计算机系统结构实验Lab05:简单的类 MIPS 单周期处理器的实现–整体调试

张祥东

June 2,2023

摘要

本实验的实验内容是简单的类MIPS单周期处理器的实现和整体调试。实验的主要任务是理解类MIPS单周期处理器的工作原理，完成9条和16条MIPS指令的CPU实现与调试，并进行仿真测试。

在实验过程，我做的工作包含：创建顶层模块(TOP)，定义信号线，实例化各模块并连接它们，完成电路设计等等。其中涉及到指令存储器、程序计数器等模块的设计与实现。

在仿真验证中，我需要编写相关的激励文件，并进行系统任务的初始化和数据加载。通过添加时钟激励和其他输入信号，观察各个相关数值的变化情况，并与设计预期结果进行比对。

实验的最后阶段是上板调试，虽然课程对此不做硬性要求，但是我本着自主探索学习新知识的想法，我在课上尝试进行上板。但是由于软件与硬件的差异，相比仿真验证，上板时会出现意想不到的错误，难以发现，在经过5-6小时的代码调试后，成功完成了类MIPS单周期CPU的上板验证。

总的来说，通过本实验的实现和调试过程，我更深入地理解和掌握简单的类MIPS单周期处理器的设计与整体调试。

目录

1 实验目的

1. 理解简单的类 MIPS 单周期处理器的工作原理(即几类基本指令执行时所需的数据通路和与之对应的控制线路及其各功能部件间的互联定义、逻辑选择关系)
2. 完成简单的类 MIPS 单周期处理器
   (a) 9 条 MIPS 指令(lw, sw, beq, add, sub, and, or, slt, j) CPU 的实现与调试
   (b) 拓展至 16 条指令(增加 addi, andi, ori, sll, srl, jal, jr) CPU 的实设计与实现
3. 仿真测试
4. 上板验证

2 原理分析

2.1 MIPS单周期处理器部件

MIPS单周期处理器由多个部件组成，每个部件负责处理特定的功能，想要实现MIPS单周期处理器，首先需要实现处理器的各个部件。以下是一些关键的部件：

- 控制单元（Ctr）：负责生成处理器中各个部件的控制信号，以确保指令按照预期的方式执行。
- 算术逻辑单元（ALU）：执行算术和逻辑操作，例如加法、减法、与、或等。
- ALU控制单元（ALUCtr）：负责根据ALUOp和Funct生成控制ALU操作类型的信号，控制ALU进行相应算数操作。
- 数据存储器（Data memory）：用于存储数据的存储器。它根据地址提供数据，并接收写入请求。
- 指令存储器（Instruction memory）：用于存储指令的存储器。它根据程序计数器（PC）中的地址提供指令。
- 寄存器文件（Register file）：用于存储和读取寄存器的数据。它包含多个寄存器，每个寄存器都有唯一的编号。
- 程序计数器（PC）：用于存储当前执行指令的地址，并在每个时钟周期根据控制信号进行更新。

2.2 MIPS单周期处理器运行逻辑

MIPS单周期处理器的运行逻辑可以通过顶层模块来描述。顶层模块是处理器中各个部件的总体功能模块，它定义了各功能子模块之间的互联关系。

单周期处理器执行一条指令需要有以下步骤：

1. 根据PC的值从指令存储器中获取指令，并将其传递给控制单元进行解析。
2. 根据指令中的操作码和功能码生成相应的控制信号，以操纵寄存器文件、ALU、数据存储器等部件。
3. 寄存器文件根据指令中的寄存器编号进行数据的读取和写入。
4. ALU根据控制信号执行相应的算术和逻辑操作，并将结果传递给寄存器文件或数据存储器。
5. 数据存储器根据地址满足外界的读写请求。
6. 更新PC。

则在顶层模块中，我们需要将这些步骤转换为代码，定义功能部件实例化对象，将模块之间进行互联，实现MIPS单周期处理器的功能。

2.3 整体设计思路

MIPS单周期处理器的整体设计思路是基于指令的执行流程和数据通路的互联关系的。设计过程可以简化为以下几个步骤：

1. 根据MIPS指令集，确定所需实现的指令（实验指导书中已经给出），并分析每个指令的执行流程和所需的控制信号。
2. 设计和实现各个功能模块。由于前几个实验已经实现部分模块，可以直接导入本实验的工程文件中。在本实验中，需要额外添加模块有：Inst memory（指令内存）。
3. 根据指令的类型和执行流程，确定数据通路中各个部件之间的连接关系。
4. 根据控制信号的生成情况，编写控制单元的逻辑，以确保正确的控制信号传递给各个部件。
5. 对功能模块进行实例化，确保正确传递输入输出。

