计算机体系结构实验

# 实验5报告书

作者：李珉超

学号：515030910361

2017.4.2

# 1 实验介绍

## 1.1 实验名称

简单的类MIPS单周期处理器实现- 整体调试

## 1.2 实验目的

1.完成单周期的类MIPS处理器

## 1.3 实验范围

1.ISE 13.4 的使用

2.Xilinx Spartan 3E 实验板的使用

3.使用VerilogHDL进行逻辑设计

4.仿真测试，下载验证

# 2　实验过程

## 2.1　PC

### 2.1.1　实验原理

PC(Program Counter) 是CPU中计算下一条指令的模块。 par 对于非跳转的指令，PC自增4。对于jump指令，低26 位左移两位，再填充PC+4的高四位的结果作为下一条指令的地址。对于branch 指令，将扩展的16 位数左移两位再加上PC+4的结果作为下一条指令的地址。

由于其他的数据写入都是在时钟下降沿的，为了使得指令在一个周期内完成，PC的写入在时钟上升沿触发。 par PC的初始化是0，表明从instruction memory的0 地址开始读。

### 2.1.2　实验模块

```
module ProgramCounter(
    input clock_in,
    input reset,
    input [31:0] sgn_extend,
    input branch,
    input zero,
    input jump,
    input [25:0] inst,
    output reg [31:0] read_address
    );

    reg [31:0] PC;
    reg [31:0] PC_4;
    reg [28:0] inst_shft;
    reg [31:0] jump_address;
    reg [31:0] add_2;
    reg [31:0] mux_1;
    reg [31:0] mux_2;

    always @ (negedge reset)
    begin
        read_address = 0;
        PC = 0;
    end

    always @ (posedge clock_in)
    begin
        PC_4 = PC + 4;
        inst_shft = inst << 2;
        jump_address = inst_shft + (PC_4 & 8'hf0000000);
        add_2 = (sgn_extend << 2) + PC_4;
        mux_1 = (branch & zero) ? add_2 : PC_4;
        mux_2 = jump ? jump_address : mux_1;
        PC = mux_2;
        read_address = PC;
    end
endmodule
```

## 2.2 IntructionMemory

### 2.2.1 实验原理

Instruction memory 是计算机结构中存放指令的外存设备。注意在冯诺依曼架构中，指令和数据放在同一个存储设备中，但是为了让硬件更高效，或者为了避免structure hazard (pipeline)，我们将instruction memory和data memory分离。

Instruction memory在本实验中是只读的，故不需要clock信号。预设的指令通过reset写入。

### 2.2.2 实验模块

```
module instruction_memory(
    input reset,
    input [31:0] read_address,
    output [31:0] instruction
    );

    reg [31:0] instruction_mem[255:0];

    always @ (negedge reset)
    begin
        instruction_mem[0] = 32'b00001000000000000000000000000100;
        instruction_mem[1] = 32'b00000000000000000000000000000000;
        instruction_mem[2] = 32'b00000000000000000000000000000000;
        instruction_mem[3] = 32'b00000000000000000000000000000000;
        instruction_mem[4] = 32'b10001100000000010000000000000000;
        instruction_mem[5] = 32'b10001100000000100000000000000001;
        instruction_mem[6] = 32'b00000000001000100001100000100000;
        instruction_mem[7] = 32'b00000000001000100010000000100010;
        instruction_mem[8] = 32'b10101100000001000000000000100000;
        instruction_mem[9] = 32'b10001100000001010000000000000101;
        instruction_mem[10] = 32'b10001100000001100000000000000110;
        instruction_mem[11] = 32'b00000000101001100011100000100100;
        instruction_mem[12] = 32'b00000000101001100100000000100101;
        instruction_mem[13] = 32'b00000000101001100100100000100111;
    end

    assign instruction = instruction_mem[read_address >> 2];
endmodule
```

