CPU设计实验报告

一、实验概述

本实验使用的环境是vscode+verilog编译器进行编译，使用命名格式如testbench_的文件进行输入的测试，最后使用gtkwave来查看波形。
本实验主要完成了11条指令的MIPS32单周期CPU，11条指令的MIPS32多周期CPU（包括组合逻辑控制cpu和微程序控制cpu），以及为总线结构多周期CPU画了状态转换图。

二、MIPS指令集进行汇编程序设计

对于MIPS指令集，使用MARS设计汇编代码，调试正确后，并且生成相应的机器码，作为输入用于单周期和多周期cpu的测试。本实验中所运行的测试程序如下
测试程序1：

```
//MIPS32中有li指令，为给一个通用寄存器赋初值，但是使用MARS编译之后发现生成的中间代码使用的是addiu $tx,$0,立即数 的形式，故这里写成了这种形式
//同时addiu是无符号加法指令，且不判断溢出
.txt
main:
    addiu $t0,$0,1 #置$t0的初值为1，为指令li $t0 1的实现，因为MIP32的寄存器$0值规定恒为0
    addiu $t1,$0,3
    addiu $t3,$0,4
    add $t2,$t0,$t1
    beq $t2,$t3,next
    sub $t0,$t1,$t0
next:
    sub $t0,$t0,$t1
```

汇编后得到的机器码如下：

```
24080001
24090003
240b0004
01095020
114b0001
01284022
01094022
```

测试程序2：

```
.txt
main:
    lw $2 0($1)
```

汇编后得到的机器码如下：

```
8c220000
```

三、逻辑模块设计

对于单周期和多周期，逻辑模块设计都包括寄存器，存储器，寄存器堆RF，ALU，ALU_CU，控制单元CU，多路选择器，符号扩展器，同时单周期还包括加法器Add。在逻辑模块设计完成后，则可以对模块进行实例化，然后用于单周期和多周期CPU的设计。
在处理这些模块时，我认为对于一个模块，输入之后如果立刻得到输出，在时序波形图上的图象是在时钟上跳沿到达的情况下，就立刻刷新完成的。我认为这样子的时序波形图看起来有一点不太符合实际，所以我为访存设置了1个时间单位的延时，为其他所有的组合逻辑操作设置了0.1个时间单位的延时。
同时由于单周期多周期的cpu很多模块都具有共通性，因此在这里先介绍具有共同性的模块，包括寄存器，存储器，寄存器堆RF，ALU，ALU_CU，多路选择器，符号扩展器，最后的控制单元CU和CPU的整体连线放在后面详细介绍。

3.1 寄存器

该寄存器代码可以用来生成所有的寄存器，为了模拟逻辑部件的输出延时，写数据设置有0.1个时间单位的延迟。读数据输出类型为reg，可以使输出一直可读，always语句每当时钟上跳沿的时候就会被触发，如果写信号有效，就会输入；该寄存器没有输出使能，输出是一直可读的。

```
module Reg(clk,data,Wr,Q);
    input clk, Wr;
    input [31:0] data;
    output reg [31:0] Q;

    always @(posedge clk ) begin
        #0.1
        if(Wr)
            Q <=  data;
    end
endmodule
```

3.2 通用寄存器堆RF

对于该通用寄存器堆的设计，有32个32位寄存器。只有写使能，没有输出使能。可以同时读两个寄存器，写一个寄存器。
对于其时序逻辑，该RF任何时候都可以读，时钟上跳沿来写数据。
使用 `reg [31:0] RF [31:0]` 来存储通用寄存器堆中的数据；输出使用的是assign赋值，输

出是一直有赋值的，尽管可能存在不需要该输出的情况

```
module RF(R_Reg1,R_Reg2,W_Reg,W_data,R_data1,R_data2,clk,RegWr);
    input [4:0] R_Reg1, R_Reg2, W_Reg;
    input [31:0] W_data;
    input clk, RegWr;
    output [31:0] R_data1, R_data2;

    reg [31:0] RF [31:0];//注意第二个是元素个数不是位宽，和数组下标一样，一共有32个通用寄存器

    assign #0.1 R_data1 = RF[R_Reg1];
    assign #0.1 R_data2 = RF[R_Reg2];

    always @(posedge clk ) begin
        if(RegWr)
            #0.1 RF[W_Reg] <= W_data;
    end
endmodule
```

3.3 ALU

ALU的输入端为ALU_a,ALU_b,输出端为ALU_c，通过ALUCtrl来控制ALU采用何种操作，同时还有Zero和O输出线。
ALU可以实现的操作有AND,OR,ADD,SUB,NOR,ADDU(不判断溢出的加法)。
对于ALU的内部实现，这里并没有用门电路，而是直接给出了输入输出的逻辑定义，节选代码如下：前面的always块是alu的主要功能，进行运算；后面的always块是对Zero信号线进行赋值。

```
always @(ALU_a or ALU_b or ALUCtrl) begin
    #0.1
    case(ALUCtrl)
        AND: ALU_c <= ALU_a & ALU_b;
        OR: ALU_c <= ALU_a | ALU_b;
        ADD: {O,ALU_c} <= ALU_a + ALU_b;
        SUB: {O,ALU_c} <= ALU_a - ALU_b;
        NOR: ALU_c <= ~ (ALU_a | ALU_b);
        ADDU: ALU_c <= ALU_a + ALU_b;
    endcase
end

always @(ALU_c) begin
    if (ALU_c == 0)
        Zero <= 1;
    else
        Zero <= 0;
end
```

3.4 ALU_CU

ALU_CU是ALU的控制单元，接受CU的ALUOp输入，和指令的func作为输入，产生ALUCtrl以控制ALU，对于单周期和多周期的cpu，ALU的设计是相同的。

```
module ALU_CU (ALUOp,func,ALUCtrl);
input [5:0] func;
input [1:0] ALUOp;
output reg [2:0] ALUCtrl;

always @(func or ALUOp) begin
    #0.1
    casex(ALUOp)
        2'b00: ALUCtrl <= 100;
        2'b01: ALUCtrl <= 110;
        2'b1x: begin
          case (func)
            6'b100000: ALUCtrl <= 100;
            6'b100001: ALUCtrl <= 101;
            6'b100010: ALUCtrl <= 110;
            6'b100100: ALUCtrl <= 000;
            6'b100101: ALUCtrl <= 001;
            6'b101010: ALUCtrl <= 011;
          endcase
        end
    endcase
end
endmodule
```