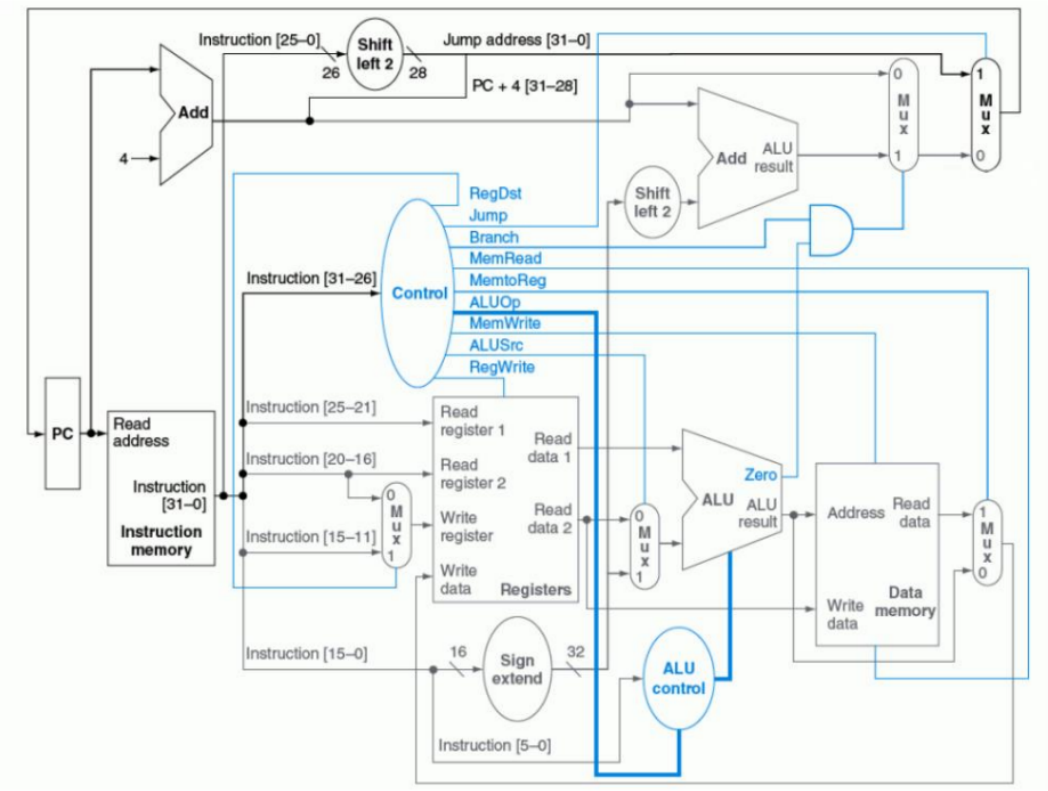

Figure 1: MIPS单周期处理器电路设计图

整体设计思路可以用一个MIPS单周期CPU工作原理图来概括，实验指导书中给出了一个清晰的CPU工作原理图，如图 figure 1。

2.4 时序逻辑设计

为了对各个部件进行数据同步，防止出错，设计中需要添加时序逻辑。对于MIPS单周期CPU，所需的时序逻辑并不复杂，我的MIPS单周期处理器的时序逻辑大致如下：

- 时钟上升沿：更新程序计数器（PC）的值
- 时钟下降沿：写Registers与Data memory。

其余操作均为组合逻辑。

3 功能实现

3.1 命名规则

由于各种变量或者信号名，我创建了一套简易的命名规则，方便代码的编写与阅读。规则如下：

- PC,ALU特殊名词大写
- 模块名称首字母小写，单词之间用大写字母分隔

- 模块的I/O变量名称首字母小写，单词之间用大写字母分隔
- TOP模块内部的变量名首字母大写，单词之间用大写字母分隔
- 模块实例化对象名称均小写，单词间用_分隔

