Lab5 单周期 MIPS CPU 设计

唐凯成 PB16001695

一. 实验目的

1、学习搭建单周期 MIPS CPU

- 理解单周期 CPU 的设计准则
- 熟练掌握单周期 CPU 的数据通路，合理编写对应的数据通路
- 学习对单周期 CPU 不同指令的信号控制

2、使用搭建的单周期 CPU 运行一段 32 位汇编指令

- 设计 CPU，完成右侧所示程序代码的执行，其功能是起始数为 3 和 3 的斐波拉契数列的计算。只计算 20 个数。
- 本段代码中主要要求 CPU 能实现 6 条基本指令，分别为（add、addi、lw、sw、bgtz、j）。
- 对于本次试验中涉及的两个 RAM 和一个 Reg，均采用异步读，同步写，以实现单周期即可完成任易指令的执行（lw 在下个周期开始时写回）。

```
.data
fibs: .word  0 : 20     # "array" of  20 words to contain fib values
size: .word  20         # size of "array"
temp: .word 3 3
    .text
    la   $t0, fibs      # load address of array
    la   $t5, size      # load address of size variable
    lw   $t5, 0($t5)    # load array size
    la   $t3, temp                  # load
    lw   $t3, 0($t3)
    la   $t4, temp
    lw   $t4, 4($t4)

    sw   $t3, 0($t0)    # F[0] = $t3
    sw   $t4, 4($t0)    # F[1] = $t4
    addi $t1, $t5, -2   # Counter for loop, will execute (size-2) times
loop: lw   $t3, 0($t0)     # Get value from array F[n]
    lw   $t4, 4($t0)    # Get value from array F[n+1]
    add  $t2, $t3, $t4   # $t2 = F[n] + F[n+1]
    sw   $t2, 8($t0)    # Store F[n+2] = F[n] + F[n+1] in array
    addi $t0, $t0, 4    # increment address of Fib. number source
    addi $t1, $t1, -1   # decrement loop counter
    bgtz $t1, loop      # repeat if not finished yet.
out:
            j out
```

