# 计算机系统结构试验

# Lab05: 简单的类 MIPS 单周期处理器的实现

姓名：N/A

## 摘要

在 Lab05 中，我实现了 MIPS 单周期处理器的主要功能，支持 31 条指令的译码、执行功能，并在此基础之上实现了拨码乘法器。实验包括了 Program Counter、Branch Control、Immediate Extension 等功能的设计和实现，以及综合各部件的 Top 模块。通过本次实验，我进一步加深了对 Verilog 语言的理解和运用，掌握 MIPS 处理器的组成、设计和实现方法，并且能够使用仿真工具进行验证和调试，给我带来宝贵的经验和收获。

## 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；

1）9 条 MIPS 指令（lw, sw, beq, add, sub, and, or, slt, j）CPU 的实现与调试。

2）拓展至 16 条指令（增加 addi, andi, ori, sll, srl, jal, jr）CPU 的实现与调试。

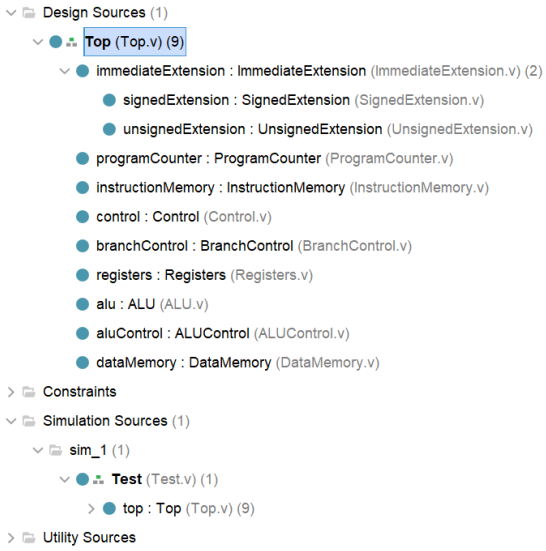
（3）乘法代码的仿真测试；

（4）上板验证拨码乘法器。

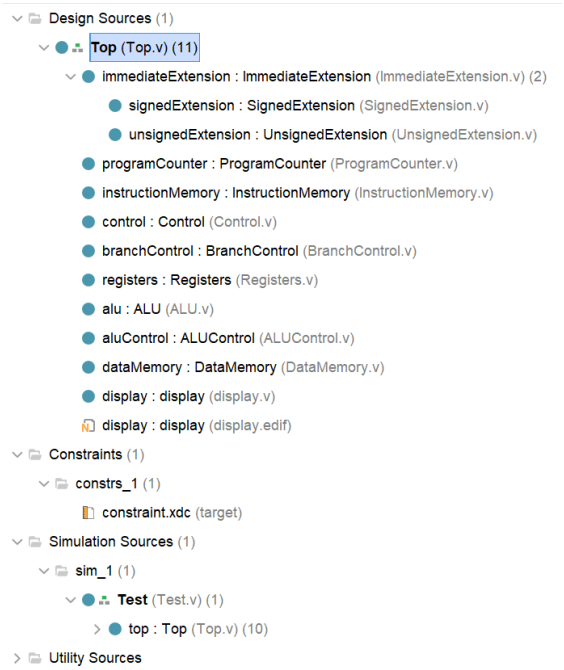
# 2. 原理分析

## 2.1 Vivado 工程的基本组成

仿真测试的 Vivado 工程的结构如下：




拨码乘法器的 Vivado 工程的结构如下：



### 2.2 ProgramCounter 模块的原理

Program Counter (PC)模块是用于跟踪程序执行位置的关键组件，PC 模块会保存当前正在执行的指令的内存地址。每执行完一条指令，PC 会自动增加 4 字节，指向下一条要执行的指令。

当遇到跳转指令(如 jal、jalr、beq 等)时,PC 会根据跳转地址更新自己的值，使程序跳转到目标位置继续执行。跳转地址由指令中的立即数解码得到，或者从 31 号寄存器 ra 读取。因此 PC 模块需要接收指令立即数、31 号寄存器以及各跳转控制信号的输入。