3.5 多路选择器

对于多路选择器，就设计的为纯组合逻辑设备，没有存储单元。然后还加入了parameter width的参数，从而使这个多路选择器可以改变输入端的线的宽度。同理，还有四路多路选择器，但是由于原理差不多，代码就不附上了。

```
module mux2(in0,in1,out,addr);
parameter width = 31;
input [width:0] in0, in1;
input addr;
output [width:0] out;

assign #0.1 out = addr? in1 : in0;

endmodule
```

3.6 符号扩展

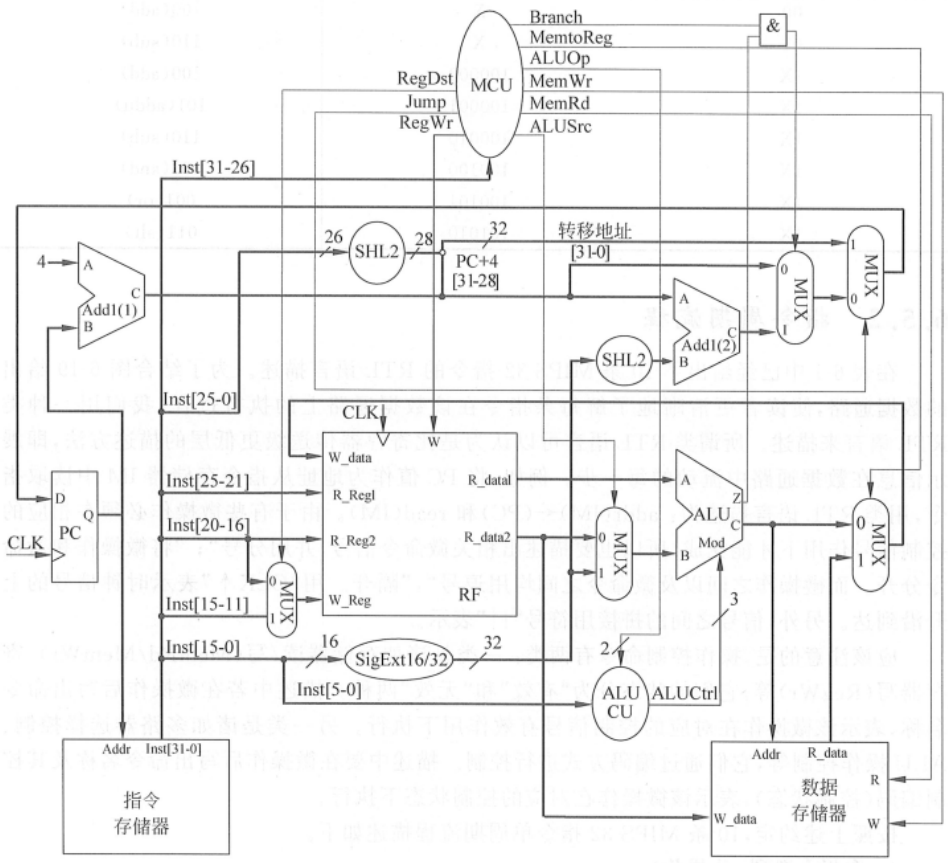
对于符号扩展模块，关键的就是怎么识别符号，而输入的最高位就一定是符号位，把该符号位进行扩展，然后和低位直接进行拼接，就得到符号扩展

```
module SigExt(in,out);
input wire signed [15:0] in;
output wire signed [31:0] out;

assign #0.1 out = {{16{in[15]}}, in};

endmodule
```

其他的逻辑模块还有加法器，由于实现原理很简单（只是ALU的一个小功能单独拿出来，且不涉及控制信号），就在此不再做详述了

四、单周期cpu设计

单周期cpu数据通路如下图所示：



本单周期cpu支持11条MIPS32指令，除了基本的10条指令外，还支持指令 `addiu` ，该指令为I型指令，具体格式为 `addiu Rt Rs Imm16` ，作用为将(Rs)+Imm16赋值给Rt。对于单周期CPU，需要在一个时钟周期内产生所有需要的微操作控制信号，微操作控制信号的产生和书上设计的相同，直接在时钟上跳沿到达的时候，给需要赋值的信号赋值即可。其对应的操作码和设计的主控制单元真值表为：

指令	操作码	RegDst	RegWr	ALUSrc	MemRd	MemWr	MemtoReg	Branch	Jump	ALUOp
R-类型	000000	1	1	0	0	0	0	0	0	10
lw	100011	0	1	1	1	0	1	0	0	00
sw	101011	x	0	1	0	1	x	0	0	00
beq	000100	x	0	0	0	0	x	1	0	01
j	000010	x	0	x	0	0	x	0	1	xx
addiu	001001	0	1	1	0	0	0	0	0	00