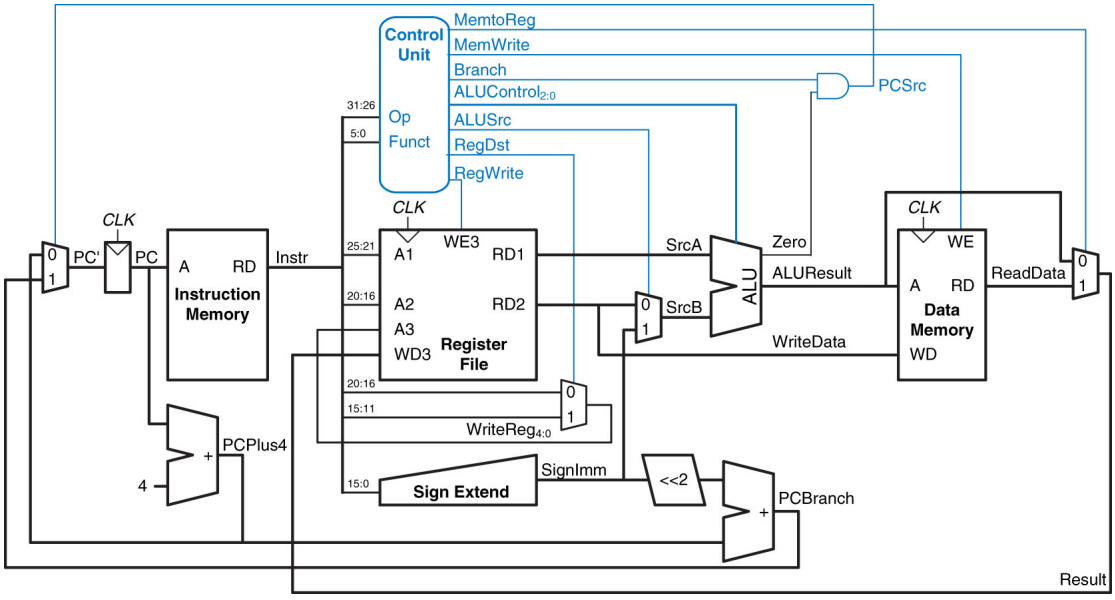
二. 实验平台

EDA 工具：Xilinx ISE14.7

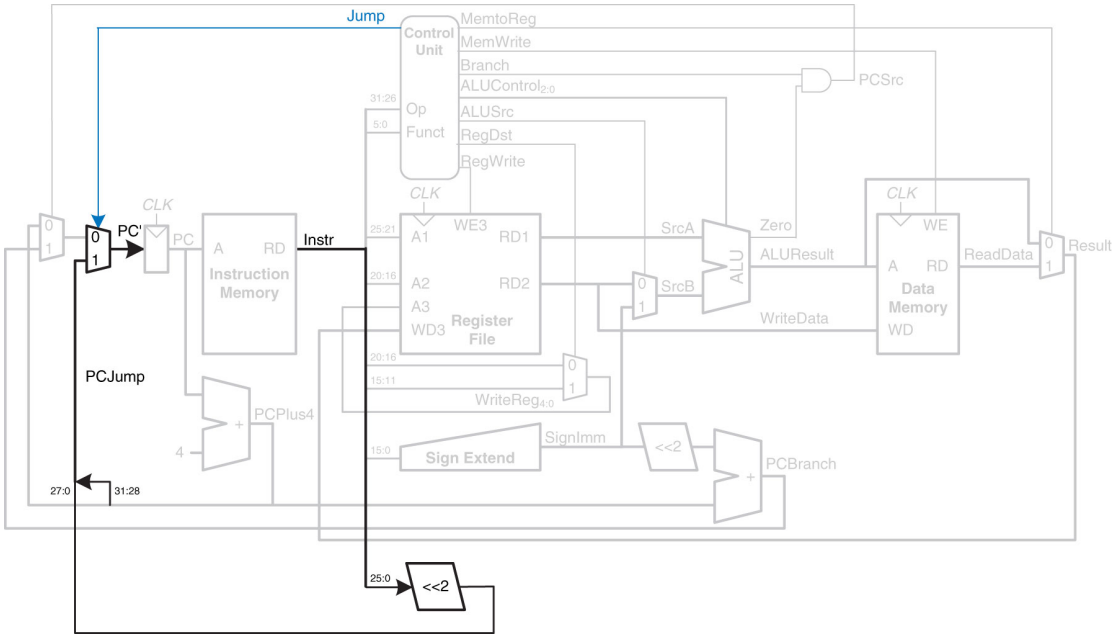
三. 实验过程

1、单周期 CPU 数据通路的设计

对于本次单周期 CPU 所需实现的六条基本指令，包括了三大类指令和转跳指令，故在采用普通单周期 CPU 数据通路后，还需加入转跳指令 j 的数据通路。本次单周期 CPU 的数据通路图如下：



转跳指令的数据通路为：



2、对数据通路中各个底层模块的设计

（1）PC 模块：

对于 PC 模块，主要为一个随时钟周期变化的锁存器。在单周期中，每个周期改变一次当前的 PC，将下个 PC 的值赋值给当前 PC。另外由于是单周期 CPU，每个周期完成整条指令，故无需 PC 模块的使能信号。

（2）指令存储器：

指令存储器可采用生成 IP 核的方式完成，可以简单的使用 Distribute Memory 来完成。不过由于现实应用中往往使用 8 位存储器来存储指令，且每条指令应拆分为四段分别存放在存储器的相邻四个单元中，而使用直接生成的 ROM 难以实现同时对相邻四个单元的读取，生成的黑箱结构难以修改，故在本次实验中采用了自己设计的指令存储器模块来代替。

自己设计的指令存储器的大小为 256*8bit，由 PC 模块输入当前所需执行指令的地址，找到对应的指令并直接在存储器中完成对指令的拆分，将原 32 位指令从 4*8bit 中提取出 op、rs、rt、rd、立即数等各个指令数。

（3）信号控制模块：

Control 模块为本次单周期 CPU 设计的核心模块，完成了每个指令对各个模块的不同控制。由于是单周期，每个指令在同一周期内完成，故在 Control 模块中采用组合逻辑即可，即使用一个 Case 语句完成对不同操作数所导致的各模块信号的不同赋值即可。

另外为了使 ALU 运算控制的结构更清晰，额外构建了 ALUControl 模块，用以接收 control 模块产生的 ALUop 及指令中的 funct 后生成 4 位的 ALU 控制信号，相当于一个二级译码器。