### 2.3 InstructionMemory 模块和 DataMemory 模块的原理

InstructionMemory 模块将指令内存中相应地址的数据读出，需注意 PC 按字节编址，而 Memory 按 4 字节编址，因此需要将输入 PC 右移 2 位。DataMemory 在此基础之上还有写入功能，当接收到写使能信号时将读入数据写入相应位置。

### 2.4 ImmediateExtension 模块的原理

ImmediateExtension 模块将 I-type 指令中的立即数扩展到 32 位，由于指令的不同，立即数有无符号数和有符号数两种，因此分别实现了 UnsignedExtension 和 SignedExtension 两个子模块。模块通过指令的 opCode 判断扩展方式并输出相应结果。

### 2.5 Control 模块，ALUControl 模块和 BranchControl 模块的原理

Control 模块进行指令的译码，R-type 指令含义与 funct 字段有关，转发给 ALUControl 进行进一步判断，其余指令设置相应的 regDst，ALUSrc，memToReg，regWrite，memRead，memWrite 和 ALUOp 控制信号。

ALUControl 模块根据 ALUOp 和 funct 字段译码，得到 ALU 控制信号，控制 ALU 的计算行为。ALUOp 和 ALU 控制信号可以自行编写。

BranchControl 模块使用 opCode 和 funct 字段判断当前指令是否属于跳转指令，并设置相应的控制信号位。

### 2.6 Registers 模块的原理

Registers 模块类似于 DataMemory 模块，但有两个地址输入和两个寄存器值输出。需要注意 0 号寄存器应当保持值为 0 不变，因此要忽略对 0 号寄存器的写入。

### 2.7 ALU 模块的原理

ALU 模块根据 ALUControl 模块约定的 ALUCtr 信号对两个输入进行运算，运算类型包括 and，or，xor，nor，add，sub，slt，sltu，sll，srl，sra，sllv，srlv，srav，lui。由于 sll，srl，sra 指令需要读取 shamt 字段，ALU 模块也需要一个来自指令的指令。

# 3. 功能实现

## 3.1 ProgramCounter 模块的实现

Program Counter (PC)模块接收三种跳转的立即数：j 和 jal 指令接收 Instruction[25:0]，beq 和 bne 指令接收有符号扩展的 Instruction[15:0]，jr 指令接收 31 号寄存器 ra 的值。

模块首先对 jumpAddr 和 branchAddr 进行译码。因为指令地址均为 4 的倍数，jumpImm 首先要左移 2 位，jump 指令会保留 PC + 4 的最高 4 位，因此这里加上 PC + 4 的最高四位。branchImm 同理左移 2 位，由于 branch 指令的立即数表示 offset，因此加上 PC + 4。

在时序部分，通过输入的控制信号以及 ALU 的 zero flag 进行相应的地址跳转，否则 PC = PC + 4。

```
module ProgramCounter (
    input wire clk,
    input wire reset,

    input wire [25:0] jumpImm,
    input wire [31:0] branchImm,
    input wire [31:0] ra,

    input wire beq,
    input wire bne,
    input wire jr,
    input wire j,
    input wire jal,
    input wire ALUZero,

    output reg [31:0] programCounter
);
```

```
  wire [31:0] jumpAddr = (jumpImm << 2) + ((programCounter + 4) & 32'hf0000000);
  wire [31:0] branchAddr = (branchImm << 2) + programCounter + 4;

  always @(posedge clk or posedge reset) begin
    if (reset) begin
      programCounter = 32'h00000000;
    end else if (beq && ALUZero) begin
      programCounter = branchAddr;
    end else if (bne && !ALUZero) begin
      programCounter = branchAddr;
    end else if (jr) begin
      programCounter = ra;
    end else if (j) begin
      programCounter = jumpAddr;
    end else if (jal) begin
      programCounter = jumpAddr;
    end else begin
      programCounter = programCounter + 4;
    end
  end
```

