计算机系统结构实验 Lab5

David Wang

2020 年 5 月

1 概述

1.1 实验名称

类 MIPS 单周期处理器的设计与实现

1.2 实验目的

完成单周期的类 MIPS 处理器，设计支持 16 条指令的单周期 CPU.

1.3 实验内容

1. Instruction memory 等模块的实现
2. 单周期 CPU 的实现与调试
3. 功能仿真

2 指令分析

2.1 之前实验已经完成的指令

之前实验已经完成的指令可以直接拿来复用。

2.2 有常数作为运算数的算术指令

本次实验要求实现 addi、andi、ori 三条有常数作为运算数的指令。在处理时，需要借助多路选择器将位扩展后的运算数送入 ALU 的第二个操作数。

2.3 移位指令

移位指令需要使用到指令中的 shamt 域，是指令的第 6-10 位。而移位指令的 rs 域总为 0。所以可以将 shamt 域送入 ALU 的第一个操作数。这时需要做额外的判断工作。

2.4 跳转指令

之前已经实现的 j 指令无需太多的修改。而 jr 指令需要读取寄存器，将寄存器的值作为跳转的目标地址。jal 需要写寄存器，将跳转前的 PC 值写入 $31。

3 顶层模块

顶层模块只需要将之前几次实验完成的模块实例化，并添加一些信号线将这些模块连接起来。在此之前，需要完成指令存储器的构建

3.1 指令存储器

指令存储器的构建十分简单。

```
    module instrMemory(
    input [31:0] readAddress,
    output reg [31:0] instruction
    );
    reg [31:0] instrFile[31:0];
    always @ (readAddress)
    begin
        instruction=instrFile[readAddress>>2];
    end
endmodule
```

本次实验中所需要设计的指令存储器与 MIPS 指令集架构相适应，给出地址后输出四个字节的数据作为一条指令。且要求指令必须对齐，即输入的地址必须为 4 的倍数才能保证给出正确的指令。如果指令在存储时没有对齐，则即使给出正确地址也不保证输出正确的指令。目前主流的内存都会在接收一个地址作为输入后向 CPU 输出多个字节的数据，以减少访问相同字节时的时间损耗。英特尔家族的 CPU 能够保证在内存中的数据没有对齐时依然可以向内存发出多次请求访问，并将得到的数据拼接在一起从而得出正确结果。而其他的一些 CPU 不保证数据不对齐时能正确访问到数据。无论是从 CPU 的角度还是从存储器的角度，按字节寻址，但一次性输出 4 字节数据都是合理的。

3.2 程序计数器和多路选择器

程序计数器实现为一个单独的寄存器类型的变量。多路选择器使用三目运算符实现。

3.3 信号线的定义

```
    wire REG_DST,JUMP,JUMP1,BRANCH,MEM_READ;
    wire MEM_TO_REG,MEM_WRITE,JUMPTARGET,CALL;
    wire [2:0] ALU_OP;
    wire ALU_SRC,REG_WRITE;

    wire [3:0] ALUCTR;

    wire [31:0] INST;

    wire [4:0] WRITEREG;
    wire [4:0] READREG1;
    wire [4:0] READREG2;
    wire [31:0] REGREADDATA1;
    wire [31:0] REGREADDATA2;
    wire [31:0] REGWRITEDATA;

```

```
    wire [31:0] INSTSHIFTED;
    wire [31:0] SIGNEXTENDED;
    wire [31:0] EXTENDSHIFTED;

    wire [31:0] ALUSRC1;
    wire [31:0] ALUSRC2;
    wire ZERO;
    wire BRANCHOPERAND;
    wire [31:0] ALURSLT;

    wire [31:0] MEM_DATA;
    wire [31:0] PC_PLUS_4;
```

3.4 模块的修改

3.4.1 指令存储器

指令存储器需要读取 PC 值，以 PC 值作为地址给出相应的指令。

```
    reg [31:0] PC;
    instrMemory instruction_memory(
         .readAddress(PC),
         .instruction(INST)
     );
```