本次单周期中所需的各信号功能如下：

PCSrc：PC 地址的输入选择信号，用以默认加 4 或选择转跳成功时的 PC 值。

ALUSrcB：ALUB 的输入选择信号，用以选择使用寄存器中数据还是立即数。

RegDst：Reg 写入地址的选择信号，用以选择 rt 或 rd 输入寄存器写入地址。

MemtoReg：写回寄存器的选择信号，用以选择写回存储器数据或 ALU 结果。

RegWrite：寄存器的写使能控制信号。

MemWrite：数据存储器的写使能控制信号。
ExtSel：位拓展器控制信号，1 代表有符号拓展，0 代表无符号拓展。
ALUop：ALU 运算控制信号，输入 ALUcontrol 模块中得到二级控制信号。
ALUControl：4 位的 ALU 控制信号。
Jump：转跳指令的控制信号。
根据不同指令所得到的信号控制表为：

	PCSrc	RegDst	ALUSrcB	ExtSel	MemWrite	RegWrite	MemtoReg	ALUop	jump
R-type	0	1	0	0	0	1	0	00	0
Lw	0	0	1	1	0	1	1	01	0
Sw	0	0	1	1	1	0	0	01	0
Addi	0	0	1	1	0	1	0	01	0
Bgtz	0/1	0	0	1	0	0	0	10	0
j	0	0	0	0	0	0	0	01	1

（另外有关 ALU 的二级控制信号与书上内容相同，故省略。）

由上表即可完成组合逻辑模块的编写，另外考虑到整个表中 0 的数量较多且对每个指令均对 9 个信号赋值显得较为复杂，故在 case 语句之前就先将每个信号的值设为 0，后面 case 语句的每个分支中只需列出信号值为 1 的信号即可，如此既简化了程序，也便于理解与阅读，且无需再对信号赋初值。

（4）Reg 模块：

Reg 模块的设计基本与之前相同，主要加入了一个新的读取端口，另外将本来的异步读改为同步读，异步写保留不变，使用 assign 语句即可。

（5）ALU 模块：

对于 ALU 模块，由于需要处理 Branch 类型的指令，故需要加入一个 Zero 的输出用来判断运算结果是否为 0，当 ALU 输出为 0 时，Zero 输出为 1，反之为 0，在 always 语句里添加这一判断语句即可。

（6）数据存储器：

数据存储器由 IP 核生成，选择使用 Distribute Memory 中的 Single Port RAM 来实现，设置为 256*32bit，直接实现同步读异步写的要求。

（7）数据扩展器：

为实现对 16 位立即数的位扩展，加入该模块，用于将 16 位立即数拓展为 32 位立即数，但由于不同情况下需要进行有符号和无符号扩展，故在扩展时，若为无符号扩展，只需在高 16 位补 0 即可；若为有符号扩展，判断最高位第 16 位是否为 1，为 1 时在高 16 位补 1，不为 1 时在高 16 位补 0。如此即可完成对 16 位立即数的符号扩展。

另外为了使程序更简洁，可以使用带参数的模块引用。即在定义 Mux 模块时，对于位数[31:0]使用[EM:0]代替，这样可以一次性实现任意位数的数据选择器，在引用模块时只需采用 Mux #(n) Muxn（……）的格式，即代表引入了一个 n 位的数据选择器。

（8）数据选择器：

为使顶层模块相对简单，使用数据选择器模块。本次实验中需要使用到两种数据选择器，分别为 32 位和 5 位的 2 选 1 数据选择器。分别设计即可。

3、顶层模块 SingleCPU 的设计

完成各个底层模块的构建后，顶层模块主要用于各个底层模块的连线，以及对于 PC 地址的位移处理，即将立即数和 jump 的目标数向左位移 2 位。另外

由于加法在顶层模块中很容易实现，故不考虑单独写加法的 add 模块来专门用于几个有关 PC 地址的加法运算。对于其余连线部分，只需要按照数据通路和上述信号表连线即可。

