目　录

一、 设计目的与目标

1.1 实验内容

1） 本实例所设计 CPU 的指令格式的拟定；

2） 基本功能部件的设计与实现；

3） CPU 各主要功能部件的设计与实现；

4） CPU 的封装；

5） 对各个单元组合而成的 CPU 进行指令测试，配合使用模拟仿真，了解指令和数据在各个单元中的传输过程及方向。

1.2 实验要求

1) 至少支持 add、sub、and、or、addi、andi、ori、lw、sw、beq、bne 和 j 十二条指令。

二、 课程设计器材

2.1 硬件平台

无

2.2 软件平台

1） 操作系统：Win 10。

2） 开发平台：Vivado 2017.2。

3） 编程语言：VerilogHDL 硬件描述语言。

三、 CPU 逻辑设计总体方案

单周期 CPU 可以看成由数据通路和和控制部件两大部分组成。数据通路是指在指令执行过程中，数据所经过的路径和路径上所涉及的功能部件。而控制部件则根据每条指令的不同功能，生成对不同数据通路的不同控制信号，正确地控制指令的执行流程。

因此，要设计处理器，首先需要确定处理器的指令集和指令编码，然后确定每条指令的数据通路，最后确定数据通路的控制信号。

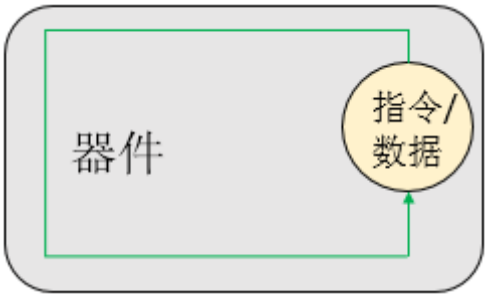


图 3-0 CPU 宏观设计方案

3.1 指令模块

单周期（Single Cycle）CPU 是指 CPU 从取出 1 条指令到执行完该指令只需 1 个时钟周期。

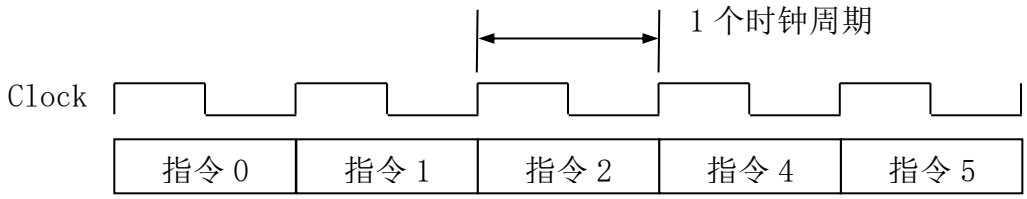


图 3-1 时钟周期和单周期 CPU 指令的执行

一条指令的执行过程包括：取指令→分析指令→执行指令→保存结果（如果有的话）。对于单周期 CPU 来说，这些执行步骤均在一个时钟周期内完成。

3.1.1 MIPS 指令格式

MIPS 指令系统结构有 MIPS-32 和 MIPS-64 两种。本实验的 MIPS 指令选用 MIPS-32。以下所说的 MIPS 指令均指 MIPS-32。

MIPS 的指令格式为 32 位。图 3-3 给出了 MIPS 指令的 3 种格式。

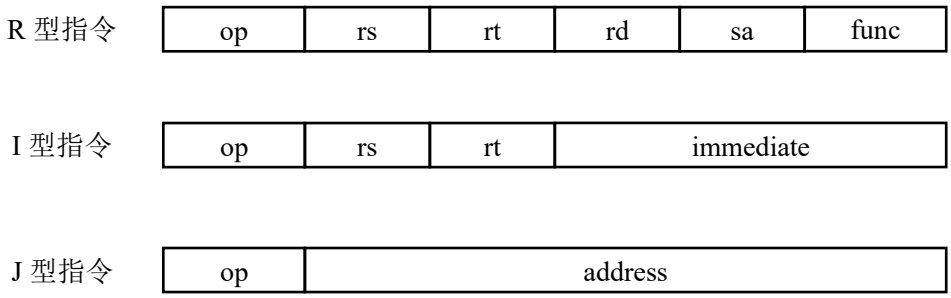


图 3-2 MIPS 指令格式

本实验只选取了 20 条典型的 MIPS 指令来描述 CPU 逻辑电路的设计方法。表 3-1 列出了本实验的所涉及到的 20 条 MIPS 指令。

R 型指令							
指令	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	功能
Add	000000	rs	rt	rd	000000	100000	寄存器加
Sub	000000	rs	rt	rd	000000	100010	寄存器减
And	000000	rs	rt	rd	000000	100100	寄存器与