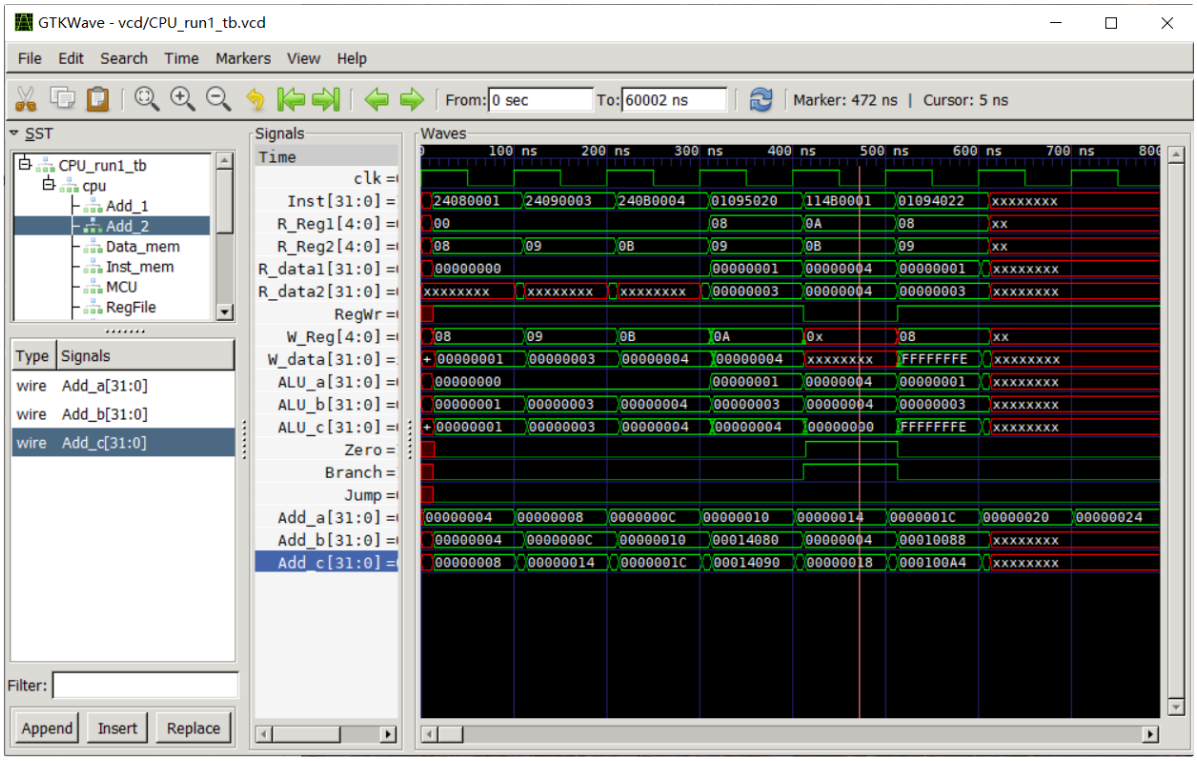
该单周期cpu试运行的MIPS32汇编程序如下：

这里试运行和分析的是第二部分程序设计的测试代码1：该测试程序的作用是首先给$t0置初值1，$t1置初值3，$t3置初值4，然后把$t0+$t1赋值给$t2，接着用beq指令，比较$t2和$t3，如果相等则跳转到next。

首先想简述一下testbench的写法，因为这里涉及到给指令存储器，数据存储器赋初值的问题，摘要testbench的代码如下。其中readmemh就是给指令存储器和数据存储器赋初值，分别把由MARS翻译的测试程序1的代码赋给指令存储器，和把自定义的RF寄存器初值赋值给RF。并且这里涉及到机器如何启动的问题，也就是说对于pc，应该有一个正确的初始值，在这里是指令寄存器的首地址，应当从该首地址来读取数据，对应的代码为 `#0.2 cpu.pc.Q <= 32'h0;` 之所以有一个#0.2个时间单位的延时，是因为开始设置的clk即为上跳沿，如果开始的时候立刻赋值，pc的值就会立刻改变，所以稍稍比开始时间延后一点再进行赋值，赋值后就可以执行程序了。

```
module CPU_run1_tb;
    reg clk;
    parameter clk_period = 10;

    initial begin
        $dumpfile("../vcd/CPU_run1_tb.vcd");
        $dumpvars(0, CPU_run1_tb);
        $readmemh("../file/run1.txt",cpu.Inst_mem.mem);//设置指令存储器
初值
        $readmemh("../file/run1_RF.txt",cpu.RegFile.RF);//设置RF初
值,$0寄存器的值恒为0

        clk <= 1 ;
        #0.2 cpu.pc.Q <= 32'h0;

        #6000 $finish;
    end
    always #(clk_period/2) clk = ~clk;
    CPU cpu(.clk(clk));
endmodule
```

下图为实验结果，下面对他进行简要的分析



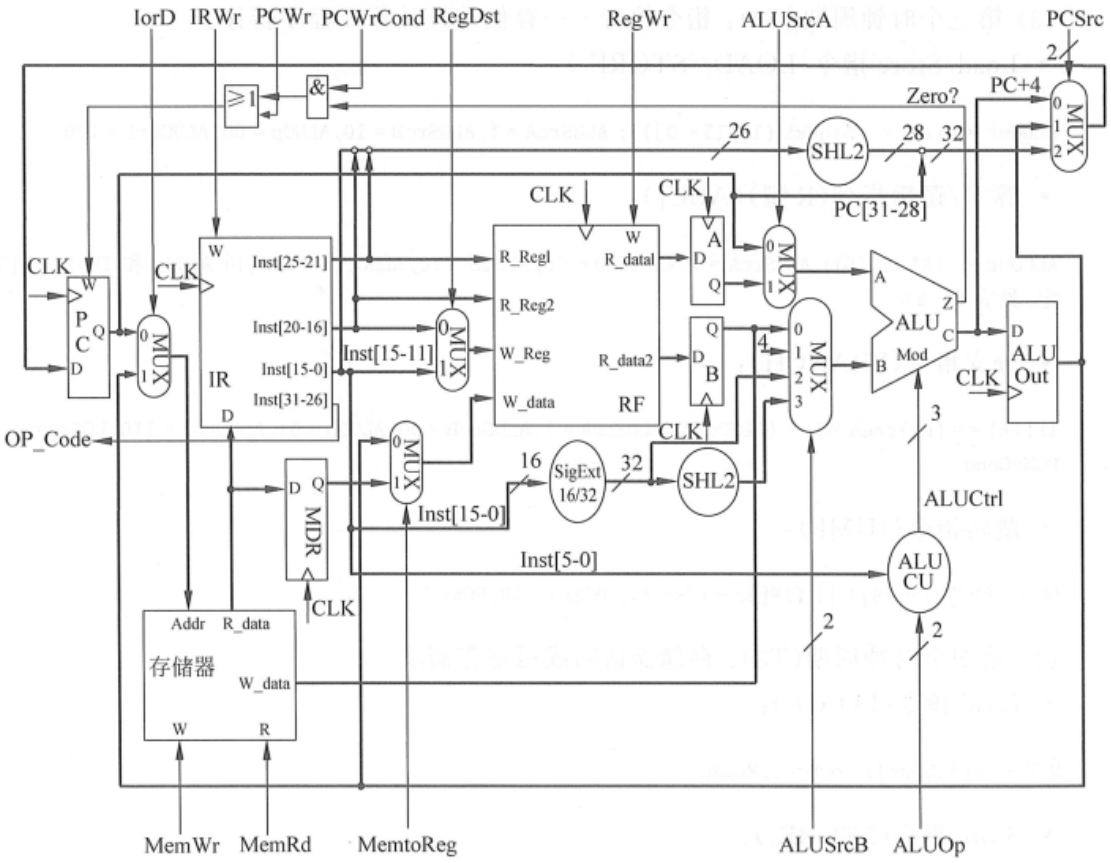
在给每个寄存器赋初值的时候，对于addiu指令，$0送到alu_a,也就是32'b0，立即数根据ALUSrc=1,送到alu_b。随后alu的运算结果，在MemtoReg=0的情况下就会送回RF堆的写数据端，W_Reg在RegDst=0的情况下，就会送入正确的通用寄存器地址。