3.2 各部件实现

根据原理分析，实现MIPS单周期处理器所需的模块包括：Ctr,ALUCtr,ALU,Data memory, registers,sign extend,Inst memory。其中大部分模块在lab01-lab04中已经实现，具体内容及原理在此不赘述。此处仅介绍之前未实现的Inst memory模块。

Inst memory模块的主要功能是接受PC输入，读取对应地址的指令并输出。因此实现较为简单，只需定义模块IO端口，定义指令内存空间，进行指令读取即可。代码实现如下：

```
23  module instMemory(
24          input PC_Clk,
25          input [31:0] address,
26          output [31:0] readData
27      );
28      reg [31:0] memFile[0:31];
29      integer i;
30      initial begin
31          for(i = 11; i <= 31; i = i + 1)
32              memFile[i] = 0;
33      end
34      assign readData = memFile[(address >> 2)];
35  endmodule
```

Figure 2: instMemory模块实现

需要注意的是，PC的位置是按字节寻址，而指令内存是字（4-bytes）对齐的，所以读取`memFile`时需要将`PC`右移两位作为地址索引。

3.3 顶层模块实现

顶层模块在`Top.v`文件中实现，代码主要分为六部分：指令读取，控制信号生成，寄存器读写，ALU操作，内存读写，更新PC。

1. **指令读取部分实现**。定义好必要变量，将时钟信息和PC传入作为输入，将`Inst`传入作为输出即可。核心代码如下：

```
1   wire [31:0] Inst;
2   instMemory inst_memory(
3       .PC_Clk(Clk),
4       .address(PC),
5       .readData(Inst)
6   );
```

2. **控制信号生成部分**。控制信号由两个模块生成，分别是Ctr模块与ALUCtr模块，针对Ctr模块，根据原理图定义好变量，将Ctr模块使用适当的输入与输出实例化即可，控制信号定义如下：

```
    wire RegDst,ALUSrc,MemToReg,RegWrite,MemRead;
    wire MemWrite,Branch,Jump,SignExtFlag,JalFlag;
    wire [2:0] ALUOp;
```

定义好变量后，实例化Ctr模块即可，实例化代码如下：

```
    Ctr ctr (
        .opCode(Inst[31:26]),
        .regDst(RegDst),
        .aluSrc(ALUSrc),
        .memToReg(MemToReg),
        .regWrite(RegWrite),
        .memRead(MemRead),
        .memWrite(MemWrite),
        .branch(Branch),
        .aluOp(ALUOp),
        .jump(Jump),
        .signExtFlag(SignExtFlag),
        .jalFlag(JalFlag)
    );
```

同样，对于ALUCtr模块，首先定义必要变量再进行实例化即可，主要代码如下：

```
    ALUCtr alu_ctr(
        .ALUOp(ALUOp),
        .Funct(Inst[5:0]),
        .jrFlag(JrFlag),
        .ALUCtrOut(ALUCtrOut)
    );
```

至此，主要控制信号便已经处理完毕，能够由Ctr与ALUCtr模块给出。除此之外，对于Branch，其由Ctr模块产生，若为Branch相关指令则Branch=1，同时，将其与Zero的值结合，生成是否跳转的控制信号，代码如下：

```
    assign BranchTaken = (Branch & Zero);
```

3. **寄存器读写部分**。回忆MIPS单周期处理器原理图 figure 1，针对寄存器模块定义变量如下：

```
    wire [31:0] RegWriteData,RegReadData1,RegReadData2;
```

实例化部分如下：

```
    Registers registers(
        .reset(Reset),
        .Clk(Clk),
        .readReg1(Inst[25:21]),
```

```
        .readReg2(Inst[20:16]),
        .writeReg(WriteReg),
        .writeData(RegWriteData),
        .regWrite(RegWrite ^ JrFlag),
        .jalFlag(JalFlag),
        .readData1(RegReadData1),
        .readData2(RegReadData2)
    );
```

对于不同指令的WriteReg（目标写入寄存器），它们在指令中所处的位置是不同的，所以我们需要利用多路选择器mux区分目标寄存器，我并未独立定义mux模块，而是用了三目表达式代替mux模块进行代码编写，如下：

```
    assign WriteReg = RegDst ? Inst[15:11] : Inst[20:16];
```

针对写入数据（RegWriteData），我利用根据控制信号MemToReg对写入数据进行选择：

```
    assign RegWriteData = MemToReg ? MemReadData : ALURes;
```