## 3.2 InstructionMemory 模块和 DataMemory 模块的实现

InstructionMemory 模块较简单，初始化时读入 Instruction.txt 的内容，代码如下：

```
module InstructionMemory (
    input clk,
    input reset,
    input [31:0] address,

    output reg [31:0] readData
);

  reg [31:0] memory[0:63];

  always @(*) begin
    if (reset) begin
      readData = memory[0];
    end else begin
      readData = memory[address>>2];
    end
  end

  initial begin
    $readmemb("Instruction.txt", memory);
  end

endmodule
```

DataMemory 模块需要注意 hazard 问题，读数据和其他部件不能在同一时钟沿，否则地址和数据将在同一时钟沿更新。此外，当 read 和 write 同时使能时，选择直接将写入数据 forward 到输出。代码如下：

```
module DataMemory (
    input clk,
    input [31:0] address,
    input [31:0] writeData,
    input memWrite,
    input memRead,

    output reg [31:0] readData
);

  reg [31:0] memory[0:63];

  always @(*) begin
    if (memRead) begin
      if (memWrite) begin
        readData = writeData;
      end else begin
        readData = memory[address>>2];
      end
    end else begin
      readData = 0;
    end
  end

  always @(posedge clk) begin
    if (memWrite) begin
      memory[address>>2] = writeData;
    end
  end

  initial begin
    $readmemh("Data.txt", memory);
  end

endmodule
```

3.3 ImmediateExtension 模块的实现

ImmediateExtension 模块接收 opCode 的输入，判断指令中立即数应做无符号扩展还是有符号扩展。在所有指令中，addiu、andi、ori、xori 和 sltiu 指令需要无符号扩展，其余指令均为有符号扩展，代码实现如下：

```
module ImmediateExtension (
    input wire [15:0] source,
    input wire [ 5:0] opCode,

    output reg [31:0] extension
);

  wire [31:0] signedExt;
  wire [31:0] unsignedExt;

  SignedExtension signedExtension (
      .source(source),
      .extension(signedExt)
  );

  UnsignedExtension unsignedExtension (
      .source(source),
      .extension(unsignedExt)
  );

  always @(*) begin
    case (opCode)
      6'b001001: extension = unsignedExt;  // addiu
      6'b001100: extension = unsignedExt;  // andi
      6'b001101: extension = unsignedExt;  // ori
      6'b001110: extension = unsignedExt;  // xori
      6'b001011: extension = unsignedExt;  // sltiu
      default:   extension = signedExt;
    endcase
  end

endmodule
```

```verilog
module SignedExtension (
    input [15:0] source,
    output reg [31:0] extension
);

  always @(*) begin
    if (source[15] == 1) begin
      extension = {16'b1111111111111111, source};
    end else begin
      extension = {16'b0000000000000000, source};
    end
  end

endmodule
```

```verilog
module UnsignedExtension (
    input [15:0] source,
    output reg [31:0] extension
);

  always @(*) begin
    extension = {16'b0000000000000000, source};
  end

endmodule
```