在执行beq指令的情况下，alu_a送入$t2的值，alu_b送入$t3的值，然后alu执行减法运算，置Zero=1，从而在Zero和Branch都为1的情况下，pc+偏移量的计算结果会被送到pc的输入端，从而在下一个时钟周期可以实现更新。

五、组合逻辑多周期cpu设计

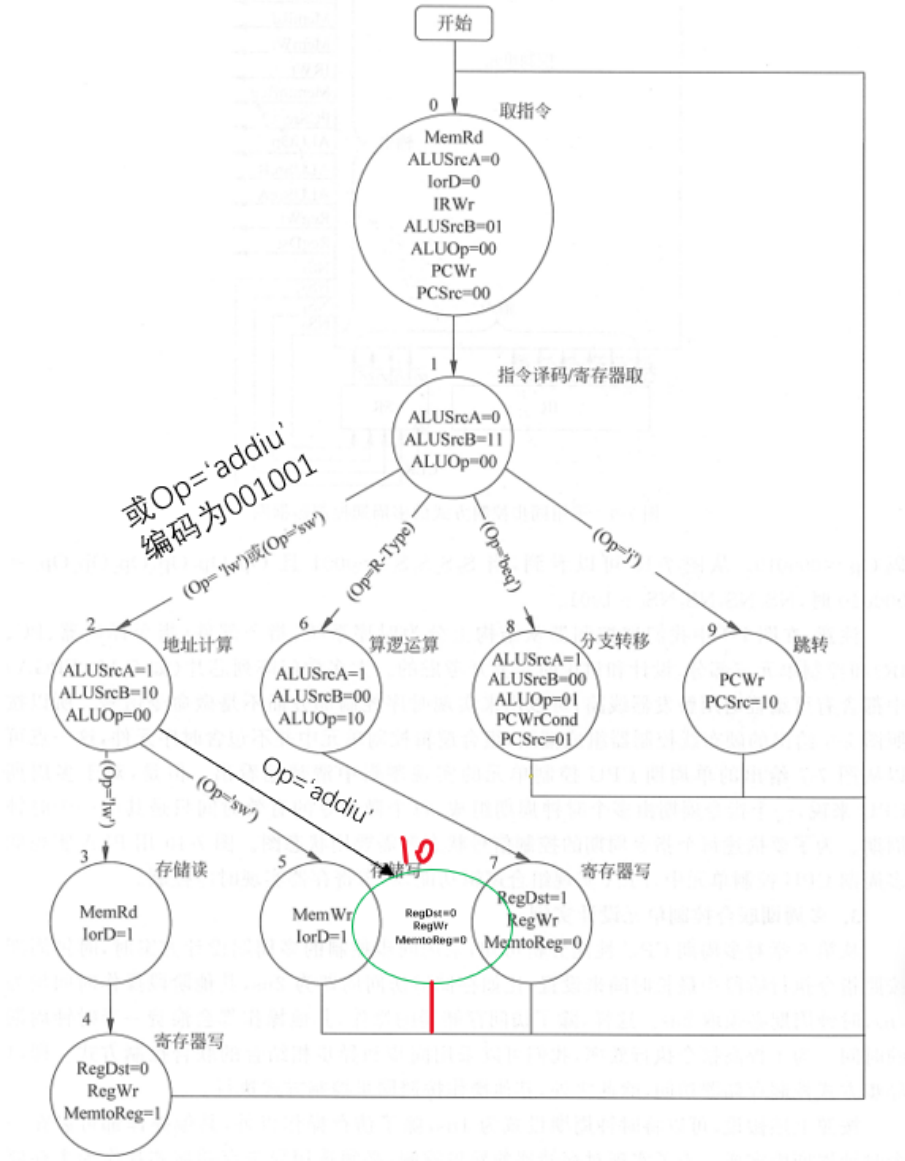
多周期cpu共支持11条指令，对于组合逻辑和微程序CPU都是如此，都在原来的基础上，增添了addiu指令。

首先给出多周期CPU的数据通路如下图所示



对于组合逻辑和微程序控制的多周期CPU，其数据通路都如上图所示。唯一不同的在CU。而对于多周期cpu的控制单元，微操作控制信号的变化本质上是一个有限状态转化图，所以就需要能够合理的表示现在的状态和转移到下一个状态，由于新增了一个指令addiu，相应的，对于该指令的状态转化，就会使有限状态转化图发生变化，下面首先给出对于组合逻辑CPU和微程序控制CPU所共有的状态转化图。