四. 实验问题及解决方法

1、问题：原 MARS 代码中使用的 la 指令不在此次实验要求中故无法运行。
解决方法：

由对 la 指令的理解可得，其相当于取得对应数在内存中的地址并存在目标寄存器中，故一套 la+lw 指令恰好完成了从内存中取出预设的数的作用，可用简单的 addi 来代替，使用$zero 作为源寄存器，直接将我们所需的立即数与 0 相加后存入目标寄存器中，同样实现了 la+lw 指令的目的。

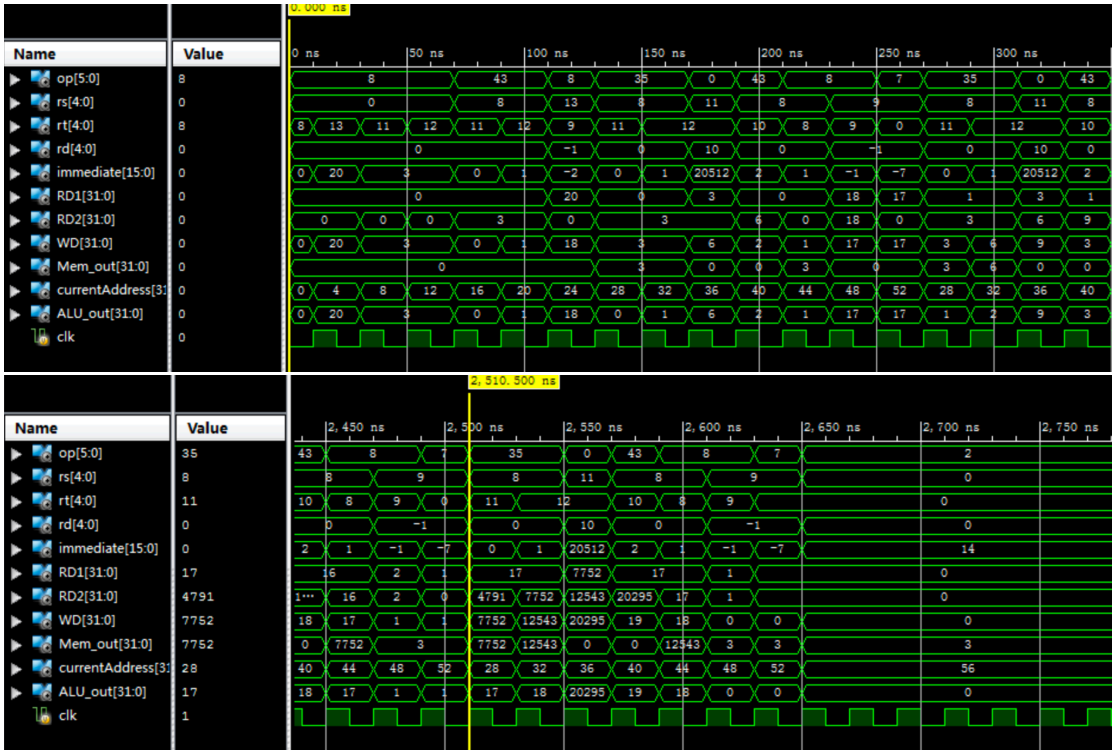
2、问题：原指令所对应的数据存储器为 8 位，而编写时使用的为 32 位。
解决方法：

考虑到方便本次实验的结果检验，仍保留 32 位的数据存储器，修改 MARS 的指令来解决该问题。本来指令中 lw 与 sw 命令中的 0、4、8 等立即数，均对应的是 8 位寄存器的数据，使用 32 位寄存器时，原来 8 位寄存器的每四个单元即为现在的一个单元，故将 0、4、8 改为 0、1、2 即可。

具体经过这两个问题修改后的 MARS 汇编指令见附录 3。

五. 实验结果

1、SingleCPU(top)模块的仿真结果：



2、存储器文件存储数据：

19	20295	12543
17	7752	4791
15	2961	1830
13	1131	699
11	432	267
9	165	102
7	63	39
5	24	15
3	9	6
1	3	3

3、寄存器文件存储数据：

19	0	0
17	0	0
15	0	0
13	20	12543
11	7752	20295
9	0	18
7	0	0
5	0	0
3	0	0
1	0	0

六. 附录

1、实验源码：

（1）PC 模块

```
module PC(
    input clk,
    input [31:0] newAddress,
    output reg[31:0] currentAddress
    );

    initial begin
        currentAddress = 0;
    end

    always@(posedge clk)
    begin
        currentAddress <= newAddress;
    end
endmodule
```

