Lab6 多周期 MIPS CPU 设计

唐凯成 PB16001695

一. 实验目的

1、学习搭建多周期 MIPS CPU

- 理解多周期 CPU 的设计准则
- 熟练掌握多周期 CPU 的数据通路，合理编写对应的数据通路
- 学习对多周期 CPU 不同指令的信号控制
- 学习使用状态机编写 Control 模块
- 思考多周期与单周期的本质区别

2、使用搭建的多周期 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。

```
.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
```

- 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）。

3、使用单个同步 RAM 实现多周期 CPU

- 自主调整原本异步 RAM 的数据通路和状态机信号控制
- 对比同步和异步 RAM 的不同即优缺点，为流水线做好准备

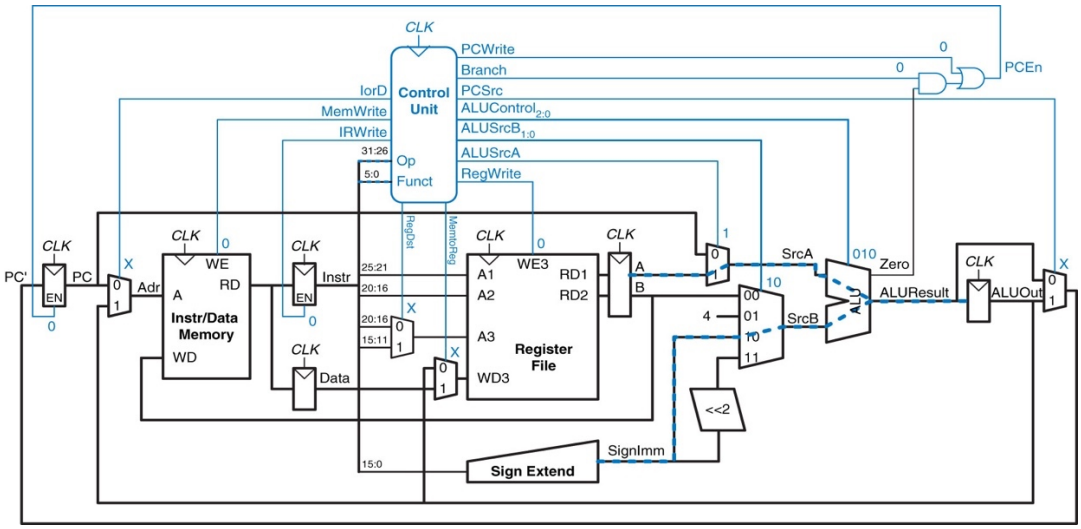
二. 实验平台

EDA 工具：Xilinx ISE14.7

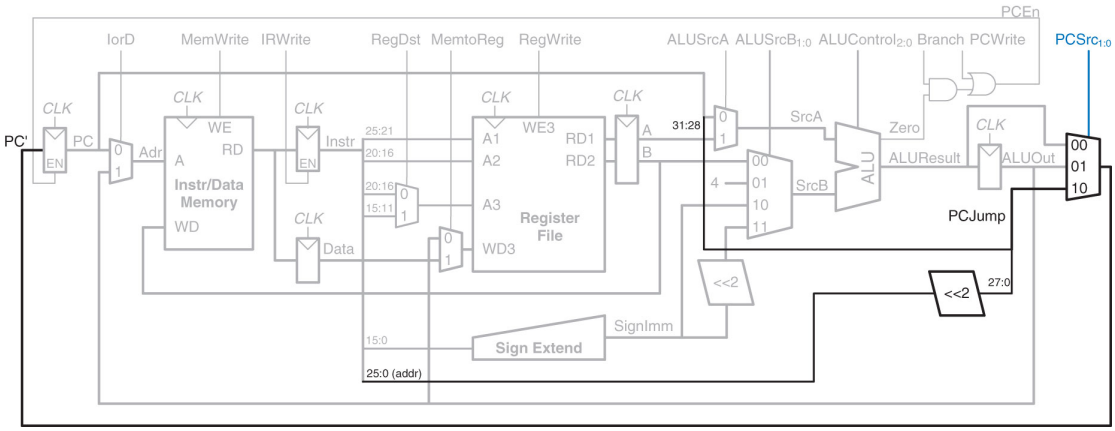
三. 实验过程

1、多周期 CPU 数据通路的设计

对于本次多周期 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，故采用经典的多周期数据通路，本次多周期 CPU 的数据通路图如下：