3.4.2 控制模块

控制模块需要读入指令的高 6 位，输出控制信号。因为需要支持的指令比之前多，所以需要进行一些修改。主要是引入 call 数据线用来表示是否存储跳转前的 PC 值，引入 jumpTarget 数据线用来选择跳转地址来源，以及将 ALUop 扩展到三位以表示更多的指令。

```
    module Ctr(
    input [5:0] opCode,
    output reg regDest,
    output reg [1:0] aluSrc,
    output reg memToReg,
    output reg regWrite,
    output reg memRead,
    output reg memWrite,
    output reg Branch,
    output reg [2:0] ALUop,
    output reg Jump,
    output reg jumpTarget,
    output reg Call
    );
    always @(opCode)
    begin
        case(opCode)
            6'b000000://add,sub,and,or,slt,jr
            begin
                regDest=1;
```

```
                aluSrc=0;
                memToReg=0;
                regWrite=1;
                memRead=0;
                memWrite=0;
                Branch=0;
                Jump=0;
                jumpTarget=0;
                Call=0;
                ALUop=3'b010;
            end
            6'b000010://jump
            begin
                regDest=0;
                aluSrc=0;
                memToReg=0;
                regWrite=0;
                memRead=0;
                memWrite=0;
                Branch=0;
                Jump=1;
                jumpTarget=1;
                Call=0;
                ALUop=3'b000;
            end
            6'b000011://jal
            begin
                regDest=0;
                aluSrc=0;
                memToReg=0;
                regWrite=1;
                memRead=0;
                memWrite=0;
                Branch=0;
                Jump=1;
                jumpTarget=1;
                Call=1;
                ALUop=3'b000;
            end
            6'b001000://addi
            begin
                regDest=0;
                aluSrc=1;
                memToReg=0;
                regWrite=1;
                memRead=0;
                memWrite=0;
                Branch=0;
                Jump=0;
                jumpTarget=0;
```

```
                Call=0;
                ALUop=3'b000;
            end
            6'b001100://andi
            begin
                regDest=0;
                aluSrc=1;
                memToReg=0;
                regWrite=1;
                memRead=0;
                memWrite=0;
                Branch=0;
                Jump=0;
                jumpTarget=0;
                Call=0;
                ALUop=3'b000;
            end
            6'b001101://ori
            begin
                regDest=0;
                aluSrc=1;
                memToReg=0;
                regWrite=1;
                memRead=0;
                memWrite=0;
                Branch=0;
                Jump=0;
                jumpTarget=0;
                Call=0;
                ALUop=3'b000;
            end
            6'b100011://lw
            begin
                regDest=0;
                aluSrc=1;
                memToReg=1;
                regWrite=1;
                memRead=1;
                memWrite=0;
                Branch=0;
                Jump=0;
                jumpTarget=0;
                Call=0;
                ALUop=3'b000;
            end
            6'b101011://sw
            begin
                aluSrc=1;
                regWrite=0;
                memRead=0;
```

```
                memWrite=1;
                Branch=0;
                Jump=0;
                jumpTarget=0;
                Call=0;
                ALUop=3'b000;
            end
            6'b000100://beq
            begin
                aluSrc=0;
                regWrite=0;
                memRead=0;
                memWrite=0;
                Branch=1;
                Jump=0;
                jumpTarget=0;
                Call=0;
                ALUop[0]=3'b001;
            end
        endcase
    end
endmodule
```

3.4.3 ALU 控制器

ALU 控制器也需要进行修改。主要是为新增的指令添加控制信号。

```
    module ALUCtr(
    input [2:0] ALUop,
    input [5:0] functField,
    output reg [3:0] operation,
    output reg Jump
    );
    always @ (ALUop or functField)
    begin
        Jump=0;
        casex({ALUop,functField})
            9'b000xxxxxx:operation=4'b0010;//lw,sw,addi,jump,jal
            9'b011xxxxxx:operation=4'b0000;//andi
            9'b100xxxxxx:operation=4'b0001;//ori

            9'b001xxxxxx:operation=4'b0110;//beq
            9'b010100000:operation=4'b0010;//add
            9'b010000000:operation=4'b0011;//sll
            9'b010000010:operation=4'b0100;//srl
            9'b010100010:operation=4'b0110;//subtract
            9'b010100100:operation=4'b0000;//and
            9'b010100101:operation=4'b0001;//or
            9'b010101010:operation=4'b0111;//slt
            9'b010100111:operation=4'b1100;//nor
```

```
            9'b010001000:
            begin
            operation=4'b0010;//jr
            Jump=1;
            end
        endcase
    end
endmodule
```