其中红色的部分使我自己添加的部分，因为addiu指令需要在第三个时钟周期把寄存器内容和立即数送到ALU中，所以在状态1之后需要转化到状态2，但是随后又需要写入寄存器，并且写入寄存器来源的数和add等单纯寄存器加法不一样（RegDst=0而不是1，因为addiu是一条I型指令，不是R型指令），所以需要新设一个状态，在状态2之后，跟随addiu指令转化到状态10



接下来详述组合逻辑CU的设计：

对于组合逻辑单元，本质上来说，给定输入的opcode，目前的状态以及需要的输出，就能得到一张真值表。但是在书写程序的时候，为了组合逻辑表达式更好理解，可以用一些位来表示当前的状态，用一些位来表示opcode。如下代码所示：组合逻辑中间变量 `P[0]-P[9]和P[19]` 分别表达有限状态转化图中的状态1-10

```
//组合逻辑上方的与门
//P[0]-P[9]是代表0-9的状态
assign #0.1 P[0] = ~S[3] && ~S[2] && ~S[1] && ~S[0]; //0000
assign #0.1 P[1] = ~S[3] && ~S[2] && ~S[1] &&  S[0]; //0001
assign #0.1 P[2] = ~S[3] && ~S[2] &&  S[1] && ~S[0]; //0010
assign #0.1 P[3] = ~S[3] && ~S[2] &&  S[1] &&  S[0]; //0011
assign #0.1 P[4] = ~S[3] &&  S[2] && ~S[1] && ~S[0]; //0100
assign #0.1 P[5] = ~S[3] &&  S[2] && ~S[1] &&  S[0]; //0101
assign #0.1 P[6] = ~S[3] &&  S[2] &&  S[1] && ~S[0]; //0110
assign #0.1 P[7] = ~S[3] &&  S[2] &&  S[1] &&  S[0]; //0111
assign #0.1 P[8] =  S[3] && ~S[2] && ~S[1] && ~S[0]; //1000
assign #0.1 P[9] =  S[3] && ~S[2] && ~S[1] &&  S[0]; //1001
assign #0.1 P[19] = S[3] && ~S[2] &&  S[1] && ~S[0]; //1010
```

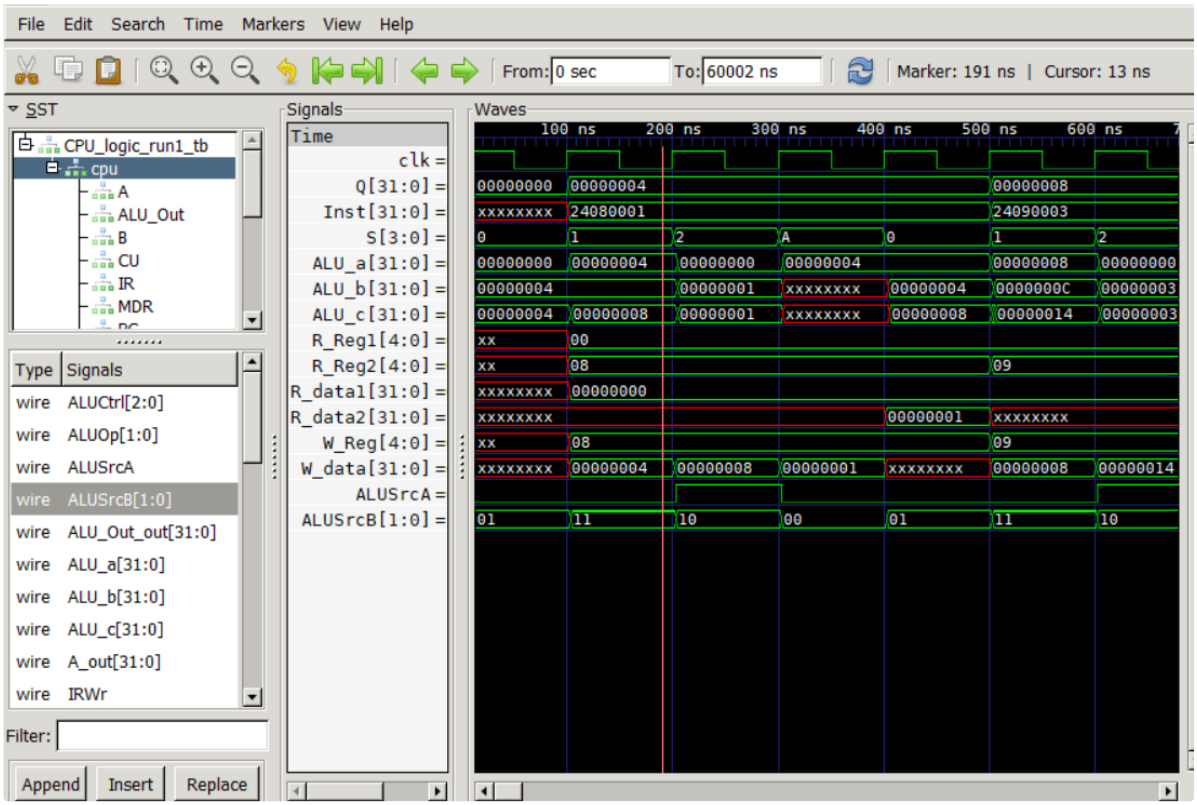
```
    //下面的是代表什么情况下有分支的情况下
    assign #0.1 P[10] = ~Op[5] && ~Op[4] && ~Op[3] && ~Op[2] &&
Op[1] && ~Op[0] && ~S[3] && ~S[2] && ~S[1] &&  S[0]; //000010 0001 →
j 0001
    assign #0.1 P[11] = ~Op[5] && ~Op[4] && ~Op[3] &&  Op[2] &&
~Op[1] && ~Op[0] && ~S[3] && ~S[2] && ~S[1] &&  S[0]; //000100 0001 -
> beq 0001
    assign #0.1 P[12] = ~Op[5] && ~Op[4] && ~Op[3] && ~Op[2] &&
~Op[1] && ~Op[0] && ~S[3] && ~S[2] && ~S[1] &&  S[0]; //000000 0001 -
>R_type 0001
    assign #0.1 P[13] =  Op[5] && ~Op[4] && ~Op[3] && ~Op[2] &&
Op[1] &&  Op[0] && ~S[3] && ~S[2] &&  S[1] && ~S[0]; //100011 0010 -
>lw 0010
    assign #0.1 P[14] =  Op[5] && ~Op[4] &&  Op[3] && ~Op[2] &&
Op[1] &&  Op[0] && ~S[3] && ~S[2] && ~S[1] &&  S[0]; //101011 0001 -
>sw 0001
    assign #0.1 P[15] =  Op[5] && ~Op[4] && ~Op[3] && ~Op[2] &&
Op[1] &&  Op[0] && ~S[3] && ~S[2] && ~S[1] &&  S[0]; //100011 0001 -
>lw 0001
    assign #0.1 P[16] =  Op[5] && ~Op[4] &&  Op[3] && ~Op[2] &&
Op[1] &&  Op[0] && ~S[3] && ~S[2] &&  S[1] && ~S[0]; //101011 0010 -
>sw 0010
    //addiu指令新增的
    assign #0.1 P[17] = ~Op[5] && ~Op[4] &&  Op[3] && ~Op[2] &&
~Op[1] &&  Op[0] && ~S[3] && ~S[2] && ~S[1] &&  S[0]; //001001 0001 -
>addiu 0001
    assign #0.1 P[18] = ~Op[5] && ~Op[4] &&  Op[3] && ~Op[2] &&
~Op[1] &&  Op[0] && ~S[3] && ~S[2] &&  S[1] && ~S[0]; //001001 0010 -
>addiu 0010
```