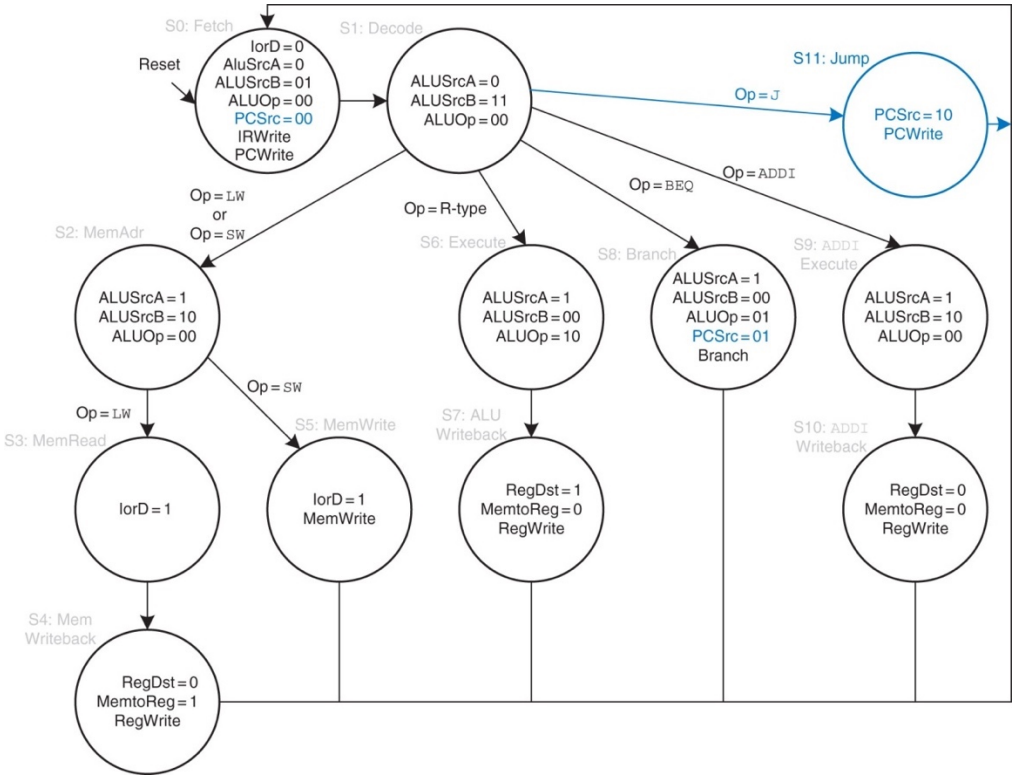


对于转跳指令还需多添加一条数据通路如下：



2、多周期 CPU 的状态机信号控制设计：

对于本次多周期的信号控制模块，主要为一个状态机控制,如下图中，每个圆圈代表一个状态以及该状态下要调整的信号。Control 模块设计中对该状态机结构还有微调，具体见下文中 Control 模块的设计。



3、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。与单周期主要的区别为加入了使能输入与重置输入。重置输入给出每次指令存放的起始地址，本次设计中在重置信号 reset 变为 1 时将地址置 0，整个 CPU 开始执行第一条指令。对于使能输入，由于多周期中对 PC 值得写会可能出现在多个周期，故必须设置一个使能信号来控制 PC 的写入，避免 PC 的错误写入。

（2）指令存储器&数据存储器：

根据本次实验的要求，将指令存储器与数据存储器合并，并采用 Block Memory 类型生成 single port RAM 来实现同步 RAM。同步 RAM 的写入和读出都需要一个额外的时钟周期，故对于状态机的信号控制也要进行调整，使得 RAM 完成异步读异步写的效果。