Or	000000	rs	rt	rd	000000	100101	寄存器或
Xor	000000	rs	rt	rd	000000	100110	寄存器异或
Sll	000000	00000	rt	rd	sa	000000	左移
Srl	000000	00000	rt	rd	sa	000010	逻辑右移
Sra	000000	00000	rt	rd	sa	000011	算术右移
Jr	000000	rs	rt	rd	000000	001000	寄存器跳
I 型指令							
Addi	001000	rs	rt	immediate			立即数加
Andi	001100	rs	rt	immediate			立即数与
Ori	001101	rs	rt	immediate			立即数或
Xori	001110	rs	rt	immediate			立即数异或
Lw	100011	rs	rt	offset			取数据
Sw	101011	rs	rt	offset			存数据
Beq	000100	rs	rt	offset			相等转移
Bne	000101	rs	rt	offset			不等转移
Lui	001111	00000	rt	immediate			设置高位
J 型指令							
J	000010	address					跳转
Jal	000011	address					调用

表 1 本实验所涉及的 20 条 MIPS 指令

R 型指令的 op 均为 0，具体操作由 func 指定。rs 和 rt 是源寄存器号，rd 是目的寄存器号。移位指令中使用 sa 指定移位位数。

I 型指令的低 16 位是立即数，计算时需扩展到 32 位，依指令的不同需进行零扩展和符号扩展。

J 型指令的低 26 位是地址，是用于产生跳转的目标地址。

3.1.2 指令处理流程

一般来说，CPU 在处理指令时需要经过以下几个过程：

（1） 取指令(IF)：根据程序计数器 PC 中的指令地址，从指令存储器中取出一条指令，同时 PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 PC，当然得到的“地址”需要做些变换才送入 PC。

（2） 指令译码(ID)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，由指令的[15-12]位产生相应的操作控制信号，用于驱动执行状态中的各种操作。

（3） 指令执行(EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

（4） 存储器访问(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

（5） 结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

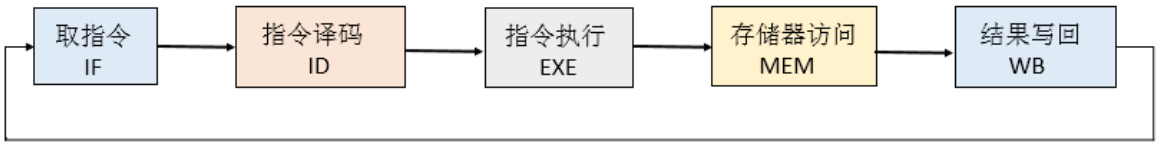


图 3-3 单周期 CPU 指令处理过程

3.2 数据通路

CPU 的电路包括数据路径（Data path）和控制部件（Control Unit）两大部分。下面先给出单周期 CPU 的总体设计图，再分别介绍每个路径和控制部件的设计。