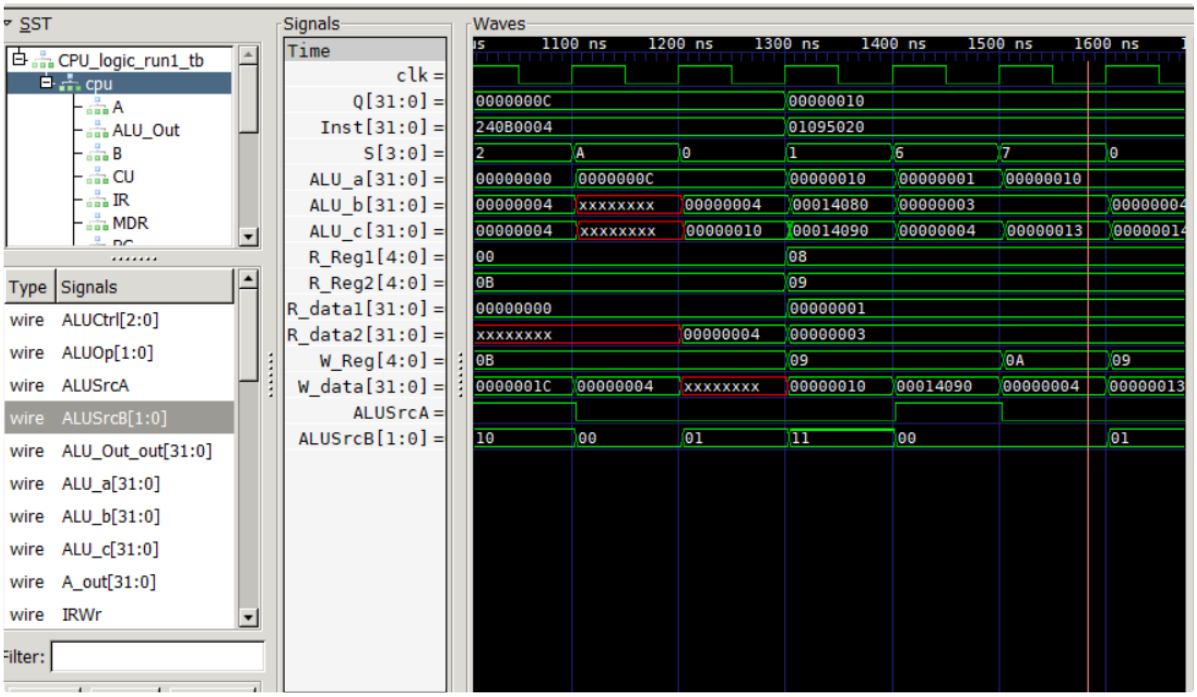
对于输出，微操作控制信号的产生只和当前所处的状态有关，因此只需要`P[0]-P[9]和P[19]`的或运算；而对于下一个状态的NS，他的产生不仅和当前状态有关，而且和目前的指令有关。所以对于会产生的分支的状态，使用一个逻辑符号来标识他下一步对应的转移，为`P[10]-P[18]`，如`P[10]`为真当且仅当输出的操作码为000010且当前状态为0001，也就是说，`P[10]`为真当且仅当输入的指令为jump，且位于状态1上，那么他下一步就要转化到状态9。NS的赋值就是通过`P[0]-P[9]和P[19]`以及`P[10]-P[18]`来完成的。
最后对于该组合逻辑，需要一个寄存器来存储当前的状态，每当一个时钟上跳沿到达的时候，将NS存到寄存器S中。

```
    //组合逻辑下方或门，这些
    assign #0.1 PCWr = P[0] || P[9];
    assign #0.1 PCWrCond = P[8];
    assign #0.1 IorD = P[3] || P[5];
    assign #0.1 MemRd = P[0] || P[3];
    assign #0.1 MemWr =P[5];
    assign #0.1 IRWr = P[0];
    assign #0.1 MemtoReg = P[4];
```

```
    assign #0.1 PCSrc = {P[9],P[8]};
    assign #0.1 ALUOp = {P[6],P[8]};
    assign #0.1 ALUSrcB = {P[1]||P[2],P[0]||P[1]};
    assign #0.1 ALUSrcA = P[2] || P[6] || P[8];
    // assign #0.1 RegWr = P[4] || P[7];
    assign #0.1 RegWr = P[4] || P[7] || P[19];
    assign #0.1 RegDst = P[7];
    // assign #0.1 NS[3] = P[10] || P[11];
    // assign #0.1 NS[2] = P[3] || P[6] || P[12] || P[16];
    // assign #0.1 NS[1] = P[6] || P[12] || P[13] || P[14] || P[15];
    // assign #0.1 NS[0] = P[0] || P[6] || P[10] || P[13] || P[16];
    assign #0.1 NS[3] = P[10] || P[11] || P[18];
    assign #0.1 NS[2] = P[3] || P[6] || P[12] || P[16];
    assign #0.1 NS[1] = P[6] || P[12] || P[13] || P[14] || P[15] ||
P[17] || P[18];
    assign #0.1 NS[0] = P[0] || P[6] || P[10] || P[13] || P[16];

    always @(posedge clk) begin
        #0.1
        S <= NS;
    end
```