3.4.4 ALU

ALU 需要处理新增的移位指令。因为移位指令的 rs 域总为 0，所以可以将 shamt 域输入到 input1。

```
    module ALU(
    input [31:0] input1,
    input [31:0] input2,
    input [3:0] aluCtr,
    output reg zero,
    output reg [31:0] aluRes
    );
    always @(input1 or input2 or aluCtr)
    begin
        case(aluCtr)
            4'b0000://add
                begin
                    aluRes=input1 & input2;
                    if(aluRes==0) zero=1;
                    else zero=0;
                end
            4'b0001://or
                begin
                    aluRes=input1 | input2;
                    if(aluRes==0) zero=1;
                    else zero=0;
                end
            4'b0010://addi
                begin
                    aluRes=input1+input2;
                    if(aluRes==0) zero=1;
                    else zero=0;
                end
            4'b0011://sll
                begin
                    aluRes=input2<<input1;
                    if(aluRes==0) zero=1;
                    else zero=0;
                end
            4'b0100://srl
                begin
```

```
                    aluRes=input2>>input1;
                    if(aluRes==0) zero=1;
                    else zero=0;
                end
            4'b0110://sub
                begin
                    aluRes=input1-input2;
                    if(aluRes==0) zero=1;
                    else zero=0;
                end
            4'b0111://slt
                begin
                    if(input1<input2) aluRes=1;
                    else aluRes=0;
                end
            4'b1100://nor
                begin
                    aluRes=~(input1|input2);
                    if(aluRes==0) zero=1;
                    else zero=0;
                end
        endcase
    end
endmodule
```

3.4.5 跳转逻辑

跳转指令和分支指令需要控制器解析控制信号。针对特殊指令如 jal，需要特殊处理，引入 call 数据线。根据控制信号，决定是否写入 PC，写入哪一个新值。需要注意，跳转指令的目标地址可能来自常数或寄存器。

```
    Ctr ctr(
         .opCode(INST[31:26]),
         .regDest(REG_DST),
         .aluSrc(ALU_SRC),
         .memToReg(MEM_TO_REG),
         .regWrite(REG_WRITE),
         .memRead(MEM_READ),
         .memWrite(MEM_WRITE),
         .Branch(BRANCH),
         .ALUop(ALU_OP),
         .Jump(JUMP),
         .Call(CALL),
         .jumpTarget(JUMPTARGET)
     );
     wire [31:0] NEXTPC;
     assign PC_PLUS_4=PC+4;
     assign NEXTPC=(BRANCH&ZERO)?
     (PC_PLUS_4+(SIGNEXTENDED<<2)):
     ((JUMP||JUMP1)?
```

```
        (JUMPTARGET?{PC_PLUS_4[31:28],INST[25:0]<<2}:
        ALURSLT):
            PC_PLUS_4);
    always @(posedge clock)
    begin
        if(reset) PC<=0;
        else PC<=NEXTPC;
    end
```

4 仿真测试

4.1 验证程序

```
    lw $1,0($4)
    lw $2,0($8)
    add $3,$1,$2
    beq $0,$0,1
    xor $3,$1,$2
    j 7
    xor $1,$1,$1
    addi $2,$1,64
    sll $3,$1,2
    srl $6,$3,2
    sw $2,0($4)
    jal 1
    xor $2,$2,$2
```

上面的程序中，有加载指令、存储指令、分支指令、两种跳转指令，还有三种前几次实验没有要求的指令。在跳转后面紧跟一个将寄存器清零的指令，可以较好地验证分支指令和跳转指令执行的正确性。

4.2 顶层模块控制程序

```
    module top_tb();
    reg clock,reset;
    always #50 clock=!clock;
    Top top(.clock(clock),.reset(reset));
    initial begin
        $readmemh("mem_data.txt",top.data_memory.memFile);
        $readmemh("inst.txt",top.instruction_memory.instrFile);
        clock=1;
        reset=1;
        #75
        reset=0;
        #2000;
    end
    endmodule
```

4.3 仿真波形图

上面的图片展示了仿真波形中的 PC 值、寄存器、数据存储器的变化情况。从波形图中可以看出，寄存器和数据存储器的数据变化和预期相吻合。跳转指令后将寄存器置零的指令没有执行，说明跳转指令正常。