3.4 Control 模块，ALUControl 模块和 BranchControl 模块的实现

Control 模块根据输入指令的 opCode 输出 regDst、ALUSrc、memToReg、regWrite、memRead、memWrite 控制信号以及 ALUOp。ALUControl 模块对于非 R-type 指令根据 ALUOp 选取 ALUCtr，对于 R-type 指令根据 funct 字段选取 ALUCtr。非 R-type 指令的 ALUOp 可以自行选取，只要和 Control 模块保持一致即可。Control 模块代码此处省略，ALUControl 模块代码如下：

```verilog
module ALUControl (
    input [3:0] ALUOp,
    input [5:0] funct,

    output reg [3:0] out
);

  always @(ALUOp, funct) begin
    case (ALUOp)
      4'b0000: begin  // R-type
        casex (funct)
          6'b10000x: out = 4'b0100;  // add addu
          6'b10001x: out = 4'b0101;  // sub subu
          6'b100100: out = 4'b0000;  // and
          6'b100101: out = 4'b0001;  // or
          6'b100110: out = 4'b0010;  // xor
          6'b100111: out = 4'b0011;  // nor
          6'b101010: out = 4'b0110;  // slt
          6'b101011: out = 4'b0111;  // sltu
          6'b000000: out = 4'b1000;  // sll
          6'b000010: out = 4'b1001;  // srl
          6'b000011: out = 4'b1010;  // sra
          6'b000100: out = 4'b1011;  // sllv
          6'b000110: out = 4'b1100;  // srlv
          6'b000111: out = 4'b1101;  // srav
          default:   out = 4'b1111;
        endcase
      end

      4'b0001: out = 4'b0100;  // addi addiu lw sw
      4'b0010: out = 4'b0000;  // andi
      4'b0011: out = 4'b0001;  // ori
      4'b0100: out = 4'b0010;  // xori
      4'b0101: out = 4'b1110;  // lui
      4'b0110: out = 4'b0101;  // beq bne
      4'b0111: out = 4'b0110;  // slti
      4'b1000: out = 4'b0111;  // sltiu
      default: out = 4'b1111;
    endcase
  end
```

BranchControl 模块根据指令的 opCode 和 funct 字段判断跳转指令类型，代码如下：

```
module BranchControl (
    input wire [5:0] opCode,
    input wire [5:0] funct,

    output reg beq,
    output reg bne,
    output reg jr,
    output reg j,
    output reg jal
);

  always @(opCode, funct) begin
    beq = (opCode == 6'b000100) ? 1'b1 : 1'b0;
    bne = (opCode == 6'b000101) ? 1'b1 : 1'b0;
    jr = (opCode == 6'b000000 && funct == 6'b001000) ? 1'b1 : 1'b0;
    j = (opCode == 6'b000010) ? 1'b1 : 1'b0;
    jal = (opCode == 6'b000011) ? 1'b1 : 1'b0;
  end

endmodule
```

3.5 Registers 模块的实现

由于不确定 writeReg，writeData，regWrite 信号的先后次序，采用时钟的下降沿作为写操作的同步信号，防止发生错误。因此模块逻辑为：当 readReg1，readReg2，writeReg 其中一个发生变化时，将寄存器内值读到输出；当遇到 clk 下降沿时，若 regWrite 使能，则将输入写入寄存器。代码如下：

```
module Registers (
    input clk,
    input [25:21] readReg1,
    input [20:16] readReg2,
    input [4:0] writeReg,
    input [31:0] writeData,
    input regWrite,

    output reg [31:0] readData1,
    output reg [31:0] readData2
);

  reg [31:0] regFile[31:0];

  always @(readReg1, readReg2, writeReg) begin
    readData1 = regFile[readReg1];
    readData2 = regFile[readReg2];
  end

  always @(negedge clk) begin
    if (regWrite && writeReg != 0) begin
      regFile[writeReg] = writeData;
    end
  end

  initial begin
    $readmemh("Register.txt", regFile);
  end

endmodule
```

3.6 ALU 模块的实现

ALU 模块根据 ALUCtr 的控制信号进行相应运算，ALUCtr 信号也可自行约定。需要注意不同指令的有符号数版本与无符号数版本的区别。代码如下：

```
module ALU (
    input [31:0] input1,
    input [31:0] input2,
    input [4:0] shamt,
    input [3:0] ALUCtr,
    output reg zero,
    output reg [31:0] result
);

  always @(input1, input2, ALUCtr, shamt) begin
    case (ALUCtr)
      4'b0000: result = input1 & input2;  // and
      4'b0001: result = input1 | input2;  // or
      4'b0010: result = input1 ^ input2;  // xor
      4'b0011: result = ~(input1 | input2);  // nor

      4'b0100: result = input1 + input2;  // add
      4'b0101: result = input1 - input2;  // sub

      4'b0110: result = ($signed(input1) < $signed(input2)) ? 32'b1 : 32'b0;  // slt
      4'b0111: result = (input1 < input2) ? 32'b1 : 32'b0;  // sltu

      4'b1000: result = input2 << shamt;  // sll
      4'b1001: result = input2 >> shamt;  // srl
      4'b1010: result = $signed(input2) >>> shamt;  // sra
      4'b1011: result = input2 << input1;  // sllv
      4'b1100: result = input2 >> input1;  // srlv
      4'b1101: result = $signed(input2) >>> input1;  // srav

      4'b1110: result = input2 << 16;  // lui

      default: result = 32'b0; // nop
    endcase
    zero = (result == 32'b0) ? 1'b1 : 1'b0;
  end

endmodule
```