4. **ALU操作**。ALU模块的功能便是根据控制信号ALUCtrOut对两个输入数据执行相应操作。在顶层模块中，我们需要定义两个输入数据，控制信号，输出的结果以及Zero信号。定义如下：

```
    wire [31:0] ALUSrc1,ALUSrc2;
    wire [3:0] ALUCtrOut;
    wire Zero;
    wire [31:0] ALURes;
```

ALU模块实例化如下：

```
    ALU alu(
        .input1(ALUSrc1),
        .input2(ALUSrc2),
        .aluCtr(ALUCtrOut),
        .zero(Zero),
        .aluRes(ALURes)
    );
```

对于ALU的两个输入数据，ALUSrc1即为ReadData1，而ALUSrc2则需要一个mux进行选择，对于不同指令ALUSrc2的来源会有所不同，它们的赋值代码如下：

```
    assign ALUSrc1 = RegReadData1;
    assign ALUSrc2 = ALUSrc ? Ext : RegReadData2;
```

5. **内存读写**。内存模块逻辑基本与寄存器模块一致，不过在变量命名以及读取数据个数等方面略有不同。为了在顶层模块中接入内存读写模块，首先定义变量并赋值如下：

```
        wire [31:0] MemAddress;
        wire [31:0] MemWriteData;
```

```
        wire [31:0] MemReadData;
        assign MemAddress = ALURes;
        assign MemWriteData = RegReadData2;
```

接着利用这些变量将dataMemory实例化即可，实例化代码如下：

```
    dataMemory data_memory(
        .Clk(Clk),
        .address(MemAddress),
        .writeData(MemWriteData),
        .memRead(MemRead),
        .memWrite(MemWrite),
        .readData(MemReadData)
    );
```

6. **PC更新**。首先定义PC，再通过always语句结合时序信息对PC进行更新。具体来说，在CLk的上升沿，进入always块中对PC进行更新，如果Reset信号为高电平，则PC清空，否则，如果为跳转指令则将PC赋值为跳转目的地的地址，为BranchTaken则进行分支跳转，为Jr指令则将ALU运算结果赋值给PC，否则PC直接加4即可，不用进行其他额外操作。

```
    always @ (posedge Clk)
    begin
        if(Reset)
        begin
            PC = 0;
        end
        else
        begin
            PC = PC + 4;
            if (Jump == 1)//jmp , jal
            begin
                PC = {PC[31:28],(Inst[25:0] << 2)};
            end
            else if (BranchTaken == 1)
            begin
                PC = PC + (Ext << 2);
            end
            else if (JrFlag == 1)
            begin
                PC = ALURes;
            end
        end
    end
```

3.4 指令扩展实现

成功实现支持9指令的MIPS单周期处理器后，我继续将其拓展为支持16条指令的MIPS单周期处理器。从支持 (lw, sw, beq, add, sub, and, or, slt, j) 扩展为支持 (lw, sw, beq,

add，sub，and，or，slt，j，addi，andi，ori，sll，srl，jal，jr)。为了实现这种扩展，我需要在原有MIPS单周期处理器的基础上对相应部分进行更改。

1. **扩展R type指令（sll，srl）**

   首先我们需要明确指令的具体内容，如下图：

sll	000000	00000	rt	rd	shamt	000000	sll $1,$2,10	$1=$2<<10	rd <- rt << shamt ； shamt存放移位的位数，也就是指令中的立即数，其中rt=$2, rd=$1
srl	000000	00000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	rd <- rt >> shamt ； (logical)，其中rt=$2, rd=$1

Figure 3: sll,srl指令

   这两条指令都是R型指令，需要用到rt,rd与shamt字段，将结果写入寄存器rd中，所以我们需要修改的模块有：

   - ALUCtr：针对这两种指令生成移位控制信号输出给ALU。

```
        9'b010_000000 :
        begin
            ALUCtrOut = 4'b1000;//sll
            jrFlag = 0;
        end
        9'b010_000010 :
        begin
            ALUCtrOut = 4'b1001;//srl
            jrFlag = 0;
        end
```

   - ALU：新增两种移位运算，逻辑左移与逻辑右移。

```
        4'b1000://sll
        begin
            aluRes = input2 << shamt;
        end
        4'b1001://srl
        begin
            aluRes = input2 >> shamt;
        end
```