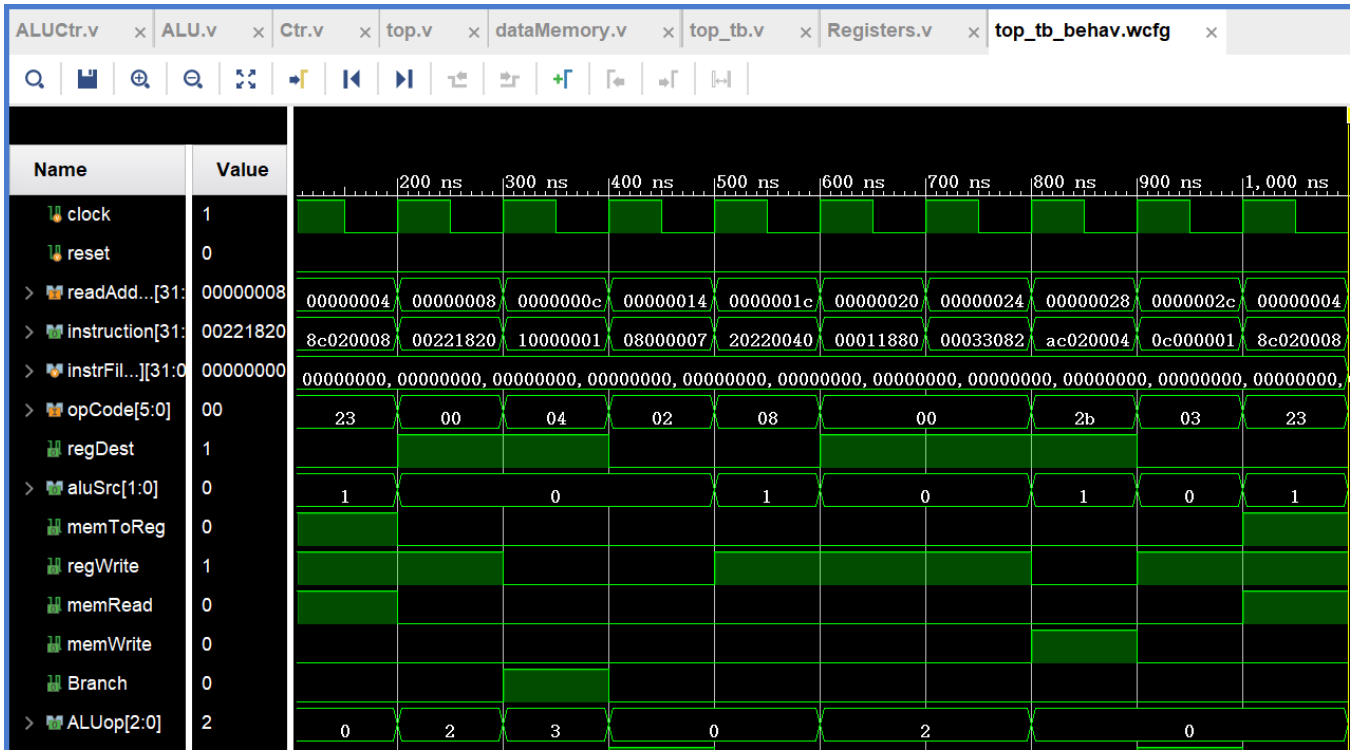

图 1:

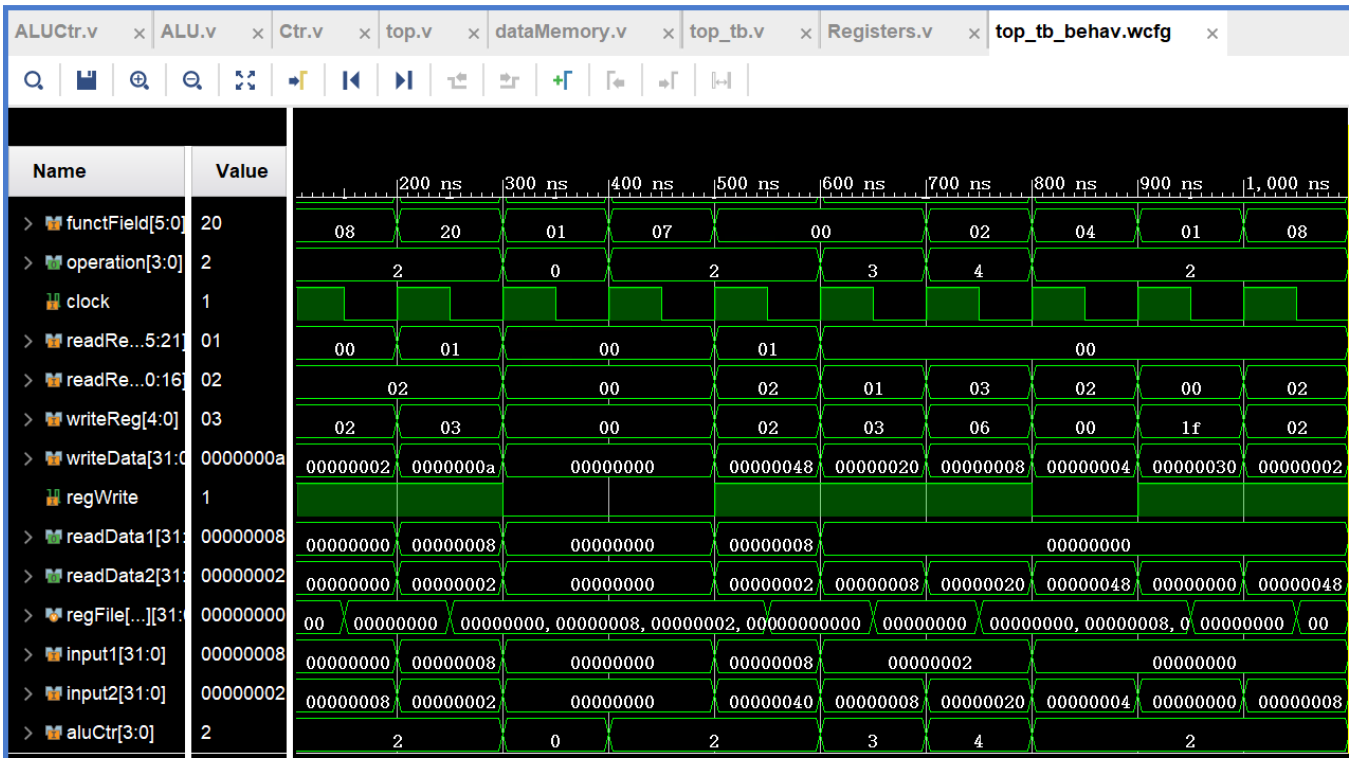
ALUCtr.v
ALU.v
Ctr.v
top.v
dataMemory.v
top_tb.v
Registers.v
top_tb_behav.wcfg
Name
Value
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
1,000 ns
functField[5:0]
operation[3:0]
clock
readRe...5:21]
readRe...0:16]
writeReg[4:0]
writeData[31:0]
regWrite
readData1[31:
readData2[31:
regFile[...][31:
input1[31:0]
input2[31:0]
aluCtr[3:0]

图 2:

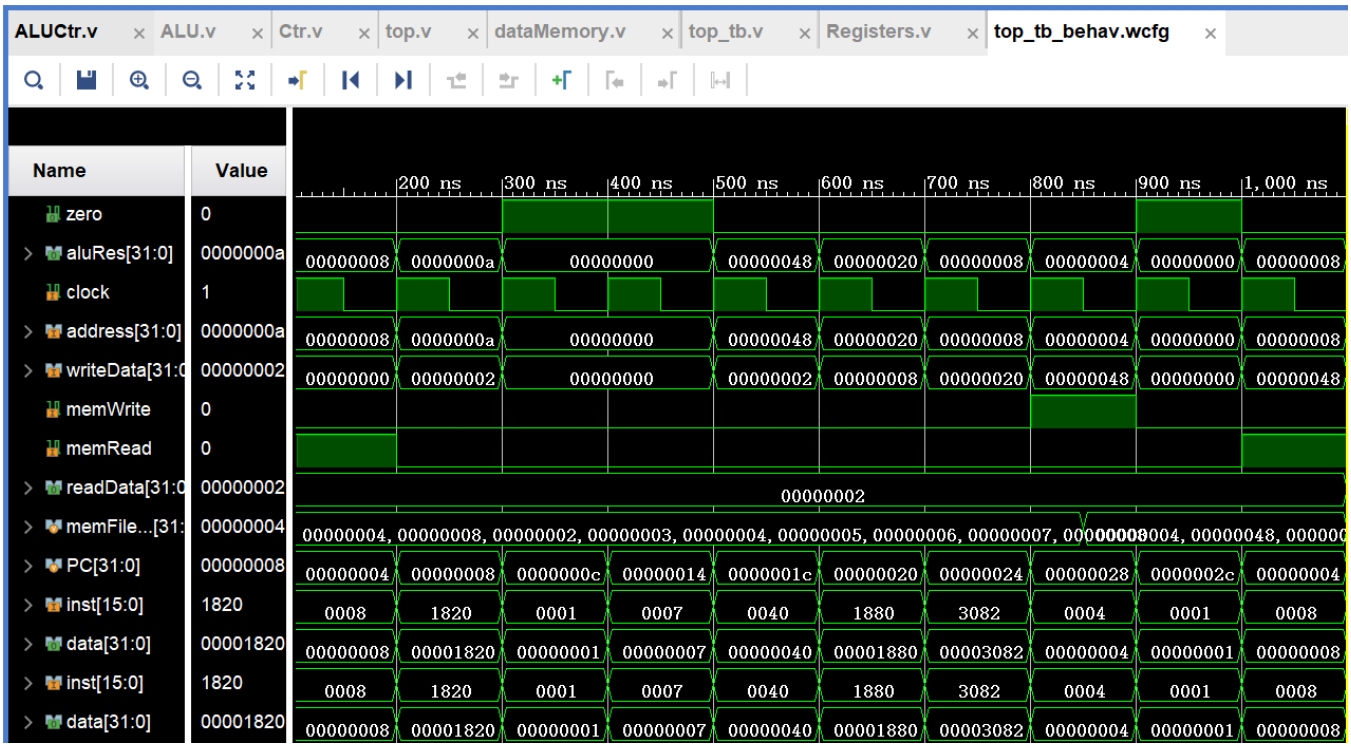
ALUCtr.v
ALU.v
Ctr.v
top.v
dataMemory.v
top_tb.v
Registers.v
top_tb_behav.wcfg
Name
Value
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
1,000 ns
zero
aluRes[31:0]
clock
address[31:0]
writeData[31:0]
memWrite
memRead
readData[31:0]
memFile...[31:
PC[31:0]
inst[15:0]
data[31:0]
inst[15:0]
data[31:0]