3.7 Top 模块的实现

Top 模块连接上述模块，根据 Control 模块的控制信号输出选择相应的输入。部分代码如下：

```
Registers registers (
    .clk(clk),
    .readReg1(instruction[25:21]),
    .readReg2(instruction[20:16]),
    .writeReg(jal ? 5'd31 : (regDst ? instruction[15:11] : instruction[20:16])),
    .writeData(jal ? PC + 4 : (memToReg ? memReadData : ALUResult)),
    .regWrite(regWrite),
    .readData1(regReadData1),
    .readData2(regReadData2)
);

ALU alu (
    .input1(regReadData1),
    .input2(ALUSrc ? immediate : regReadData2),
    .shamt (instruction[10:6]),
    .ALUCtr(ALUControlCode),
    .zero  (ALUZeroFlag),
    .result(ALUResult)
);
```

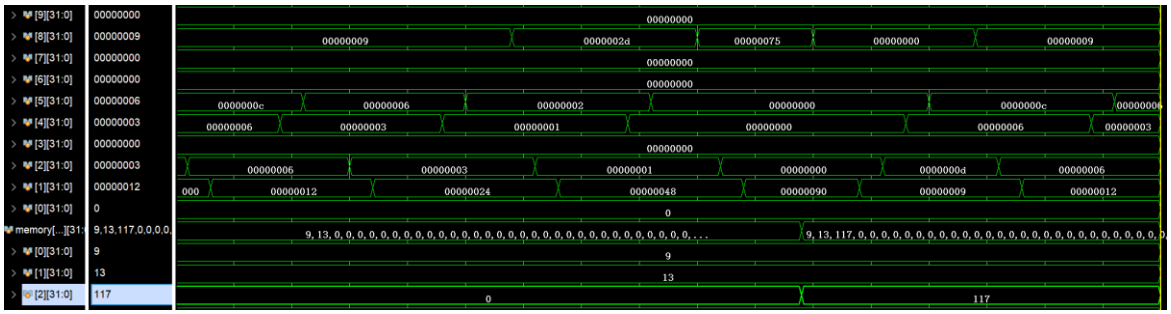
# 4. 结果验证

4.1 乘法代码的测试

编写 Instruction.txt 文件如下：

```
100011_00011_00001_0000000000000000   // lw $1, 0($3)
100011_00011_00010_0000000000000100   // lw $2, 4($3)
000000_00000_00010_00100_00001_000010 // srl $4, $2, 1
000000_00000_00100_00101_00001_000000 // sll $5, $4, 1
000100_00010_00101_0000000000000001   // beq $5, $2, 1
000000_01000_00001_01000_00000_100000 // add $8, $8, $1
000000_00000_00010_00010_00001_000010 // srl $2, $2, 1
000000_00000_00001_00001_00001_000000 // sll $1, $1, 1
000100_00011_00010_0000000000000001   // beq $2, $3, 1
000010_00000000000000000000000010     // j 2
101011_00011_01000_0000000000001000   // sw $8, 8($0)
000000_00000_00000_01000_00000_100000 // add $8, $0, $0
000010_00000000000000000000000000     // j 0
```