另外由于要同时存放指令与数据，对数据的分割会使实验结果较难观察，故使用 256*32bit 的 RAM，每个单元存放一条指令或数据，因此在 PC 值自加的过程中将+4 改为+1，对于 bgtz 和 j 指令则无需将对应的立即数或 26 位地址左移两位。最后定义 RAM 的 0-99 单元为指令存储单元，100 以上的单元为数据存储单元，以此来分开两种存入数据。

（3）数据通路里的寄存器：

由于在多周期中，每个周期数据在数据通路中均要有一周期的寄存，所以需在各个本来没有 clk 控制的数据通路上加入带 clk 的数据寄存器。下面依次分析数据通路中各个位置的数据寄存器：

1）RAM 输出的指令寄存器：

因此对于本来的异步 RAM 需加入指令与数据寄存器，而对于修改后的同步 RAM，由于异步读已经使得数据多了一周期的寄存时间，相当于已经实现了指令与数据寄存器的功能，故无需再添加该寄存器。但由于原设计中指令寄存器中有使能输入，而 若直接去掉会使得一条指令在运行中发生改变而导致错误，故在此将指令寄存器设计为一简单的锁存器，通过信号 IRWrite 来确认是否允许锁存器的输出被输入修改，即可完成对原数据通路的修改。

2）RAM 输出的数据寄存器：

原因和 1 中相同，由于选用同步 RAM，故可直接去除该处的数据寄存器。

3）其余数据寄存器：

其余寄存器包括主寄存器两个读出 RD1 与 RD2 的两个寄存器与 ALU 输出后的一个寄存器。这三个寄存器结构完全一样，均为一个含 clk 的触发器构成，故只需要写一个模块，例化时分别用在以上三处即可。

（4）信号控制模块：

Control 模块为本次多周期 CPU 设计的核心模块，完成了每个指令在不同周期对各个模块的不同控制。对 ALU 运算控制仍使用 Control+ALUControl 双模块实现。对多周期的控制信号如下：

PCSrc：2 位的 PC 地址的输入选择信号，用以选择各种方式得到的新 PC 地址。

LorD：存储器地址的输入选择信号，用以选择是输入 PC 值和写回数据。

ALUSrcA：ALUA 的输入选择信号，用以选择当前 PC 地址和寄存器读出数据。

ALUSrcB：2 位 ALUB 的输入选择信号，用以选择使用寄存器读出数据、数字 1、扩展后的立即数和扩展并左移两位的立即数。

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。

PCWrite：PC 寄存器的使能信号，控制 PC 地址的改变。

IRWrite：指令寄存器的使能信号，控制指令的改变。

Branch：分支指令的使能信号，与 Zero 进行与运算来更改 PC 的使能信号。

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：结合 funct 与 ALUop 给出的最终的 4 位的 ALU 控制信号。

对于多周期的信号控制采用状态机来设计。本次状态机的编写采用了一个时序逻辑和两个组合逻辑，时序逻辑结构用于切换状态机状态，组合逻辑结构分别用来定义下一状态与给出当前状态的不同信号值。

完成了对状态机的结构设计后，定义了 6 个状态，分别为：
1）IDLE：用于在 reset 信号置 1 前循环保持状态机初始态。
2）IF：取指阶段。进行对指令的提取及 PC 地址的自加操作（本次实验中+1）。
3）ID：译指阶段。进行对指令的拆分和寄存器中数据的读取与 PC 地址的运算。
4）EX：运算阶段。进行数据的运算，在分支与转跳指令中同时完成 PC 的写回。
5）MEM：访存阶段。进行从存储器中数据读出或写入，和寄存器的写回。
6）WB：写回阶段。进行寄存器数据的写回。

该状态机的定义与之前 2 中状态机的设计图略有不同，没有采用对不同指令均设置不同状态，而将所有指令均分为取指、译指、运算、访存和写回的五个阶段，并在每个阶段里对于不同指令的 op 再使用 case 语句进行不同的信号输出与下一状态的选择，如此来实现不同周期的信号控制。