图 3:

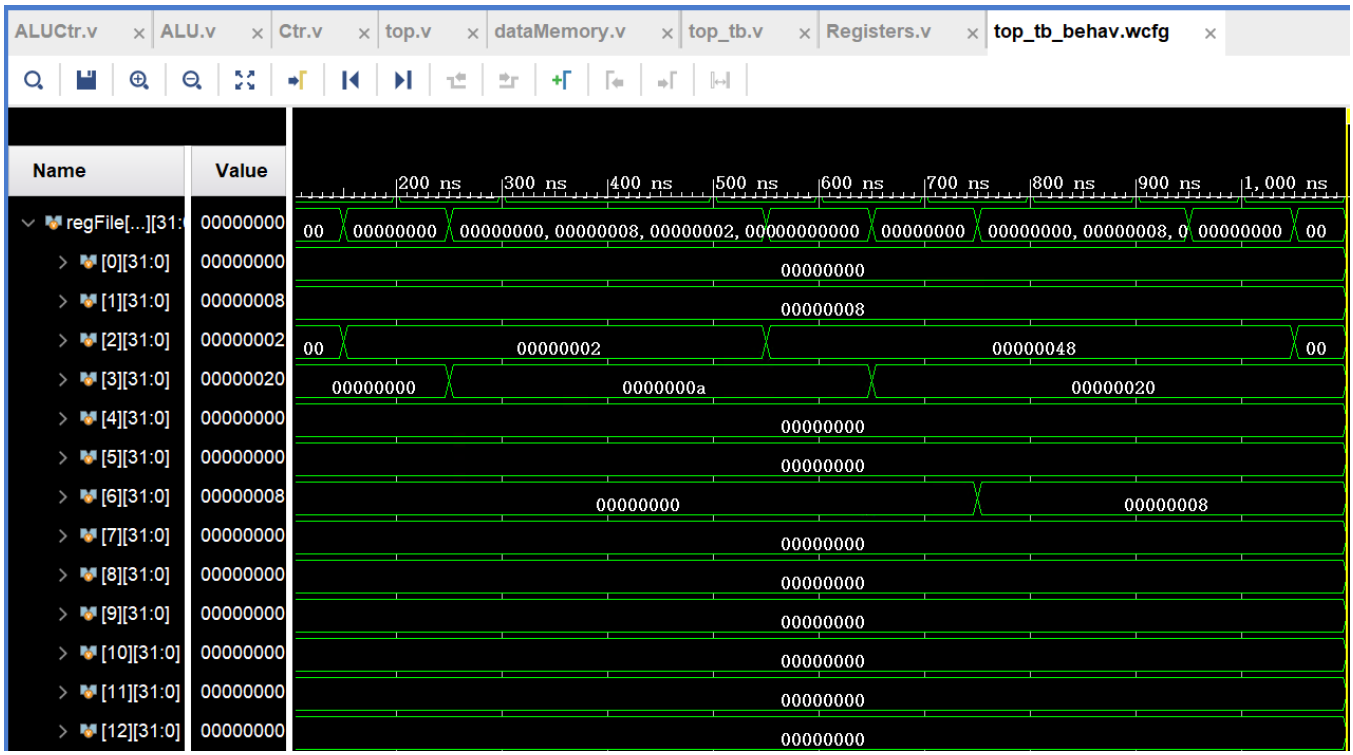
ALUCtr.v
ALU.v
Ctr.v
top.v
dataMemory.v
top_tb.v
Registers.v
top_tb_behav.wcfg
Name
Value
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
1,000 ns
regFile[...][31:
00000000
[0][31:0]
00000000
[1][31:0]
00000008
[2][31:0]
00000002
[3][31:0]
00000020
[4][31:0]
00000000
[5][31:0]
00000000
[6][31:0]
00000008
[7][31:0]
00000000
[8][31:0]
00000000
[9][31:0]
00000000
[10][31:0]
00000000
[11][31:0]
00000000
[12][31:0]
00000000
00
00000000
00000000, 00000008, 00000002, 00000000
00000000
00000000, 00000008, 0
00000000
00
00000002
00000048
0000000a
00000020