测试结果如图所示：




输出和计算过程符合预期。

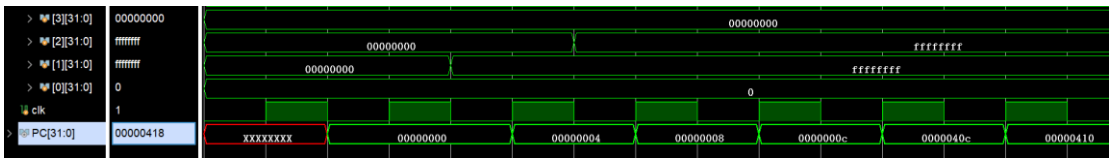
## 4.2 其他指令的测试

### 4.2.1 beq，bne 指令

编写 Instruction.txt 文件如下：

```
1  001000_00000_00001_1111111111111111 // addi $1, $0, 0xffff
2  001000_00000_00010_1111111111111111 // addi $2, $0, 0xffff
3  000101_00001_00010_0000000000000000 // bne $1, $2, 0
4  000100_00001_00010_0000000011111111 // beq $1, $2, 0xff
```

测试结果如图所示：



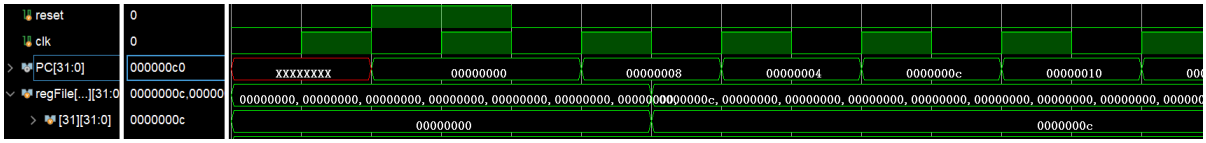

测试代码将 1 号、2 号寄存器设为 0xffff。在 bne 指令中，分支条件不满足，因此 PC 由 0x08 增加到 0x0c；在 beq 指令中，分支条件满足，PC = PC + 0x4 + 0xff << 2 = 0x40C。仿真符合预期。

### 4.2.2 j，jr，jal 指令

编写 Instruction.txt 文件如下：

```
1  000010_00000000000000000000000010 // j 2
2  000000_11111_00000_00000_00000_001000 // jr $31
3  000011_00000000000000000000000001 // jal 1
```

测试结果如图所示：



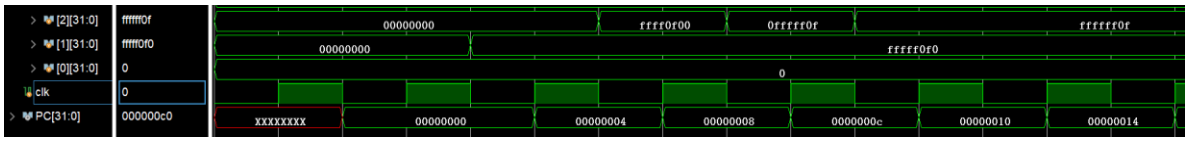

测试代码先使用 j 2 无条件跳转到 PC = 2 << 2 = 8 处，此时执行第三条指令 jal 1，将 PC + 4 = 0xc 存入 ra 寄存器，并跳转到 PC = 1 << 2 = 4 处。接下来执行第二条指令 jr $31 将 PC 设为 ra 的值 0xc。仿真符合预期。

### 4.2.3 sll，srl，sra 指令

编写 Instruction.txt 文件如下：

```
001000_00000_00001_1111000011110000   // addi $1, $0, 0xf0f0
000000_00000_00001_00010_00100_000000 // sll $2, $1, 4
000000_00000_00001_00010_00100_000010 // srl $2, $1, 4
000000_00000_00001_00010_00100_000011 // sra $2, $1, 4
```