然后同样的运行测试程序1可以得到如下波形图：





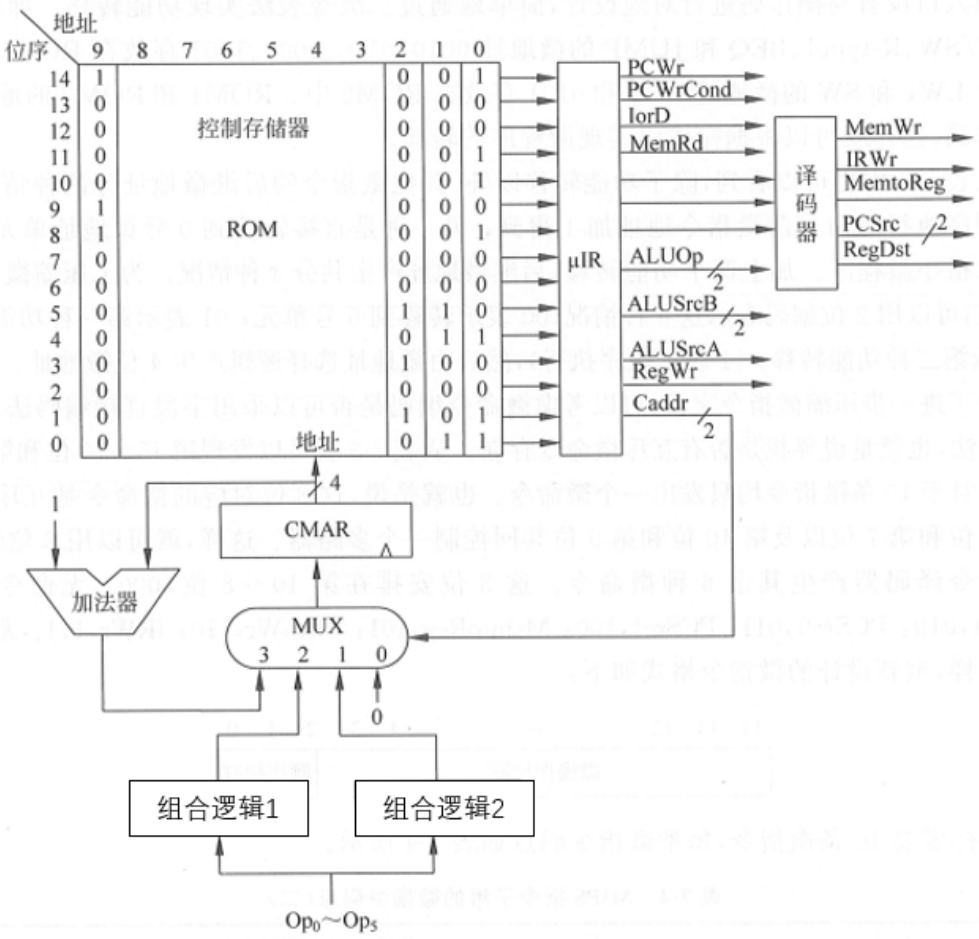
六、微程序多周期CPU设计

对于微程序多周期CPU，在设计的数据通路上和组合逻辑多周期CPU的相同；而在例如寄存器，RF堆这样的通用部件上，设计和通用的组合逻辑模块也相同。唯一有区别的就是CU，采用微程序控制的方法设计。下面简述该CU的实现方法

首先给出rom中存储的微指令如下所示，14-2位为微操作控制字段，0-1位为下地址字段

	微指令地址	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fetch1	0000	1	0	0	1	1	1	0	0	0	0	1	0	0	1	1
Fetch2	0001	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1
LW/SW	0010	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0
LW1	0011	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1
LW2	0100	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
SW	0101	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0
R-type1	0110	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1
R-type2	0111	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
BEQ	1000	0	1	0	0	0	1	0	0	1	0	0	1	0	0	0
JUMP	1001	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0
Addiu	1010	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

而在生成转移地址的时候，相比书上使用输入的opcode来进行rom查表来生成转移地址做了改进，直接由组合逻辑生成下一步的后继微地址，具体逻辑图如下图所示。



下面给出微程序多周期cpu控制单元CU的具体设计代码，设置rom来存放微指令，使用CMAR来存放要取到的rom指令的地址，2-14位表示要输出的微操作控制命令（其中一部分需要译码），0-1位来表示接下来怎么转移。00为跳转到状态0开始执行，01为按照第一种方式进行地址转移，10为按照第二种方式进行地址转移，11为顺序执行。