## 2.3 Top

### 2.3.1 实验原理

在顶层模块中，需要将之前的模块连接起来。需要在top模块中，声明许多wire变量。

这些模块是并发运行的，当一个模块的输出是另一个模块的输入，并且得到输出结果时，另一个模块能够立马使用该数据。顶层模块只有两个输入信号，clock和reset。

原先计划上板，故实现了分频器，所以仿真中main_Clock不是真正的时钟周期，而是TimeDv中的clk_dv。

### 2.3.2 实验模块

由于代码冗长，且只需要连接各个模块，故展示部分代码。

```
module top(
    input main_clk,
    input reset
    );
    wire [31:0] INST;
    wire REG_DST, JUMP, BRANCH, MEM_READ, MEM_TO_REG, MEM_WRITE;
    wire [1:0] ALU_OP;
    wire ALU_SRC, REG_WRITE;
    wire [31:0] READ_DATA1;
    wire [31:0] READ_DATA2;
    wire [31:0] READ_DATA;
    wire [31:0] SIGN_EXTEND;
    wire ZERO;
    wire [31:0] ADDRESS;
    wire [3:0] ALU_CTR;
    wire CLK_IN, RESET;
    wire [4:0] WRITE_REG;
    wire [31:0] WRITE_DATA;
    wire [31:0] READ_ADDRESS;
    wire [31:0] RESULT1;
    //connect the module
    //...
endmodule
```

### 2.3.3 仿真

先初始化main_clk=0, reset=1

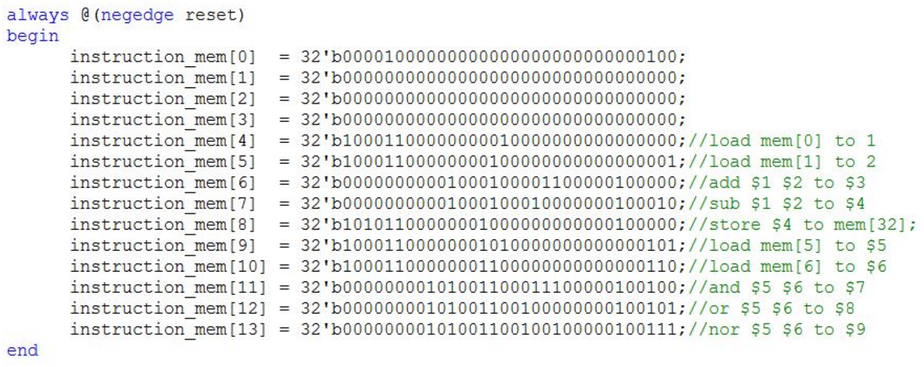
然后将reset设置为0，以初始化一些模块(register, instruction memory, PC, Control unit) 的初始数据。

注意，data memory的初始化是通过readmemh("mem_data.txt", memFile)实现的。

```
module test_for_top;
    // Inputs
    reg main_clk;
    reg reset;

    // Outputs
    wire [31:0] data;

    // Instantiate the Unit Under Test (UUT)
    top uut (
    .inst(inst),
    .data(data)
    );

    always #(100) main_clk = ~ main_clk;
    initial begin
        // Initialize Inputs
        main_clk = 0;
        reset = 1;

        // Wait 100 ns for global reset to finish
```

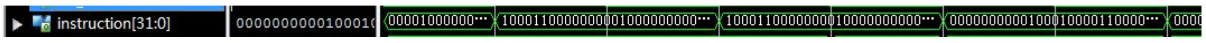
```
            #90;
            reset = 0;

        end
    endmodule
```