测试结果如图所示：



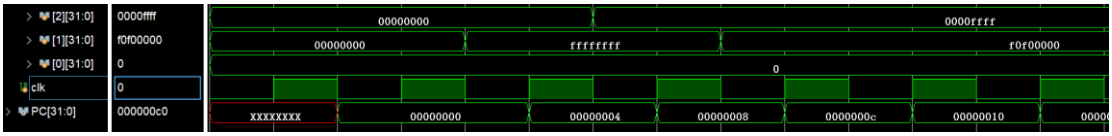

测试代码将 1 号寄存器设为 0xffffff0f0（有符号数扩展）。在 sll 指令中，1 号寄存器左移 4 位得到 0xffff0f00。在 srl 指令中，1 号寄存器右移四位得到 0x0fffff0f。在 sra 指令中，1 号寄存器右移 4 位，并在高位填充符号位 1，得到 0xffffff0f。仿真符合预期。

### 4.2.4 addi，addiu，lui 指令

编写 Instruction.txt 文件如下：

```
001000_00000_00001_1111111111111111 // addi $1, $0, 0xffff
001001_00000_00010_1111111111111111 // addiu $2, $0, 0xffff
001111_00000_00001_1111000011110000 // lui $1, 0xf0f0
```

测试结果如图所示：



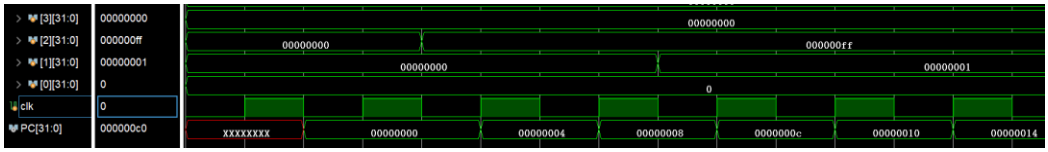

仿真代码中 addi 指令将 0xffff 做有符号数扩展得到 0xffffffff，addiu 指令将 0xffff 做无符号数扩展得到 0x0000ffff。lui 指令将立即数 0xf0f0 填入高 16 位，并把低 16 位清零得到 0xf0f00000。仿真符合预期。

### 4.2.5 slt，sltu 指令

编写 Instruction.txt 文件如下：

```
001000_00000_00010_0000000011111111 // addi $2, $0, 0xff
001010_00010_00001_1000000000000000 // slti $2, $1, 0x8000
001011_00010_00001_1000000000000000 // sltiu $2, $1, 0x8000
```

测试结果如图所示：




测试代码把 2 号寄存器设为 0x00ff。在 slti 指令中，0x8000 做有符号数扩展，为负数，0x00ff < 0xffff8000 不成立，故 1 号寄存器不设为 1。在 sltiu 指令中，0x8000 做无符号数扩展 0x00ff < 0xffff 成立，故 1 号寄存器设为 1。仿真符合预期。

4.3 拨码乘法器的测试

拨码乘法器需要引入 display 模块 IP 核并修改相应模块。实验中直接将拨码输入连接到 memory[0]与 memory[1]，并将 memory[2]连接至 display 数据。修改代码如下：

```
module DataMemory (
    input wire clk,
    input wire [31:0] address,
    input wire [31:0] writeData,
    input wire memWrite,
    input wire memRead,
    input wire [31:0] writeDataDMA1,
    input wire [31:0] writeDataDMA2,

    output reg [31:0] readData,
    output reg [31:0] readDataDMA
);

  reg [31:0] memory[0:63];

  always @(*) begin
    if (memRead) begin
      if (memWrite) begin
        readData = writeData;
      end else begin
        readData = memory[address>>2];
      end
    end else begin
      readData = 0;
    end
    readDataDMA = memory[2];
  end

  always @(posedge clk) begin
    memory[0] = writeDataDMA1;
    memory[1] = writeDataDMA2;
    if (memWrite) memory[2] = writeData;
  end
```

```
module Top (
    input wire clk_p,
    input wire clk_n,
    input wire reset_n,

    input wire [3:0] a,
    input wire [3:0] b,

    output wire led_clk,
    output wire led_en,
    output wire led_do,

    output wire seg_clk,
    output wire seg_en,
    output wire seg_do
);

  wire clk;
  wire clk_o;
  wire clkDisplay;

  IBUFGDS IBUFGDS_inst (
      .O (clk_o),
      .I (clk_p),
      .IB(clk_n)
  );

  assign clk = clk_o;

  reg [2:0] clkdiv;
  always @(posedge clk) clkdiv <= clkdiv + 1;
  assign clkDisplay = clkdiv[2];
```

