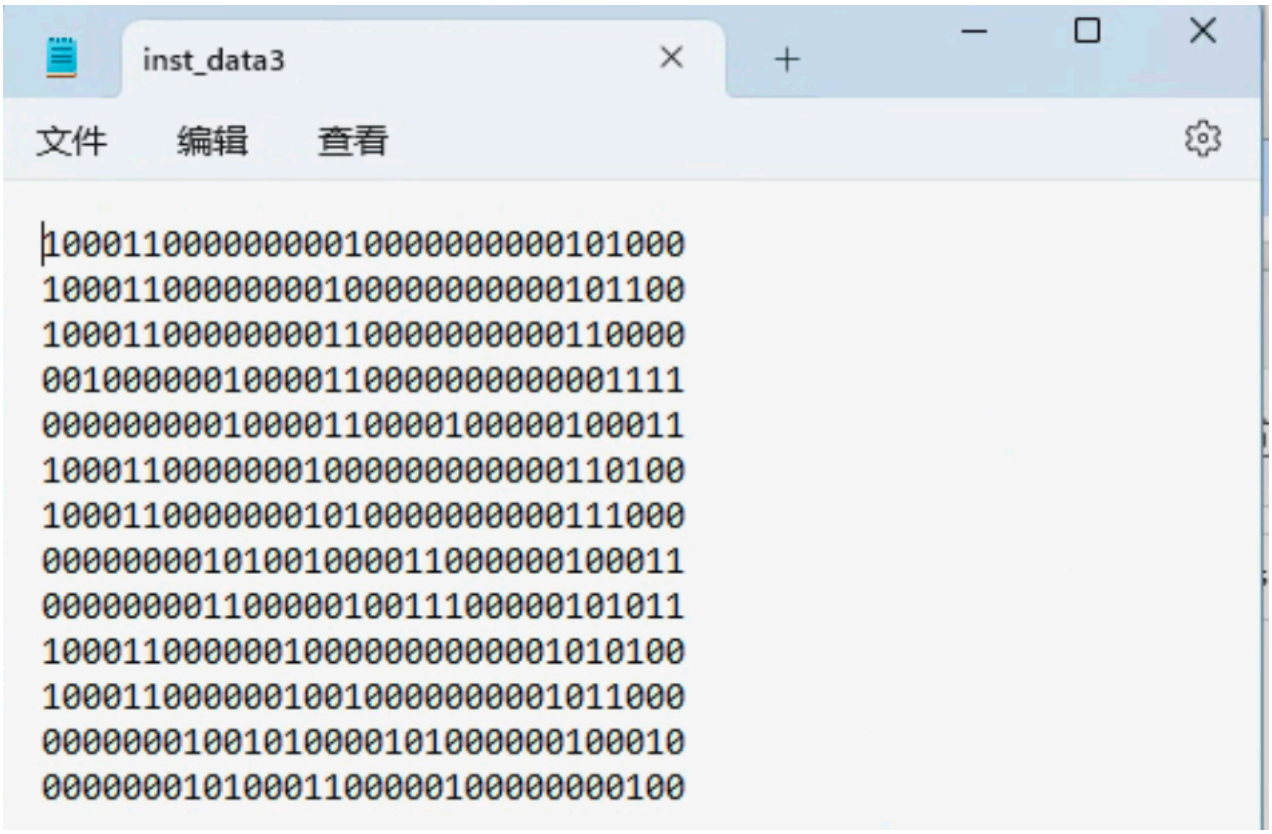
```
    initial begin
        #40 reset = 0;
        #2000;
        $finish;
    end
endmodule
```