3.2.1 总体结构图

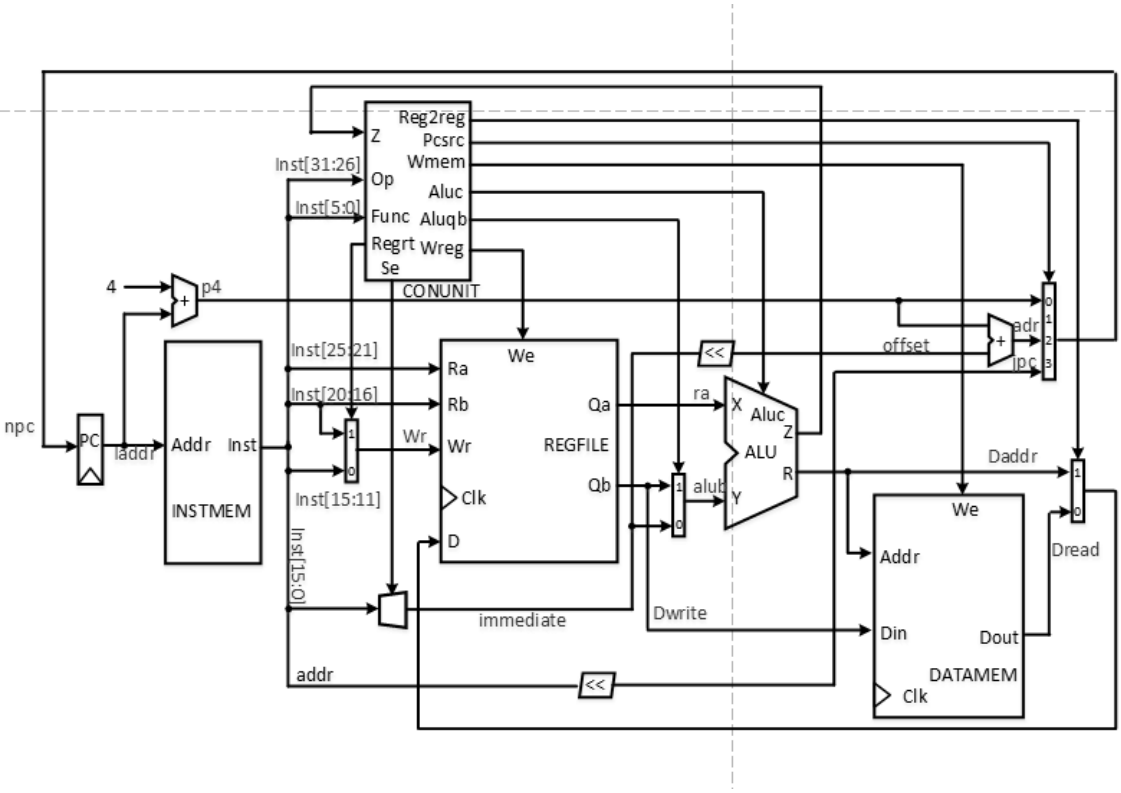


图 3-4 单周期 CPU 总体结构图

图 3-4 是一个简单的基本上能够在单周期上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令储存在指令储存器，数据储存在数据存储器。访问存储器时，先给出地址，然后由读/写信号控制。对于寄存器组，读操作时，先给出地址，输出端直接输出相应数据；而在写操作时，在 We 使能信号为 1 时，在时钟边沿触发写入。

3.2.2 设计流程逻辑图

根据实验原理中的单周期 CPU 总体结构图，我们可以清楚的知道单周期 CPU 的设计应包括 PC，PCAdd4，INSTMEM，CONUNIT，REGFILE，ALU，DATAMEM，EXT16T32 这几个核心模块，其中 PCAdd4 模块需要用到 32 位加法器 CLA_32。此外还需要左移处理模块 SHIFTER_COMBINATION,一个固定左移两位的移位器 SHIFT32_L2，一个四选一多路选择器 MUX4X32，两个 32 位二选一多路选择器 MUX2X32，一个 5 位二选一多路选择器 MUX2X5，一个数据扩展器 EXT16T32。其中为了运行整个 CPU 还需要加入一个顶层模块（SingleCycleCPU）来调用这些模块，所以自然地，这些模块为顶层模块的子模块。设计流程逻辑图如下（左边为拓展模块，右边为核心模块）。