### 2.3.4 仿真图



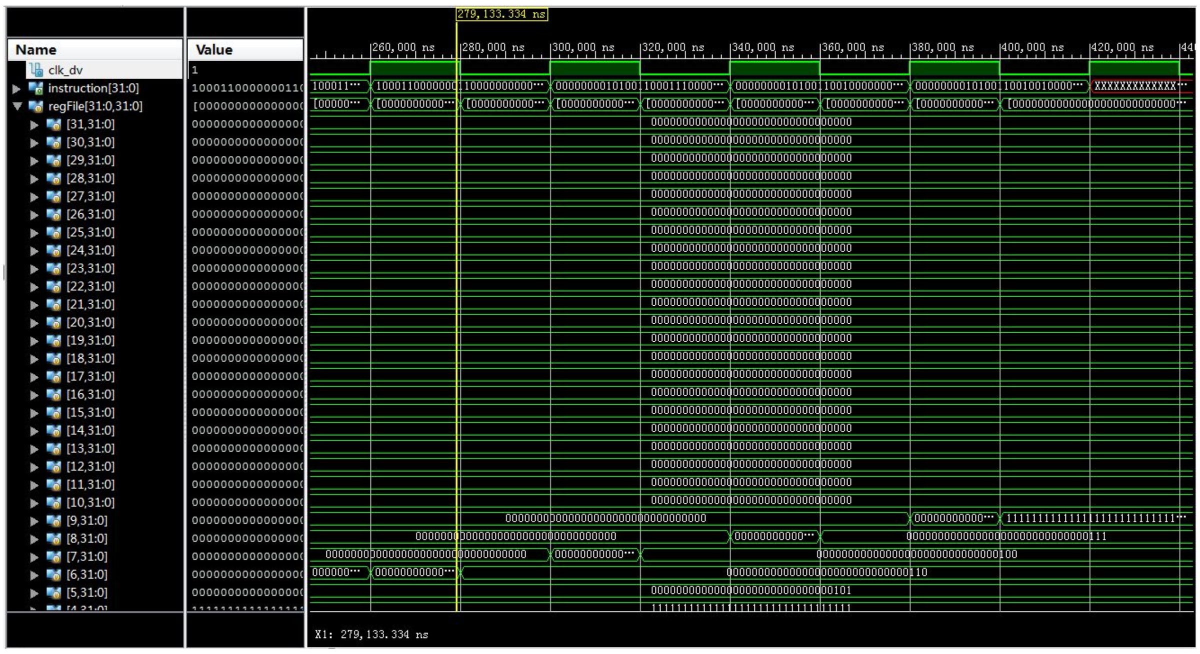
```
always @(negedge reset)
begin
      instruction_mem[0]  = 32'b00001000000000000000000000000100;
      instruction_mem[1]  = 32'b00000000000000000000000000000000;
      instruction_mem[2]  = 32'b00000000000000000000000000000000;
      instruction_mem[3]  = 32'b00000000000000000000000000000000;
      instruction_mem[4]  = 32'b10001100000000001000000000000000;//load mem[0] to 1
      instruction_mem[5]  = 32'b10001100000000010000000000000001;//load mem[1] to 2
      instruction_mem[6]  = 32'b00000000001000100001100000100000;//add $1 $2 to $3
      instruction_mem[7]  = 32'b00000000001000100010000000100010;//sub $1 $2 to $4
      instruction_mem[8]  = 32'b10101100000001000000000000100000;//store $4 to mem[32];
      instruction_mem[9]  = 32'b10001100000001010000000000000101;//load mem[5] to $5
      instruction_mem[10] = 32'b10001100000001100000000000000110;//load mem[6] to $6
      instruction_mem[11] = 32'b00000000101001100011100000100100;//and $5 $6 to $7
      instruction_mem[12] = 32'b00000000101001100100000000100101;//or $5 $6 to $8
      instruction_mem[13] = 32'b00000000101001100100100000100111;//nor $5 $6 to $9
end
```