图 4:

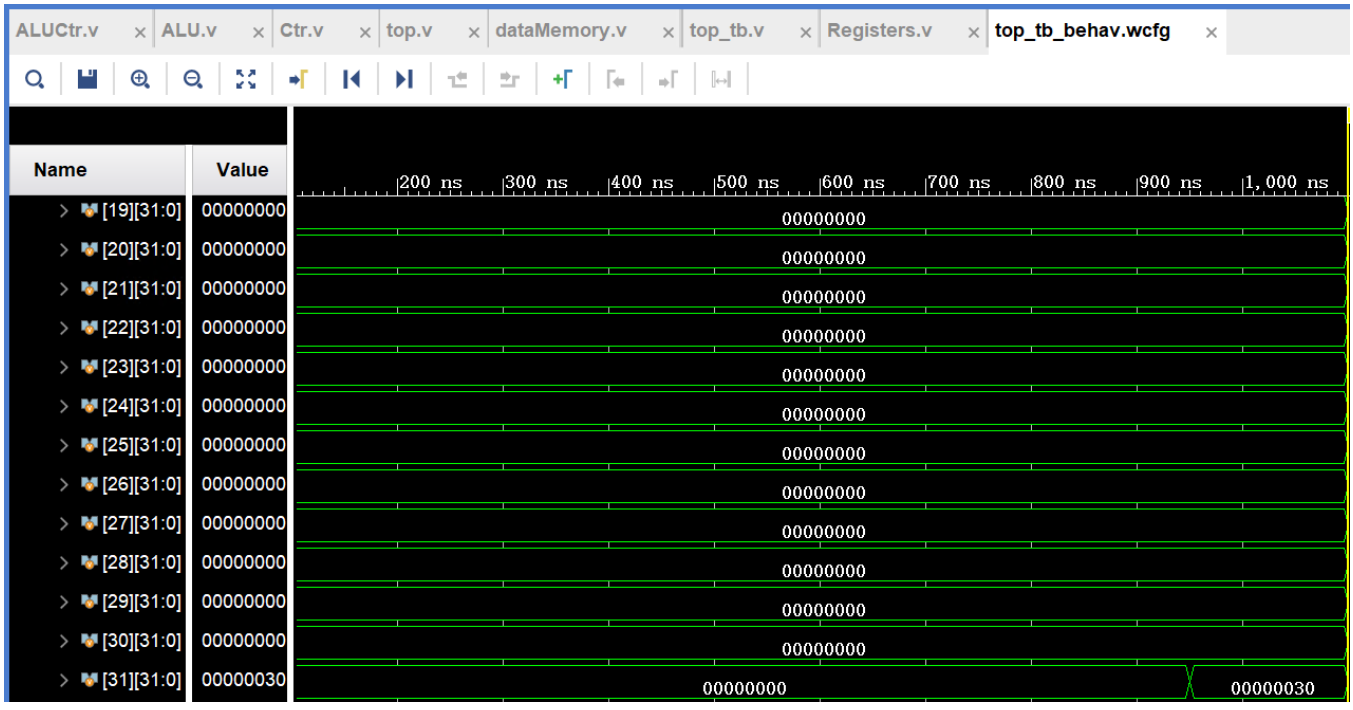
ALUCtr.v
ALU.v
Ctr.v
top.v
dataMemory.v
top_tb.v
Registers.v
top_tb_behav.wcfg
Name
Value
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
1,000 ns
[19][31:0]
00000000
[20][31:0]
00000000
[21][31:0]
00000000
[22][31:0]
00000000
[23][31:0]
00000000
[24][31:0]
00000000
[25][31:0]
00000000
[26][31:0]
00000000
[27][31:0]
00000000
[28][31:0]
00000000
[29][31:0]
00000000
[30][31:0]
00000000
[31][31:0]
00000030
00000030