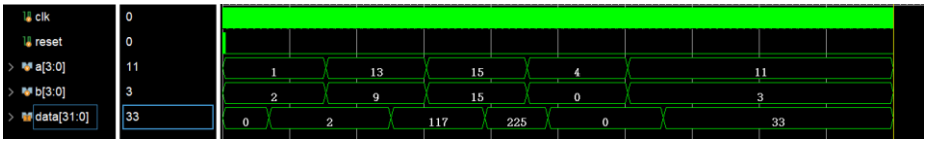
```
DataMemory dataMemory (
    .clk(clk),
    .address(ALUResult),
    .writeData(regReadData2),
    .memRead(memRead),
    .memWrite(memWrite),
    .writeDataDMA1({28'b0, a}),
    .writeDataDMA2({28'b0, b}),
    .readData(memReadData),
    .readDataDMA(product)
);

display display (
    .clk(clkDisplay),
    .rst(1'b0),
    .en(8'b00001111),
    .data(product),
    .dot(8'b00000000),
    .led(~16'b0),
    .led_clk(led_clk),
    .led_en(led_en),
    .led_do(led_do),
    .seg_clk(seg_clk),
    .seg_en(seg_en),
    .seg_do(seg_do)
);
```

修改 Instruction Memory 如下：

```
initial begin
  readData   = 32'b100011_00011_00001_0000000000000000;
  memory[0]  = 32'b100011_00011_00001_0000000000000000;  //   lw $1, 0($3)
  memory[1]  = 32'b100011_00011_00010_0000000000000100;  //   lw $2, 4($3)
  memory[2]  = 32'b000000_00000_00010_00100_00001_000010;  // srl $4, $2, 1
  memory[3]  = 32'b000000_00000_00100_00101_00001_000000;  // sll $5, $4, 1
  memory[4]  = 32'b000100_00010_00101_0000000000000001;  //   beq $5, $2, 1
  memory[5]  = 32'b000000_01000_00001_01000_00000_100000;  // add $8, $8, $1
  memory[6]  = 32'b000000_00000_00010_00010_00001_000010;  // srl $2, $2, 1
  memory[7]  = 32'b000000_00000_00001_00001_00001_000000;  // sll $1, $1, 1
  memory[8]  = 32'b000100_00011_00010_0000000000000001;  //   beq $2, $3, 1
  memory[9]  = 32'b000010_00000000000000000000000010;  //     j 2
  memory[10] = 32'b101011_00011_01000_0000000000001000;  //   sw $8, 8($0)
  memory[11] = 32'b000000_00000_00000_01000_00000_100000;  // add $8, $0, $0
  memory[12] = 32'b000010_00000000000000000000000000;  //     j 0
end
```

仿真结果如下：




上板结果通过了检查。

## **5.** 总结与反思

在 Lab05 中，我深入了解了 MIPS 单周期处理器的各部件功能和关系，通过实现 Program Counter 和 Branch Control 等功能，我熟悉了处理器取指、译码、读写、计算等部分的流程。通过使用 Vivado 开发环境，我能够更好地编写仿真文件调试 Verilog HDL 的代码。通过这次实验，我掌握了使用读写文件的方式初始化寄存器的方法，并学习了 genvar 功能。

我要感谢课程组为我们提供的详细指导书，它为我提供了清晰的实验步骤，使我能够更好地理解和实践所学的知识。通过这次实验，我不仅巩固了 Verilog 和 MIPS 处理器的基础知识，还为的学习和设计打下了坚实的基础。