这是我的指令的初始化内容。

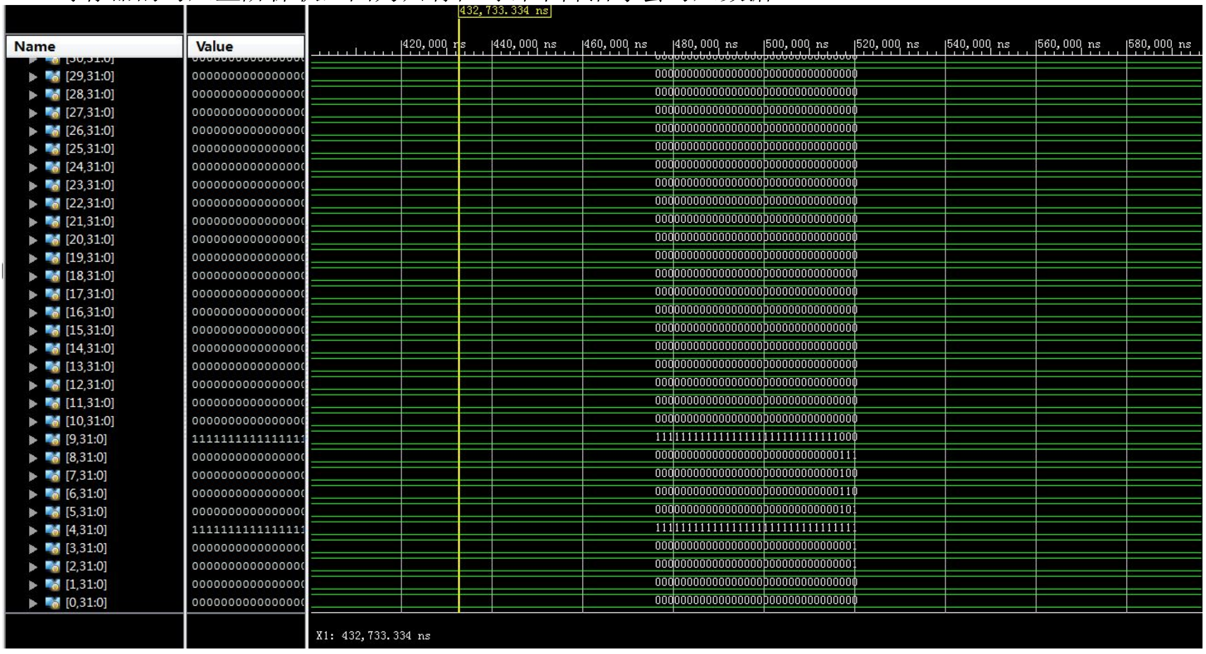


```
instruction[31:0]  000000000010001( 00001000000… 10001100000000001000000000… 10001100000000010000000000… 00000000001000100001100000… 0000
```