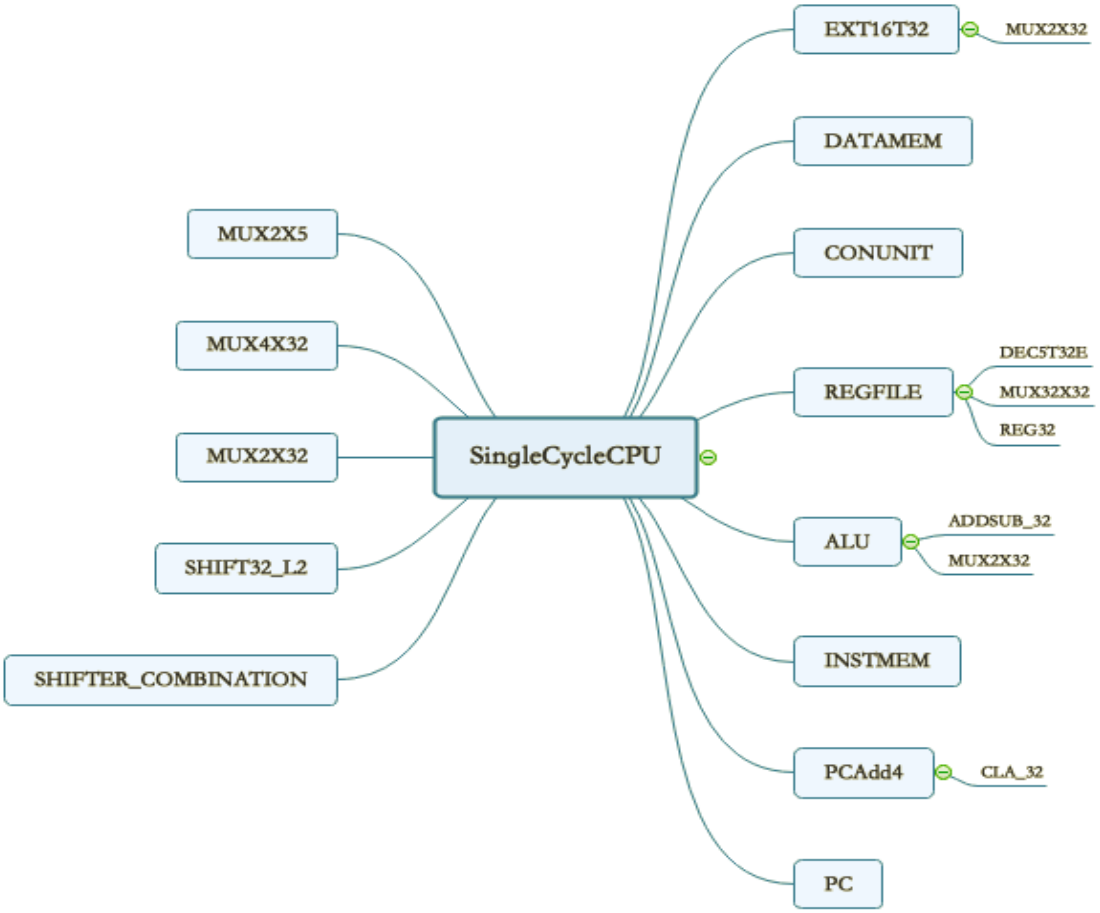
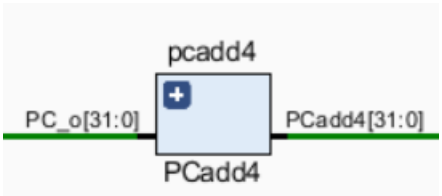


图 3-5 单周期 CPU 设计流程逻辑图

四、 模块详细设计

4.1 PCAdd4

1） 所处位置



2) 模块功能

作为PC寄存器的更新信号。

3) 实现思路

由于每条指令32位，所以增加一个32位加法器，固定与32位的立即数4进行相加，且得到的结果在当前时钟信号的上升沿更新进PC寄存器。

4) 引脚及控制信号

Addr:当前指令地址，输入端口

PCadd4:下一条指令地址，输出端口

5) 主要实现代码

1. PCAdd4

```
module PCadd4(PC_o,PCadd4);
input [31:0] PC_o;//偏移量
output [31:0] PCadd4;//新指令地址
CLA_32 cla32(PC_o,4,0, PCadd4, Cout);
endmodule
```

2. CLA_32

```
module CLA_32(X, Y, Cin, S, Cout);
input [31:0] X, Y;
input Cin;
output [31:0] S;
output Cout;
wire Cout0, Cout1, Cout2, Cout3, Cout4, Cout5, Cout6;
CLA_4 add0 (X[3:0], Y[3:0], Cin, S[3:0], Cout0);
CLA_4 add1 (X[7:4], Y[7:4], Cout0, S[7:4], Cout1);
CLA_4 add2 (X[11:8], Y[11:8], Cout1, S[11:8], Cout2);
CLA_4 add3 (X[15:12], Y[15:12], Cout2, S[15:12], Cout3);
CLA_4 add4 (X[19:16], Y[19:16], Cout3, S[19:16], Cout4);
CLA_4 add5 (X[23:20], Y[23:20], Cout4, S[23:20], Cout5);
CLA_4 add6 (X[27:24], Y[27:24], Cout5, S[27:24], Cout6);
CLA_4 add7 (X[31:28], Y[31:28], Cout6, S[31:28], Cout);
Endmodule
```