（2）指令寄存器模块

```
module InstructionMemory(
    input [31:0] InsAddr,
    output [5:0] op,
    output [4:0] rs,
    output [4:0] rt,
    output [4:0] rd,
     output [4:0] shamt,
     output [5:0] funct,
    output [15:0] immediate,
     output [25:0] target
```

```
    );
    reg[7:0] mem [255:0];
    initial
     begin
        $readmemb("test.txt", mem);
     end
    assign op = mem[InsAddr][7:2];
    assign rs[4:3] = mem[InsAddr][1:0];
    assign rs[2:0] = mem[InsAddr + 1][7:5];
    assign rt = mem[InsAddr + 1][4:0];
    assign rd = mem[InsAddr + 2][7:3];
     assign shamt[4:2] =mem[InsAddr + 2][2:0];
     assign shamt[1:0] =mem[InsAddr + 3][7:6];
     assign funct =mem[InsAddr + 3][5:0];
    assign immediate[15:8] = mem[InsAddr + 2];
    assign immediate[7:0] = mem[InsAddr + 3];
     assign target[25:24] = mem[InsAddr][1:0];
     assign target[23:16] = mem[InsAddr + 1];
     assign target[15:8] = mem[InsAddr + 2];
     assign target[7:0] = mem[InsAddr + 3];
endmodule
```

（3）主信号控制模块

```
module Control(
     input [5:0] op,
    input zero,
     output reg PCSrc,
    output reg ALUSrcB,
    output reg MemtoReg,
    output reg RegWrite,
    output reg MemWrite,
    output reg ExtSel,
    output reg RegDst,
     output reg jump,
     output reg [1:0] ALUop
);

always@(*)
begin
         PCSrc = 0;
         RegDst = 0;
         ALUSrcB = 0;
        ExtSel = 0;
        RegWrite = 0;
```

```
     MemWrite = 0;
     MemtoReg = 0;
      jump = 0;
      ALUop = 2'b01;
case(op)
    6'b000000://R-type
    begin
    RegDst = 1;
   RegWrite = 1;
    ALUop = 2'b00;
    end

    6'b100011://lw
    begin
    ALUSrcB = 1;
   ExtSel = 1;
   RegWrite = 1;
   MemtoReg = 1;
    end

    6'b101011://sw
    begin
    ALUSrcB = 1;
   ExtSel = 1;
   MemWrite = 1;
    end

    6'b001000://addi
    begin
    ALUSrcB = 1;
   ExtSel = 1;
   RegWrite = 1;
    end

    6'b000111://bgtz
    begin
    if (zero)
    begin
   PCSrc = 0;
   end
    else begin
   PCSrc = 1;
    end
   ExtSel = 1;
```

```
        ALUop = 2'b10;
        end

        6'b000010://j
        begin
        jump = 1;
        ALUSrcB = 1;
       ExtSel = 1;
        ALUop = 2'b11;
        end

    endcase
end
endmodule
```

（4）ALU 信号控制模块

```
module ALUcontrol(
    input [1:0] ALUop,
    input [5:0] funct,
    output reg [3:0] ALUControl
    );
always@(*)
begin
    case(ALUop)
        2'b00:
        begin
        case(funct)
        6'b100000: ALUControl = 4'b0010;//add
        6'b100010: ALUControl = 4'b0110;//sub
        6'b100100: ALUControl = 4'b0000;
        6'b100101: ALUControl = 4'b0001;
        6'b101010: ALUControl = 4'b0111;
        default: ALUControl = 4'b1111;
        endcase
        end

        2'b01: ALUControl = 4'b0010;
        2'b10: ALUControl = 4'b0110;
        default: ALUControl = 4'b1111;
    endcase
end
endmodule
```