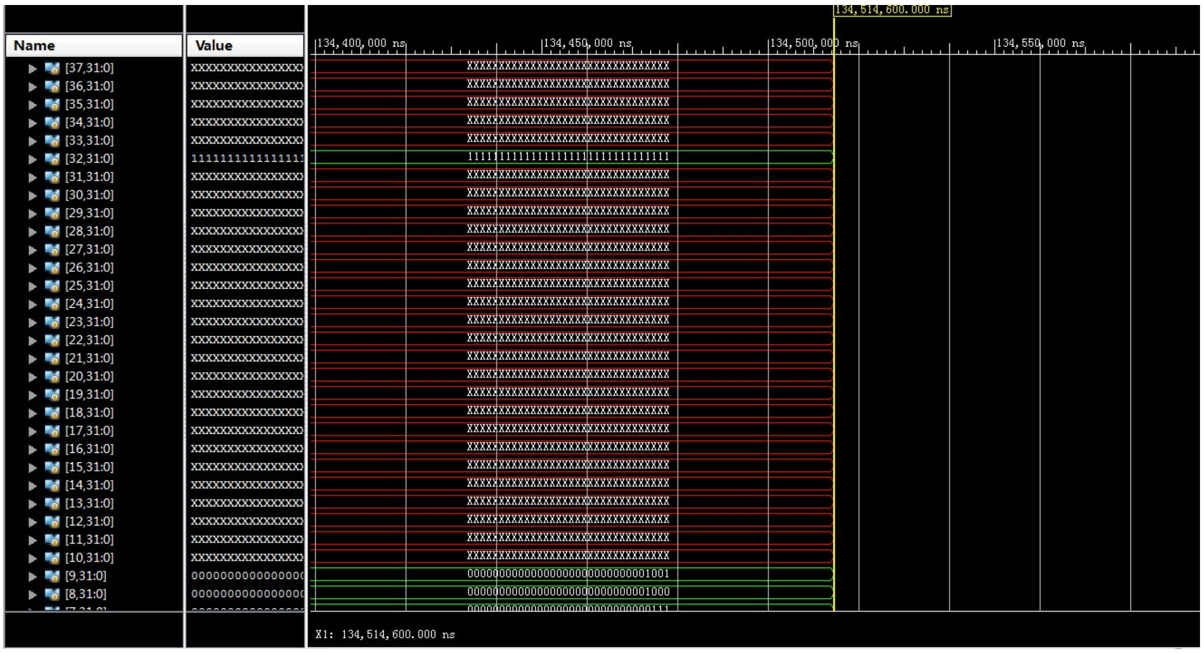
注意到,第一条jump语句执行后，后3条空指令并没有被执行，证明指令执行正确。


```
Name                Value
  clk_dv            1
▶ instruction[31:0] 100011000000011(
▼ regFile[31:0,31:0] [000000000000000(
  ▶ [31,31:0]       0000000000000000
  ▶ [30,31:0]       0000000000000000
  ▶ [29,31:0]       0000000000000000
  ▶ [28,31:0]       0000000000000000
  ▶ [27,31:0]       0000000000000000
  ▶ [26,31:0]       0000000000000000
  ▶ [25,31:0]       0000000000000000
  ▶ [24,31:0]       0000000000000000
  ▶ [23,31:0]       0000000000000000
  ▶ [22,31:0]       0000000000000000
  ▶ [21,31:0]       0000000000000000
  ▶ [20,31:0]       0000000000000000
  ▶ [19,31:0]       0000000000000000
  ▶ [18,31:0]       0000000000000000
  ▶ [17,31:0]       0000000000000000
  ▶ [16,31:0]       0000000000000000
  ▶ [15,31:0]       0000000000000000
  ▶ [14,31:0]       0000000000000000
  ▶ [13,31:0]       0000000000000000
  ▶ [12,31:0]       0000000000000000
  ▶ [11,31:0]       0000000000000000
  ▶ [10,31:0]       0000000000000000
  ▶ [9,31:0]        0000000000000000
  ▶ [8,31:0]        0000000000000000
  ▶ [7,31:0]        0000000000000000
  ▶ [6,31:0]        0000000000000000
  ▶ [5,31:0]        0000000000000000

279,133.334 ns
|260,000 ns |280,000 ns |300,000 ns |320,000 ns |340,000 ns |360,000 ns |380,000 ns |400,000 ns |420,000 ns
instruction: 100011… | 1000110000000 | 10000000000… | 0000000010100 | 10001110000… | 0000000010100 | 10010000000… | 0000000010100 | 10010010000… | XXXXXXXXXXXXX…
regFile: [00000… | [0000000000… | [0000000000… | [0000000000… | [0000000000… | [0000000000… | [0000000000… | [0000000000… | [00000000000000000000000000000…
[31]–[10]: 00000000000000000000000000000000
[9]: 00000000000000000000000000000000 | 00000000000… | 1111111111111111111111111111…
[8]: 00000000000000000000000000000000 | 00000000000… | 00000000000000000000000000000111
[7]: 00000000000000000000000000000000 | 00000000000… | 00000000000000000000000000000100
[6]: 000000… | 00000000000… | 00000000000000000000000000000110
[5]: 00000000000000000000000000000101
[4]: 11111111111111111111111111111111
X1: 279,133.334 ns
```