2. **扩展I type指令（addi,andi,ori）** 三条指令内容如下：

I-type	op	rs	rt	immediate			
addi	001000	rs	rt	immediate	addi $1,$2,100	$1=$2+100	rt <- rs + (sign-extend)immediate ； 其中rt=$1,rs=$2

Figure 4: addi指令

andi	001100	rs	rt	immediate	andi $1,$2,10	$1=$2 & 10	rt <- rs & (zero-extend)immediate ; 其中 rt=$1,rs=$2
ori	001101	rs	rt	immediate	andi $1,$2,10	$1=$2 | 10	rt <- rs | (zero-extend)immediate ; 其中 rt=$1,rs=$2

Figure 5: andi、ori指令

这三条指令都是I型指令，它们均对寄存器以及立即数进行运算，写入寄存器Rt，同时andi与ori需要零扩展。所以我们需要修改的模块有

- sign extend：需要添加一个符号区分扩展是符号扩展还是无符号扩展（零扩展）。

```
module signext(
        input  [15:0] inst,
        input  signExtFlag,
        output [31:0] data
    );
    assign data = signExtFlag ? {{16{inst[15]}}, inst} :
        {{16{1'b0}}, inst};
endmodule
```

- Ctr：需要针对这几条指令的opCode，在case语句添加对应情况的控制信号赋值。

```
6'b001000://addi
begin
    regDst = 0;
    aluSrc = 1;
    memToReg = 0;
    regWrite = 1;
    memRead = 0;
    memWrite = 0;
    branch = 0;
    aluOp = 3'b011;
    jump = 0;
    signExtFlag = 1;
end
//... andi ori ...
```

- ALUCtr：针对这几条指令的ALUOp，输出对应的ALU的控制信号。

```
9'b011xxxxxx :
begin
    ALUCtrOut = 4'b0010;//addi
    jrFlag = 0;
end
9'b100xxxxxx :
begin
    ALUCtrOut = 4'b0000;//andi
    jrFlag = 0;
```

```
        end
        9'b101xxxxxx :
        begin
            ALUCtrOut = 4'b0001;//ori
            jrFlag = 0;
        end
```

3. **扩展jr指令** jr指令内容如下：

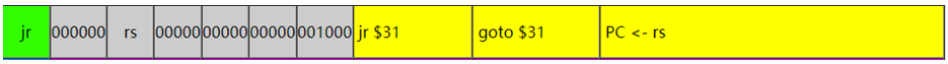

Figure 6: jr指令

jr指令是一条特殊的R型指令，作用是将$31寄存器的内容写入给PC。由于这是一个R型指令，所以控制型号会认为它将要写寄存器，且写入的是0号寄存器，但是0号寄存器应该始终保持为0，所以我们需要对jr指令进行特殊处理，具体需要修改的模块有：

- PC更新模块：添加一个jrFlag后，在PC更新的模块中使用if/else语句将jr指令对应的操作加入。

```
        else if (JrFlag == 1)
        begin
            PC = ALURes;
        end
```

- ALUCtr：对应jr指令的ALUOp与Funct，生成一个控制ALU加法操作的信号即可，（31寄存器与0寄存器加，由于0寄存器始终为0，故值还是31寄存器中的值）。

```
        9'b010_001000 :
        begin
            ALUCtrOut = 4'b0010;//jr
            jrFlag = 1;
        end
```

- Registers：当指令为jr时应该特殊判断，不能进行寄存器写，因为$0寄存器的值不能被改变。

4. **扩展jal指令** jal指令内容如下：

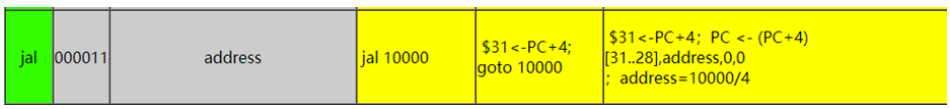

Figure 7: jal指令

jal指令是将原PC + 4的值存入31号寄存器之后进行跳转，需要添加一个写寄存器的控制信号，单独用于控制写Registers。具体来说，需要修改的模块如下：

- Ctr：添加jalFlag输出，当指令为jal时赋值为1，同时针对jal的opCode在case中加入对应情况的控制信号赋值。

```
        6'b000011://jal
        begin
            regDst = 0;
            aluSrc = 0;
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 0;
            branch = 0;
            aluOp = 3'b000;
            jump = 1;
            signExtFlag = 1;
        end
```