对于每个状态不同指令下的信号控制，基本与 2 中图片一致，但其中取指圆圈中的 IRWrite 信号应移到后面的译指圆圈中，由于取指阶段时由于采用同步 RAM，其输出仍为上条指令，故延缓一个周期再读才为本次的指令。

（5）Reg 模块：

Reg 模块的设计基本与单周期 CPU 中相同，另外添加了一个对于 0 号寄存器的优化：即 0 号寄存器中始终存储 0，无法写入 0 号寄存器。

（6）ALU 模块：

ALU 模块也大体保持单周期中的设计，对于 zero 输出改为使用 assign 语句而不再和 case(op)放在一个 always 语句中，使程序结构更清晰。

（7）数据扩展器：

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。

（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三种数据选择器，分别为 2 选 1、3 选 1 与 4 选 1 数据选择器。分别设计即可。

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。

当数据选择器存在空闲的输入端口时，将对应信号的输出定为 x 即可。

4、顶层模块 MulticycleCPU 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。

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，由于本次用的是 32 位的存储器，故无需再对立即数和 jump 的目标数进行位移操作。直接使用 assign 语句完成 jump 指令的连接即可。

对于本次多周期的指令，没有再使用指令寄存器对其进行划分，故需在顶层模块里进行划分并接入其余模块中。

对于 PC 的写入使能 PCen，由 PCWrite、Branch 与 Zero 共同决定，由于 PCWrite 只负责在 IF 阶段完成 PC 地址的默认自加，其余阶段中均置 0，Branch 与 Zero 同时为 1 时代表分支成功，故对 Branch 与 Zero 两个信号使用一个与运算，再将其结果与 PCWrite 进行一个或运算，即可得到正确的 PC 使能。实际编写时，只需用一个 assign 语句即可完成。

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：对 MARS 代码中的简单修改。

解决方法：

同单周期中修改方法：la 指令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。

对于指令中 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，故在使用 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。

2、问题：状态机编写时的调整与优化。

解决方法：

在起初编写状态机时采用了两个时序逻辑结构与一个组合逻辑结构，即对信号的输出控制采用时序逻辑，然而由于时序逻辑会使信号的输出为上一状态的，因此信号输出慢了一个时钟周期。此时采用次态来进行信号的输出控制，成功地解决了这一问题。

但由于在时序逻辑中无法先对各个信号初始化，因此会导致每个状态中都需对所有信号重新赋值（或最少对上个状态更改的信号重新赋值），如此便导致程序长度的大大增加，且容易出现信号的置 0 等错误。此时考虑到状态机中也可以使用组合逻辑来控制每个输出信号，且组合逻辑的使用比时序逻辑更加简单且可以进行各信号的初始化操作，故在尝试后最终使用组合逻辑来控制信号。