寄存器的写入呈阶梯状，因为只有在时钟下降沿才会写入数据。


```
432,733.334 ns
|420,000 ns |440,000 ns |460,000 ns |480,000 ns |500,000 ns |520,000 ns |540,000 ns |560,000 ns |580,000 ns
Name          Value
▶ [29,31:0]   0000000000000000   00000000000000000000000000000000
▶ [28,31:0]   0000000000000000   00000000000000000000000000000000
▶ [27,31:0]   0000000000000000   00000000000000000000000000000000
▶ [26,31:0]   0000000000000000   00000000000000000000000000000000
▶ [25,31:0]   0000000000000000   00000000000000000000000000000000
▶ [24,31:0]   0000000000000000   00000000000000000000000000000000
▶ [23,31:0]   0000000000000000   00000000000000000000000000000000
▶ [22,31:0]   0000000000000000   00000000000000000000000000000000
▶ [21,31:0]   0000000000000000   00000000000000000000000000000000
▶ [20,31:0]   0000000000000000   00000000000000000000000000000000
▶ [19,31:0]   0000000000000000   00000000000000000000000000000000
▶ [18,31:0]   0000000000000000   00000000000000000000000000000000
▶ [17,31:0]   0000000000000000   00000000000000000000000000000000
▶ [16,31:0]   0000000000000000   00000000000000000000000000000000
▶ [15,31:0]   0000000000000000   00000000000000000000000000000000
▶ [14,31:0]   0000000000000000   00000000000000000000000000000000
▶ [13,31:0]   0000000000000000   00000000000000000000000000000000
▶ [12,31:0]   0000000000000000   00000000000000000000000000000000
▶ [11,31:0]   0000000000000000   00000000000000000000000000000000
▶ [10,31:0]   0000000000000000   00000000000000000000000000000000
▶ [9,31:0]    1111111111111111   11111111111111111111111111111000
▶ [8,31:0]    0000000000000000   00000000000000000000000000000111
▶ [7,31:0]    0000000000000000   00000000000000000000000000000100
▶ [6,31:0]    0000000000000000   00000000000000000000000000000110
▶ [5,31:0]    0000000000000000   00000000000000000000000000000101
▶ [4,31:0]    1111111111111111   11111111111111111111111111111111
▶ [3,31:0]    0000000000000000   00000000000000000000000000000001
▶ [2,31:0]    0000000000000000   00000000000000000000000000000001
▶ [1,31:0]    0000000000000000   00000000000000000000000000000000
▶ [0,31:0]    0000000000000000   00000000000000000000000000000000
X1: 432,733.334 ns
```

最后寄存器里的数据正确，是指令运行后应该的结果。



```
Name        Value              |134,400,000 ns    |134,450,000 ns    |134,500,000 ns   |134,550,000 ns
                                                                      134,514,600.000 ns
[37,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[36,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[35,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[34,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[33,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[32,31:0]   1111111111111111   11111111111111111111111111111111
[31,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[30,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[29,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[28,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[27,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[26,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[25,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[24,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[23,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[22,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[21,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[20,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[19,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[18,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[17,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[16,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[15,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[14,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[13,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[12,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[11,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[10,31:0]   xxxxxxxxxxxxxxxx   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[9,31:0]    0000000000000000   00000000000000000000000000001001
[8,31:0]    0000000000000000   00000000000000000000000000001000

X1: 134,514,600.000 ns
```

对于sw指令，也执行正确，mem[31]被成功写入了正确的数据，由于初始化文件中只定义了0到9，所以其他地址的值未知。

这些表明，单周期类MIPS CPU的实现成功。

## 3 实验心得

了解了单周期CPU的运行方式以及各个指令data path，深刻的认识到各个模块之间的通信方式，信号和硬件之间的相互协作。

第一次实践了MIPS指令的编写，从另一个角度认识到instruction decode的过程。

单周期的处理器虽然不是当前处理器的使用方式，但却能帮我对pipeline的技术做铺垫