3. CLA_4

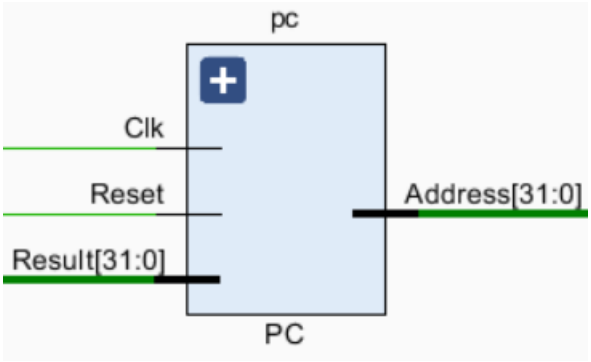
```
module CLA_4(X, Y, Cin, S, Cout);
input [3:0] X;
input [3:0] Y;
input Cin;
output [3:0] S;
output Cout;

and get_0_0_0(tmp_0_0_0, X[0], Y[0]);
or get_0_0_1(tmp_0_0_1, X[0], Y[0]);
```

```
    and get_0_1_0(tmp_0_1_0, X[1], Y[1]);
    or get_0_1_1(tmp_0_1_1, X[1], Y[1]);

    and get_0_2_0(tmp_0_2_0, X[2], Y[2]);
    or get_0_2_1(tmp_0_2_1, X[2], Y[2]);

    and get_0_3_0(tmp_0_3_0, X[3], Y[3]);
    or get_0_3_1(tmp_0_3_1, X[3], Y[3]);

    and get_1_0_0(tmp_1_0_0, ~tmp_0_0_0, tmp_0_0_1);
    xor getS0(S0, tmp_1_0_0, Cin);

    and get_1_1_0(tmp_1_1_0, ~tmp_0_1_0, tmp_0_1_1);
    not get_1_1_1(tmp_1_1_1, tmp_0_0_0);
    nand get_1_1_2(tmp_1_1_2, Cin, tmp_0_0_1);
    nand get_2_0_0(tmp_2_0_0, tmp_1_1_1, tmp_1_1_2);
    xor getS1(S1, tmp_1_1_0, tmp_2_0_0);

    and get_1_2_0(tmp_1_2_0, ~tmp_0_2_0, tmp_0_2_1);
    not get_1_2_1(tmp_1_2_1, tmp_0_1_0);
    nand get_1_2_2(tmp_1_2_2, tmp_0_1_1, tmp_0_0_0);
    nand get_1_2_3(tmp_1_2_3, tmp_0_1_1, tmp_0_0_1, Cin);
    nand get_2_1_0(tmp_2_1_0, tmp_1_2_1, tmp_1_2_2, tmp_1_2_3);
    xor getS2(S2, tmp_1_2_0, tmp_2_1_0);

    and get_1_3_0(tmp_1_3_0, ~tmp_0_3_0, tmp_0_3_1);
    not get_1_3_1(tmp_1_3_1, tmp_0_2_0);
    nand get_1_3_2(tmp_1_3_2, tmp_0_2_1, tmp_0_1_0);
    nand get_1_3_3(tmp_1_3_3, tmp_0_2_1, tmp_0_1_1, tmp_0_0_0);
    nand get_1_3_4(tmp_1_3_4, tmp_0_2_1, tmp_0_1_1, tmp_0_0_1,
Cin);
    nand get_2_2_0(tmp_2_2_0, tmp_1_3_1, tmp_1_3_2, tmp_1_3_3,
tmp_1_3_4);
    xor getS3(S3, tmp_1_3_0, tmp_2_2_0);

    not get_1_4_0(tmp_1_4_0, tmp_0_3_0);
    nand get_1_4_1(tmp_1_4_1, tmp_0_3_1, tmp_0_2_0);
    nand get_1_4_2(tmp_1_4_2, tmp_0_3_1, tmp_0_2_1, tmp_0_1_0);
    nand get_1_4_3(tmp_1_4_3, tmp_0_3_1, tmp_0_2_1, tmp_0_1_1,
tmp_0_0_0);
    nand get_1_4_4(tmp_1_4_4, tmp_0_3_1, tmp_0_2_1, tmp_0_1_1,
tmp_0_0_1, Cin);
    nand   getCout(Cout,   tmp_1_4_0,   tmp_1_4_1,   tmp_1_4_2,
```

```
tmp_1_4_3,tmp_1_4_4);

    assign S = {S3,S2,S1,S0};

    endmodule
```