3.2 仿真验证（一）

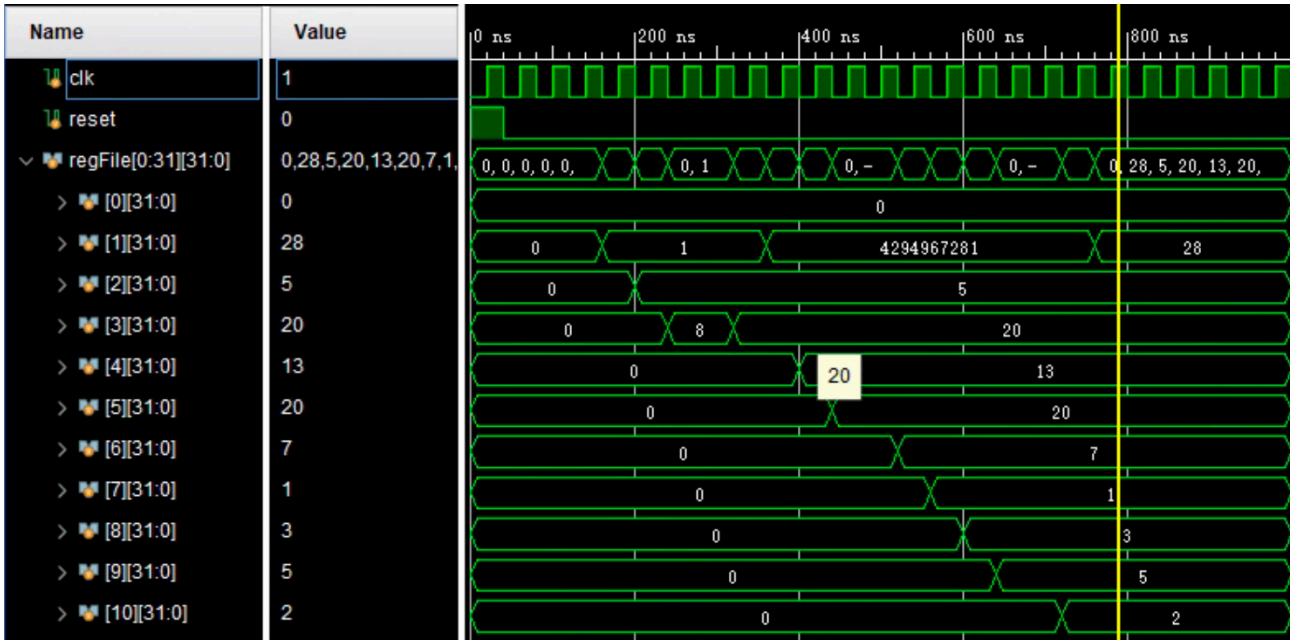
data：



inst

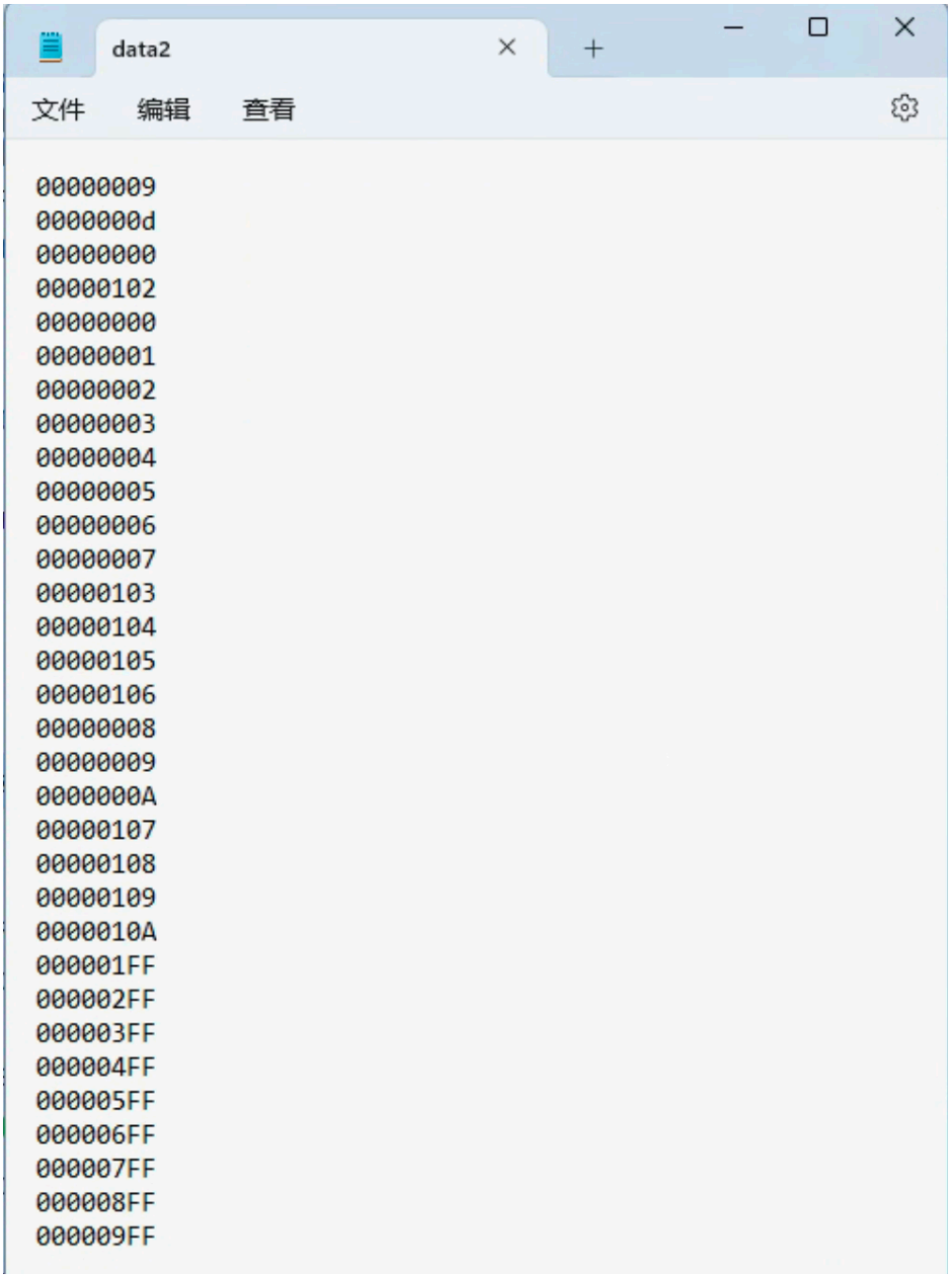


对应的仿真结果如下（助教学姐已经检查了仿真结果的正确性）。

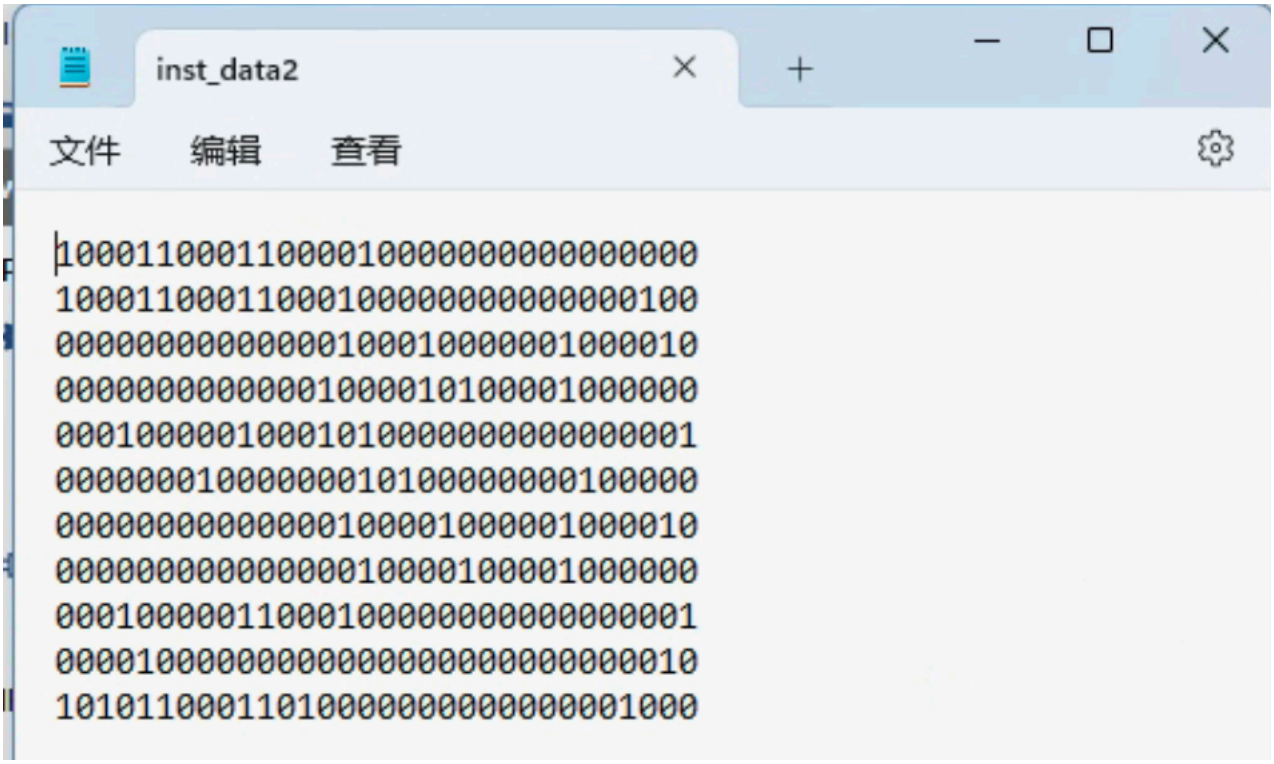


3.3 仿真验证（二）

data：



inst：



对应的仿真结果如下。（本仿真即 lab5 中助教老师提供的仿真代码，验证乘法的结果是否正确）

可以看到在 1720ns 左右，乘法的结果被写入第三个内存。

4、致谢

感谢刘老师以及三位实验助教学长学姐的悉心指导和帮助。

本实验是该课程的最后一个实验项目，难度非常大，我提前四天开始着手准备这个实验的通路设计和代码编写，但是仍然遇到很多问题，感谢几位助教学长学姐在课程群内提供的及时指导以及验证代码。

完成本次实验后，该课程内容正式结束。回顾六周以来的课程学习，我受益匪浅，从最开始的只会比照着实验指导书，一行一行写代码，到最后可以完成一个多周期 MIPS 流水线处理器的设计，感谢计算机系的课程设计，让我在短短的六周里取得如此大的进步。

感谢上海交通大学的计算机系统结构实验室，为我们准备了相关的实验环境以及 FPGA 开发板，学校和学院提供的优秀设备让我们受益匪浅。