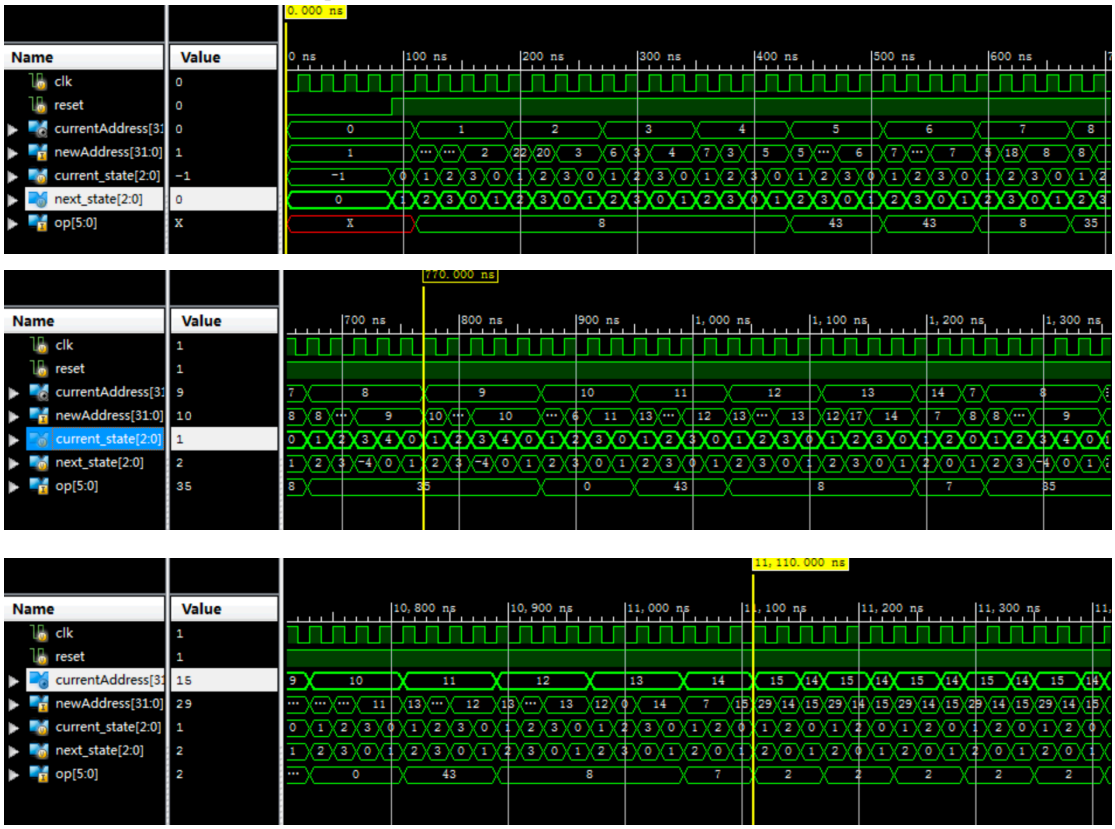
3、问题：对于指令寄存器修改的错误。

解决方法：

最初准备将指令寄存器去掉 clk 的控制后改为一个触发器，在 IRWrite 的上升沿改变寄存器的输出值即可，但在实际操作中发现存在 IRWrite 上升沿时指令寄存器的输入也正好同时改变的问题，此时输出的仍为上一个指令，因此导致了许多错误。最后采用了组合逻辑将寄存器写为一个锁存器，在 IRWrite 与输入改变时均进行判断：若此时 IRWrite 为 1 即刷新输出值。如此便解决的本来的输出问题。

五.实验结果

1、MulticycleCPU (top)模块的仿真结果：



2、存储器文件存储数据：

	0	1
100	3	3
102	6	9
104	15	24
106	39	63
108	102	165
110	267	432
112	699	1131
114	1830	2961
116	4791	7752
118	12543	20295

3、寄存器文件存储数据：

	0	1
31	X	X
29	X	X
27	X	X
25	X	X
23	X	X
21	X	X
19	X	X
17	X	X
15	X	X
13	20	12543
11	7752	20295
9	0	118
7	X	X
5	X	X
3	X	X
1	X	X

六.附录

1、实验源码：

（1）PC 模块

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

    always@(posedge clk or negedge reset)
    begin
        if (~reset)
         begin
         currentAddress <= 0;
         end
         else if (PCen) begin
         currentAddress <= newAddress;
         end
         else begin
         currentAddress <= currentAddress;
         end
    end
endmodule
```

（2）指令寄存器模块

```
module InstrReg(
     input IRWrite,
     input [31:0] Ins_in,
     output reg [31:0] Ins_out
     );
always@(*)
    begin
    if (IRWrite)
    begin
    Ins_out <= Ins_in;
    end
    else begin
    Ins_out <= Ins_out;
    end
    end
endmodule
```

（3）Control 模块