（5）ALU 模块

```
module ALU(
 input signed [31:0] alu_a,
 input signed [31:0] alu_b,
 input [3:0] alu_op,
 output reg [31:0] alu_out,
 output reg zero
);
parameter A_ADD = 4'b0010;
parameter A_SUB = 4'b0110;
parameter A_AND = 4'b0000;
parameter A_OR = 4'b0001;
parameter A_XOR = 4'b0111;
parameter A_NOR = 4'b1100;

always @(*)
begin
    case (alu_op)
      A_ADD : alu_out = alu_a + alu_b;
     A_SUB : alu_out = alu_a - alu_b;
     A_AND : alu_out = alu_a & alu_b;
     A_OR : alu_out = alu_a | alu_b;
     A_XOR : alu_out = alu_a ^ alu_b;
     A_NOR: alu_out = ~(alu_a | alu_b);
     default: alu_out = 32'h0;
    endcase
     if (alu_out)  zero = 0;
    else  zero = 1;
end
endmodule
```

（6）寄存器模块

```
module REG_FILE(
    input clk,
    input [4:0] read_addr1,
    input [4:0] read_addr2,
    output [31:0] RD1,
    output [31:0] RD2,
    input [4:0] write_addr,
    input [31:0] WD,
    input wEna
);

    reg [31:0] register [31:0];
    integer i;
```

```
    initial
    begin
        for(i = 0; i < 32; i = i + 1)  register[i] <= 0;
    end

    assign RD1 = register[read_addr1];
    assign RD2 = register[read_addr2];

    always @(posedge clk)
    begin
        if (wEna)
        begin
            register[write_addr] <= WD;
        end
    end
endmodule
```

（7）数据选择器模块

```
module Mux(control,in1,in0,out);
     parameter EM = 31;
     input control;
    input [EM:0] in1;
    input [EM:0] in0;
    output [EM:0] out;
    assign out = control ? in1 : in0;
endmodule
```

（8）信号扩展模块

```
module SignExtend(
    input ExtSel,
    input [15:0] immediate,
    output [31:0] extendImmediate
    );

    assign extendImmediate[15:0] = immediate;
    assign extendImmediate[31:16] = ExtSel ? (immediate[15] ?
16'hffff : 16'h0000) : 16'h0000;
endmodule
```

（9）信号截取模块

```
module SignReduce(
    input [31:0] alu_out,
    output [7:0] memaddr
```

```
    );
    assign memaddr = alu_out[7:0];
endmodule
```

（10）SingleCPU(top)模块

```
module SingleCPU(
    input clk,
    output [5:0] op,
    output [4:0] rs,
    output [4:0] rt,
    output [4:0] rd,
    output [15:0] immediate,
    output [31:0] RD1,
    output [31:0] RD2,
    output [31:0] WD,
    output [31:0] Mem_out,
    output [31:0] currentAddress,
    output [31:0] ALU_out,
    output [7:0] memaddr,
    output [4:0] write_addr
    );

    wire [1:0] ALUop;
    wire [3:0] ALUControl;
   wire [5:0] funct;
    wire [25:0] target;
    wire [31:0] B, newAddress;
   wire [31:0] currentAddress_4, extendImmediate,
currentAddress_immediate,Address,JumpAddress;

   wire zero, PCSrc,ALUSrcB,RegWrite,MemtoReg,MemWrite,
ExtSel,RegDst,jump,MemWea;

    assign currentAddress_4 = currentAddress + 4;
   assign currentAddress_immediate = currentAddress_4 +
(extendImmediate << 2);
    assign JumpAddress[27:0] = (target << 2);
    assign JumpAddress[31:28] = currentAddress_4[31:28];

    PC pc(clk, newAddress, currentAddress);
   Control
cont(op,zero,PCSrc,ALUSrcB,MemtoReg,RegWrite,MemWrite,ExtSel,RegDst,j
ump,ALUop);
    ALUcontrol ALU_con(ALUop,funct,ALUControl);
```

```
    InstructionMemory
Ins_Mem(currentAddress,op,rs,rt,rd,shamt,funct,immediate,target);
    REG_FILE registerfile(clk,rs,rt,RD1,RD2,write_addr,WD,RegWrite);
    ALU alu(RD1,B,ALUControl,ALU_out,zero);
    SignReduce signre(ALU_out,memaddr);
    Datamomory Data_Mem(memaddr,RD2,clk,MemWrite,Mem_out);
    Mux MuxsrcB(ALUSrcB,extendImmediate,RD2,B),
        Muxmemtoreg(MemtoReg,Mem_out,ALU_out,WD),
MuxcurAddr(PCSrc,currentAddress_immediate,currentAddress_4,Address),
        MuxnewAddr(jump,JumpAddress,Address,newAddress);
    Mux #(4)Muxregw(RegDst,rd,rt,write_addr);
    SignExtend signex(ExtSel,immediate,extendImmediate);
Endmodule
```