图 5:

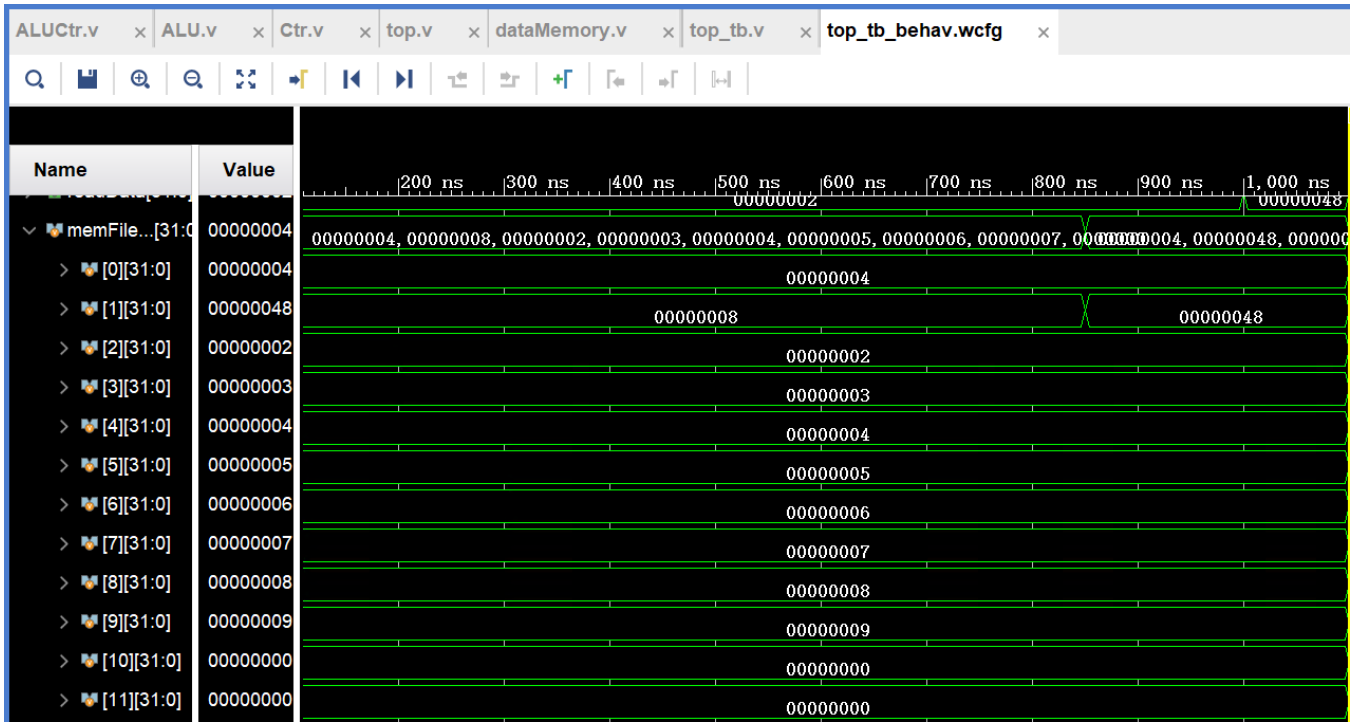

图 6:

5 总结与反思

5.1 实验难点

1. 本次实验中设计到的模块较多，且需要底层模块和顶层模块相配合。因此变量的命名规范很重要。在本次实验中，我统一规定数据线用下划线和大写字母表示，模块中的接口用驼峰命名法命名。如果不按照统一命名规范来命名，则可能出现不知道来源的未定值，给调试带来困难。所以最好要建立命名规范。

2. 本次实验要求支持更多的指令，而新加的指令与原来支持的指令在格式上有很大不同，如移位指令需要使用 shamt 域，常数运算指令的加入需要把 ALUop 扩展到三位。跳转指令的跳转地址可能来自寄存器，甚至有可能把跳转前的地址写入寄存器。这些都需要对原模块进行修改。需要在原来模块的基础上举一反三。

5.2 实验总结与感想

本次实验是对计算机系统结构课程的补充。在计算机系统结构课程中，我们学习了一种类 MIPS 处理器的实现。但这只停留在理论部分。我们对于一些具体的实现细节了解还不够深入。而实验课给了我们一个亲自动手写 CPU 的机会。这加深了我对于理论知识的理解。