- Registers：添加jalFlag作为输入，当其为1时，应该对31号寄存器进行写操作。

3.5 激励文件编写

在完成MIPS单周期处理器代码编写后，为了进行后续的仿真测试，我编写了如下激励文件。

```
module Top_tb(
    );
    reg Clk;
    reg reset;
    Top SingleCycleCPU(
        .Clk(Clk),
        .Reset(reset)
    );
    always #50 Clk = ~Clk;
    initial begin
        Clk = 1;
        reset = 1;
        # 25;
        reset = 0;
    end
endmodule
```

3.6 上板验证代码编写

通过重新阅读lab02的实验指导书，我回顾了上板的操作，计划利用8个switch作为输入（4个switch作为输入1，另外4个作为输入2），运行指令最终将输入1与输入2相乘的结果输出在七段数码管上。我将两个输入分别赋值给memFile[0],memFile[1]，并将memFile[2]处的值赋给输出，这样便能够进行动态输入计算。主体修改部分如下

```
module Top(
    input clk_p,
    input clk_n,
    input [3:0] a,
    input [3:0] b,
    input Reset,

    output led_clk,
    output led_do,
    output led_en,
    output seg_clk,
    output seg_en,
    output seg_do
);
wire [31:0] Op1;
assign Op1 = {28'b0,a};
wire [31:0] Op2;
assign Op2 = {28'b0,b};
wire [31:0] Out;
```

其余代码为利用差分时钟对输入时钟进行调频，使其适合于我的处理器，同时定义display模块。display定义如下：

```
display DISPLAY(
    .clk(Clk_25M),
    .rst(1'b0),
    .en(8'b11111111),
    .data({Op1,4'b0 ,Op2, 4'b0,Out[15:0]}),
    .dot(8'b00000000),
    .led(~Out[15:0]),
    .led_clk(led_clk),
    .led_en(led_en),
    .led_do(led_do),
    .seg_clk(seg_clk),
    .seg_en(seg_en),
    .seg_do(seg_do)
);
```

可以看到我将两个输入以及运算结果显示在了七段数码管上，方便后续上板验证。此外我们还需要编写管脚约束文件，这个部分与lab02的内容相差不大，代码过于长在此暂且不展示，简单来说就是对`module Top`的输入输出端口进行约束连线，写在名为`lab05_xdc.xdc`文件中。

3.7 整体调试

在完成主体部分后，经过仿真发现并未得到预期结果，通过加入信号进入波形图，并进行波形图分析逐步debug后，成功发现错误并改正。错误是开始将ALUOp定义成错误定义成了1位的线网类型，但ALUOp是2位线网类型。故做出如下修改：

```
wire ALUOp;  change into  wire [1:0] ALUOp;
```

错误改正后，我编写的MIPS单周期处理器便能够正常运行仿真并给出结果。

4 结果验证

4.1 仿真验证

编写好design sources文件与激励文件后，点击run behavior simulation进行仿真验证。使用系统任务`$readmemb,$readmemh`在Top的initial块中对inst memory以及data memory进行初始化，代码如下：

```
initial begin
    PC = 0;
    $readmemb("E:/Archlab/lab05/lab05.srcs/instruction.txt",
                                        inst_memory.memFile);
    $readmemh("E:/Archlab/lab05/lab05.srcs/data.txt",data_memory.memFile);
end
```

运行的指令是由课程组提供的测试用例（一个简单的乘法循环程序，初始化时，所有Registers被置为0，memFile第一个字是9，第二个字是13，结果存在第3个字中。预期最后结果是117）得到如下波形图：