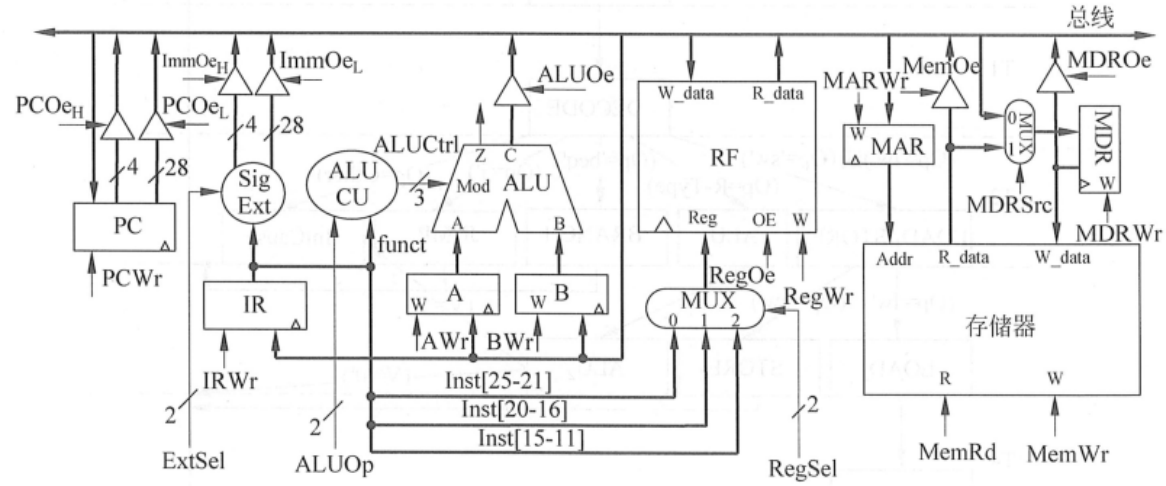
```
module
CU_micro(PCWr,PCWrCond,IorD,MemRd,MemWr,IRWr,MemtoReg,PCSrc,ALUOp,ALU
SrcB,ALUSrcA,RegWr,RegDst,Op,clk);
    input [5:0] Op;
    input clk;
    output
PCWr,PCWrCond,IorD,MemRd,MemWr,IRWr,MemtoReg,ALUSrcA,RegWr,RegDst;
    output [1:0] PCSrc;
    output [1:0] ALUOp;
    output [1:0] ALUSrcB;

    //内部的连线和部件
    wire [3:0] mux_out;//多路选择器输出
    wire [1:0] mux_addr;//选择mux的地址
    wire [2:0] decode;//译码器的输入
    wire [3:0] addr1; //转移使用组合逻辑输出线
    wire [3:0] addr2;
    wire j,beq,R_type,lw,sw,addiu;//组合逻辑的中间变量，判断输入的Opcode是
什么
    //00为转移到0号单元，01为组合逻辑1转移，10为组合逻辑2转移，11为顺序执行
```

```
    reg [3:0] CMAR;//用来存储当前地址的位置，clk来的时候更新
    reg [14:0] rom [10:0]; //用来存放微指令


    initial begin
        $readmemb("../file/rom.txt",rom,0,10);
    end

    //rom输出
    assign #0.1
{PCWr,PCWrCond,IorD,MemRd,decode,ALUOp,ALUSrcB,ALUSrcA,RegWr,mux_addr
} = rom[CMAR];
    //译码器
    assign #0.1 RegDst = (decode==3'b001)? 1 : 0;
    assign #0.1 PCSrc[0] = (decode==3'b010)? 1 : 0;
    assign #0.1 PCSrc[1] = (decode==3'b011)? 1 : 0;
    assign #0.1 MemtoReg = (decode==3'b100)? 1 : 0;
    assign #0.1 MemWr = (decode==3'b101)? 1 : 0;
    assign #0.1 IRWr = (decode==3'b110)? 1 : 0;

    //先计算组合逻辑中间变量，判断输入的是什么
    assign #0.1 R_type = ~Op[5] && ~Op[4] && ~Op[3] && ~Op[2] &&
~Op[1] && ~Op[0];
    assign #0.1 j =      ~Op[5] && ~Op[4] && ~Op[3] && ~Op[2] &&
Op[1] && ~Op[0];
    assign #0.1 beq =    ~Op[5] && ~Op[4] && ~Op[3] &&  Op[2] &&
~Op[1] && ~Op[0];
    assign #0.1 lw =      Op[5] && ~Op[4] && ~Op[3] && ~Op[2] &&
Op[1] &&  Op[0];
    assign #0.1 sw =      Op[5] && ~Op[4] &&  Op[3] && ~Op[2] &&
Op[1] &&  Op[0];
    assign #0.1 addiu  = ~Op[5] && ~Op[4] &&  Op[3] && ~Op[2] &&
~Op[1] &&  Op[0];
    //addr1组合逻辑
    assign #0.1 addr1[3] = j || beq;
    assign #0.1 addr1[2] = R_type;
    assign #0.1 addr1[1] = R_type || lw || sw || addiu;
    assign #0.1 addr1[0] = j;
    //addr2组合逻辑
    assign #0.1 addr2[3] = addiu;
    assign #0.1 addr2[2] = sw;
    assign #0.1 addr2[1] = lw || addiu;
    assign #0.1 addr2[0] = lw || sw;

    //多路选择器输出mux_addr
```

```
    mux4 #(3)
MUX(.in0(4'b0),.in1(addr1),.in2(addr2),.in3(CMAR+4'b1),.out(mux_out),
.addr(mux_addr));

    always @(posedge clk) begin //时钟信号到达更新地址
        CMAR <= mux_out;
    end

endmodule
```

七、单总线CPU的大概设计

对于单总线cpu，数据通路如下图所示



可以看到单总线相比于前面设计的分散互联多周期CPU，多加入的模块有三态控制器，用来控制是否允许到总线上的输出，对于其他的部件，基本上是和分散互联相似的，对于寄存器多了写使能；而另一比较大的改变是在状态转换图上。所以因为时间有限，这里只画出了多周期总线结构的状态转换图，和实现了三态控制器的代码。对于具体的单总线多周期的连线并没有连接和调试

单总线多周期cpu的状态转化图如下图所示：

三态控制器模块的实现如下图所示：

```
module Ctrl(in,out,Oe);
input [31:0] in;
input Oe;
output [31:0] out;

assign #0.1 out = Oe? in : 32'zzzz;

endmodule
```