2、仿真代码：

```
module test;
    reg clk;
    wire [5:0] op;
    wire [4:0] rs;
    wire [4:0] rt;
    wire [4:0] rd;
    wire [15:0] immediate;
    wire [31:0] RD1;
    wire [31:0] RD2;
    wire [31:0] WD;
    wire [31:0] Mem_out;
    wire [31:0] currentAddress;
    wire [31:0] ALU_out;
    wire [7:0] memaddr;
    wire [4:0] write_addr;
    SingleCPU uut (
        .clk(clk),
        .op(op),
        .rs(rs),
        .rt(rt),
        .rd(rd),
        .immediate(immediate),
        .RD1(RD1),
        .RD2(RD2),
        .WD(WD),
        .Mem_out(Mem_out),
        .currentAddress(currentAddress),
        .ALU_out(ALU_out),
        .memaddr(memaddr),
```

```
        .write_addr(write_addr),
        .signal(signal)
    );

    initial begin
        clk = 0;
        #100;
    end
   always
    begin
    #10 clk=~clk;
    end
endmodule
```

3、测试 MARS 汇编指令：

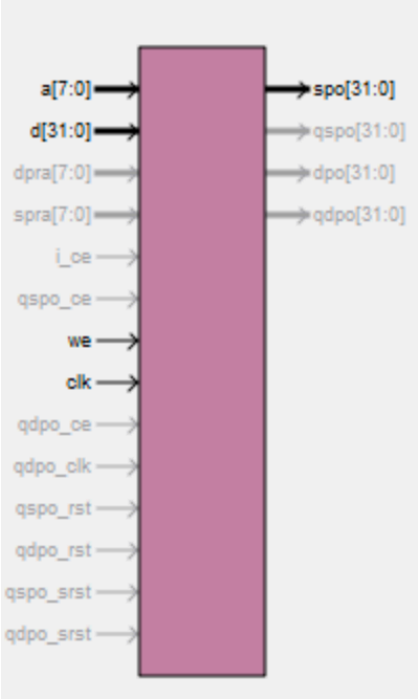
（1）MARS 中的指令

```
      addi    $t0, $zero,0
      addi    $t5, $zero,20
      addi    $t3,$zero,3
      addi   $t4,$zero,3
      sw    $t3, 0($t0)       # F[0] = $t3
      sw    $t4, 1($t0)       # F[1] = $t4
      addi $t1, $t5, -2
loop: lw    $t3, 0($t0)       # Get value from array F[n]
      lw    $t4, 1($t0)       # Get value from array F[n+1]
      add   $t2, $t3, $t4     # $t2 = F[n] + F[n+1]
      sw    $t2, 2($t0)       # Store F[n+2] = F[n] + F[n+1] in array
      addi $t0, $t0, 1        # increment address of Fib. number source
      addi $t1, $t1, -1       # decrement loop counter
      bgtz $t1, loop          # repeat if not finished yet.
out:
      j out
```

（2）指令对应的二进制数在存储器中的截图

63	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	00001110	00000000	00000000	00001000
55	11111001	11111111	00100000	00011101	11111111	11111111	00101001	00100001
47	00000001	00000000	00001000	00100001	00000010	00000000	00001010	10101101
39	00100000	01010000	01101100	00000001	00000001	00000000	00001100	10001101
31	00000000	00000000	00001011	10001101	11111110	11111111	10101001	00100001
23	00000001	00000000	00001100	10101101	00000000	00000000	00001011	10101101
15	00000011	00000000	00001100	00100000	00000011	00000000	00001011	00100000
7	00010100	00000000	00001101	00100000	00000000	00000000	00001000	00100000

4、RAM 的设置结果图：


a[7:0]
d[31:0]
dpra[7:0]
spra[7:0]
i_ce
qspo_ce
we
clk
qdpo_ce
qdpo_clk
qspo_rst
qdpo_rst
qspo_srst
qdpo_srst
spo[31:0]
qspo[31:0]
dpo[31:0]
qdpo[31:0]