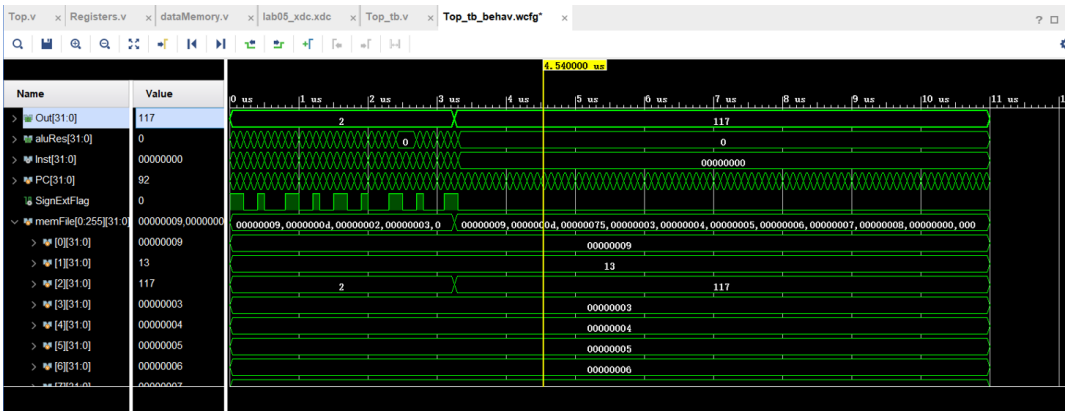

Figure 8: 仿真验证波形图

通过波形图可以看到，在完成若干个周期的操作后，我的MIPS单周期处理器将运算结果117放到了memFile[2]中，符合预期，由此可见，我的MIPS单周期处理器能够正常运行且运行结果正确。

4.2 上板验证

Generate bitstream后对FPGA板进行烧写，经验证，我成功实现了动态输入下的乘法程序，在9和13的输入下七段数码管输出117，结果正确。至此，我测试了我的MIPS单周期处理器在执行实际有意义的汇编程序时，能够很好地与硬件结合，得到正确结果，这也进一步证明了我的MIPS单周期处理器是正确的。

5 困难的思考与解决

我在本实验中遇到最大的困难是**上板验证**。编写好上板验证的代码后，我进行二进制流文件的生成与FPGA板的烧写，发现七段数码管的输出始终为0，但是仿真程序的结果完全正确。

针对这个问题，我对代码进行了详细地检查，没有发现任何错误，所以我只能通过在七段数码管上输出debug信息的方式逐步排除错误，但是由于二进制流文件生成的时间过长，导致调试异常缓慢。经过5-6个小时的尝试后，我发现是指令并没有读入到inst memory中，导致程序无法正常执行。针对这个问题，我上网查阅了相关资料，发现在上班验证时不能够通过系统任务`$readmemb,$readmemh`对数据进行读取，所以，我直接将指令在initial块中赋值给对应的inst memory单元。代码如下：

```
1    memFile[0] = 32'b10001100011000010000000000000000;  //lw $1, 0($3)
2    memFile[1] = 32'b10001100011000010000000000000100;  //lw $2, 4($3)
3    ...
```

经过这样的修改后，再次上板验证，我得到了正确的结果。

当然除此之外，在实验中还遇到了其他困难。在遇到困难后，通过我的耐心调试与不懈的努力，我最终克服了这些困难。

6 总结与反思

通过完成这个实验，我学到了许多关于MIPS单周期处理器的原理和设计方面的知识。首先，在进行原理分析时，我深入了解了MIPS单周期处理器的各个部件和整体的运行逻辑。这让我对处理器的结构和工作原理有了更清晰的理解。

在整体设计思路的探讨中，我学会了如何将各个部件有机地组合起来，形成一个完整的处理器系统。这需要考虑数据通路的设计、指令执行的顺序和流程控制等因素。通过整体设计思路的学习，我对处理器的整体架构和设计方法有了更深入的了解。

整体的时序逻辑设计是我在这个实验中接触到的新概念，我了解了如何确保各个部件之间的数据同步和正确的时序操作。时序逻辑设计的正确实现对于处理器的稳定性和可靠性至关重要。

在功能实现的过程中，想要实现不同的部件和模块需要对MIPS指令集有一定的了解，并将其转化为硬件设计。我也强化了我整体调试的能力，通过正确调试，确保处理器的各个功能正常工作。

在结果验证阶段，我学习了如何使用仿真工具来验证处理器的正确性，并通过上板验证来验证其在实际硬件中的工作情况。这一步骤对于确保处理器的正确性和可靠性非常重要。

整个实验中，我也遇到了一些困难，但通过深入思考和寻找解决方案，我成功地克服了这些困难。这让我意识到在实际工程中遇到问题是很正常的，重要的是保持积极的态度和找到解决问题的方法。

通过本次实验，我不仅学到了有关MIPS单周期处理器的理论知识，还锻炼了自己的问题解决能力和工程实践能力。这对我的学习和未来的职业发展都具有重要意义，我相信这些知识和经验能够让我受用终生。

最后，感谢老师以及助教的课上答疑以及课程组准备的指导书。