```
module Control(
     input clk,
     input reset,
     input [5:0] op,
     output reg [1:0] PCSrc,
     output reg ALUSrcA,
     output reg [1:0] ALUSrcB,
     output reg lorD,
     output reg MemtoReg,
     output reg IRWrite,
     output reg RegWrite,
     output reg MemWrite,
     output reg ExtSel,
     output reg RegDst,
     output reg Branch,
     output reg PCWrite,
     output reg [1:0] ALUop
);
reg [2:0] current_state,next_state;
parameter [5:0] addi = 6'b001000,
                R_type = 6'b000000,
                sw = 6'b101011,
                lw = 6'b100011,
                   bqtz = 6'b000111,
                j = 6'b000010,
                halt = 6'b111111;

parameter [2:0] IDLE = 3'b111,
                   IF = 3'b000,
                ID = 3'b001,
                EX = 3'b010,
                MEM = 3'b011,
                WB = 3'b100;

always @(posedge clk or negedge reset)
begin
    if (~reset)
    begin
        current_state <= IDLE;
   end
    else begin
      current_state <= next_state;
   end
```

```
end

always@(*)
    begin
        case(current_state)
            IDLE: next_state <= IF;
            IF: next_state <= ID;
            ID: next_state <= EX;
           EX:
            begin
            if (op == bqtz||op == j)
            next_state <= IF;
            else
            next_state <= MEM;
            end
            MEM:
            begin
            if(op == lw)
            next_state<=WB;
            else
            next_state <= IF;
            end
            WB: next_state <= IF;
            default: next_state <= IDLE;
        endcase
    end

always@(*)
begin
          lorD = 0;
          Branch = 0;
          PCWrite = 0;
          IRWrite = 0;
          RegDst = 0;
          ALUSrcA = 0;
          ALUSrcB = 2'b01;
         ExtSel = 0;
         RegWrite = 0;
         MemWrite = 0;
         MemtoReg = 0;
          ALUop = 2'b00;
          PCSrc = 2'b00;
    if (~reset) ;
    else
```

```
begin
case (current_state)

    IF:
    begin
    PCWrite = 1;
    end

    ID:
    begin
    ALUSrcB = 2'b11;
    IRWrite = 1;
    ExtSel = 1;
    end

    EX:
    begin
    case(op)
        lw:
        begin
        ALUSrcA = 1;
        ALUSrcB = 2'b10;
        ExtSel = 1;
        end

        sw:
        begin
        ALUSrcA = 1;
        ALUSrcB = 2'b10;
        ExtSel = 1;
        end

        R_type:
        begin
        ALUSrcA = 1;
        ALUSrcB = 2'b00;
        ALUop = 2'b10;
        end

        bqtz:
        begin
        ALUSrcA = 1;
        ALUSrcB = 2'b00;
        ALUop = 2'b01;
```

```
    PCSrc = 2'b01;
    Branch = 1;
    ExtSel = 1;
    end

    addi:
    begin
    ALUSrcA = 1;
    ALUSrcB = 2'b10;
    ExtSel = 1;
    end

    j:
    begin
    PCSrc = 2'b10;
    PCWrite = 1;
    end
    endcase
end

MEM:
begin
    case(op)
    lw:
    begin
    lorD = 1;
    end
    sw:
    begin
    lorD = 1;
    MemWrite = 1;
    end
    R_type:
    begin
    RegDst = 1;
    RegWrite = 1;
    end
    addi:
    begin
    RegWrite = 1;
    end
    endcase
end
```

```
        WB:
        begin
            MemtoReg = 1;
            RegWrite = 1;
        end
        default:;
    endcase
    end
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 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
end

assign zero= alu_out ?  0 : 1;
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];

    assign RD1 = (read_addr1 == 0)? 0 : register[read_addr1];
    assign RD2 = (read_addr2 == 0)? 0 : register[read_addr2];

    always @(posedge clk)
```

```
    begin
        if (wEna)
        begin
            if (write_addr != 0)
            register[write_addr] <= WD;
        end
    end
endmodule
```

（7）数据选择器模块

1）2 选 1 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
```

2）3 选 1 模块：

```
module Mux3to1(control,in0,in1,in2,out);
     parameter EM = 31;
     input [1:0] control;
    input [EM:0] in0;
    input [EM:0] in1;
     input [EM:0] in2;
    output reg [EM:0] out;
always@(*)
begin
    case(control)
     2'b00: out = in0;
     2'b01: out = in1;
     2'b10: out = in2;
     default: out = 32'hxxxxxxxx;
    endcase
end
```

3）4 选 1 模块：

```
module Mux4to1(control,in0,in1,in2,in3,out);
     parameter EM = 31;
     input [1:0] control;
    input [EM:0] in0;
    input [EM:0] in1;
     input [EM:0] in2;
     input [EM:0] in3;
```

```
    output reg [EM:0] out;
always@(*)
begin
     case(control)
     2'b00: out = in0;
     2'b01: out = in1;
     2'b10: out = in2;
     2'b11: out = in3;
    endcase
end
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）各处的数据寄存器模块：

```
module Reg_clk(
    input clk,
     input [31:0] in,
     output reg [31:0] out
     );
always@(posedge clk)
    begin
    out <= in;
    end
endmodule
```

（10）MulticycleCPU (top)模块

```
module MulticycleCPU(
    input clk,
    input reset,
    output out
    );

    wire [1:0] ALUop,ALUSrcB,PCSrc;
```

```
    wire [3:0] ALUControl;
   wire [4:0] regw_addr;
    wire [31:0] A,B,newAddress,RD1,RD2,WD,Mem_out,currentAddress,
ALU_out,memaddr,Ins_out,ScrA,ScrB,ALUOut;
   wire [31:0] extendImmediate,currentAddress_immediate,
Address,JumpAddress;

   wire Zero,lorD,PCen,Branch,PCWrite,IRWrite,ALUSrcA,
RegWrite,MemtoReg,MemWrite,ExtSel,RegDst;

    assign out=Ins_out;
    assign PCen=(~Zero & Branch)|PCWrite;
   assign currentAddress_immediate = extendImmediate;
    assign JumpAddress[27:0] = Ins_out[25:0];
    assign JumpAddress[31:28] = currentAddress[31:28];

    PC pc(clk,reset,PCen,newAddress,currentAddress);
    Memory Ins_Data_Memory(clk,MemWrite,memaddr[7:0],B,Mem_out);
    InstrReg Ins_reg(IRWrite,Mem_out,Ins_out);
   Control control(clk,reset,Ins_out[31:26],PCSrc,ALUSrcA,ALUSrcB,
lorD,MemtoReg,IRWrite,RegWrite,MemWrite,ExtSel,RegDst,Branch,PCWrite,
ALUop);
    ALUcontrol ALU_con(ALUop,Ins_out[5:0],ALUControl);
    SignExtend signex(ExtSel,Ins_out[15:0],extendImmediate);
    REG_FILE registerfile(clk,Ins_out[25:21],Ins_out[20:16],RD1,RD2,
regw_addr,WD,RegWrite);
    Reg_clk RegA(clk,RD1,A),
             RegB(clk,RD2,B),
             RegALU(clk,ALU_out,ALUOut);
    ALU alu(ScrA,ScrB,ALUControl,ALU_out,Zero);

    Mux4to1
MuxsrcB(ALUSrcB,B,32'd1,extendImmediate,currentAddress_immediate,ScrB
);
    Mux3to1 MuxPC(PCSrc,ALU_out,ALUOut,JumpAddress,newAddress);
    Mux Muxlord(lorD,ALUOut,currentAddress,memaddr),
        MuxsrcA(ALUSrcA,A,currentAddress,ScrA),
        Muxmemtoreg(MemtoReg,Mem_out,ALUOut,WD);
    Mux #(4)Muxregw(RegDst,Ins_out[15:11],Ins_out[20:16],regw_addr);
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

	0	1
0	00100000000010000000000001100100	00100000000011010000000000010100
2	00100000000010110000000000000011	00100000000011000000000000000011
4	10101101000010110000000000000000	10101101000011000000000000000001
6	00100001101010011111111111111110	10001101000010110000000000000000
8	10001101000011000000000000000001	00000001011011000101000000100000
10	10101101000010100000000000000010	00100001000010000000000000000001
12	00100001001010011111111111111111	00011101001000001111111111111001
14	00001000000000000000000000001110	00000000000000000000000000000000

4、RAM 的设置结果图：