4.2 PC

6) 所处位置



7) 模块功能

用于给出指令在指令储存器中的地址。

8) 实现思路

为实现稳定输出，在时钟信号的上升沿更新，而且需要一个控制信号，在控制信号为 0 的时候初始化 PC 寄存器，即全部置零。

9) 引脚及控制信号

Clk：时钟周期，输入信号

Reset：控制信号，输入信号

Result 目标地址，可能是跳转地址或者是下一条指令的地址，输入信号

Addr：指令地址，输出信号

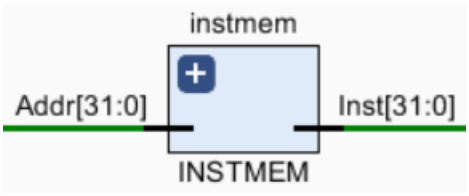
10) 主要实现代码

```
module PC(Clk,Reset,Result,Address);
input Clk;//时钟
input Reset;//是否重置地址。0-初始化 PC，否则接受新地址
input[31:0] Result;
output reg[31:0] Address;
//reg[31:0] Address;
initial begin
Address  <= 0;
end
always @(posedge Clk or negedge Reset)
```

```
begin
if (!Reset) //如果为 0 则初始化 PC，否则接受新地址
begin
Address <= 0;
end
else
begin
Address =  Result;
end
end
endmodule
```

4.3 INSTMEM

11) 所处位置



12) 模块功能

依据当前 pc，读取指令寄存器中相对应地址 Addr[6:2]的指令。

13) 实现思路

将 pc 的输入作为敏感变量，当 pc 发生改变的时候，则进行指令的读取，根据相关的地址，输出指令寄存器中相对应的指令，且在设计指令的时候，要用到 12 条给出的指令且尽量合理。

14) 引脚及控制信号

Addr：指令地址，输入信号

Inst：指令编码，输出信号

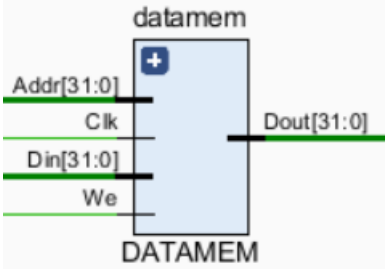
15) 主要实现代码

```
module INSTMEM(Addr,Inst);//指令存储器
input[31:0]Addr;
//input InsMemRW;//状态为'0'，写指令寄存器，否则为读指令寄存器
output[31:0]Inst;
wire[7:0]Rom[31:0];
assign Rom[5'h00]=32'h20010008;//addi $1,$0,8 $1=8
assign Rom[5'h01]=32'h3402000C;//ori $2,$0,12 $2=12
assign Rom[5'h02]=32'h00221820;//add $3,$1,$2 $3=20
assign Rom[5'h03]=32'h00412022;//sub $4,$2,$1 $4=4
```

```
    assign Rom[5'h04]=32'h00222824;//and $5,$1,$2
    assign Rom[5'h05]=32'h00223025;//or  $6,$1,$2
    assign Rom[5'h06]=32'h14220002;//bne $1,$2,2
    assign Rom[5'h07]=32'hXXXXXXXX;
    assign Rom[5'h08]=32'hXXXXXXXX;
    assign Rom[5'h09]=32'h10220002;// beq $1,$2,2
    assign Rom[5'h0A]=32'h0800000D;// J 0D
    assign Rom[5'h0B]=32'hXXXXXXXX;
    assign Rom[5'h0C]=32'hXXXXXXXX;
    assign Rom[5'h0D]=32'hAD02000A;// sw $2 10($8) memory[$8+10]=12
    assign Rom[5'h0E]=32'h8D04000A;//lw $4 10($8) $4=12
    assign Rom[5'h0F]=32'h10440003;//beq $2,$4,3
    assign Rom[5'h10]=32'hXXXXXXXX;
    assign Rom[5'h11]=32'hXXXXXXXX;
    assign Rom[5'h12]=32'hXXXXXXXX;
    assign Rom[5'h13]=32'h30470009;//andi $2,9,$7
    assign Rom[5'h14]=32'hXXXXXXXX;
    assign Rom[5'h15]=32'hXXXXXXXX;
    assign Rom[5'h16]=32'hXXXXXXXX;
    assign Rom[5'h17]=32'hXXXXXXXX;
    assign Rom[5'h18]=32'hXXXXXXXX;
    assign Rom[5'h19]=32'hXXXXXXXX;
    assign Rom[5'h1A]=32'hXXXXXXXX;
    assign Rom[5'h1B]=32'hXXXXXXXX;
    assign Rom[5'h1C]=32'hXXXXXXXX;
    assign Rom[5'h1D]=32'hXXXXXXXX;
    assign Rom[5'h1E]=32'hXXXXXXXX;
    assign Rom[5'h1F]=32'hXXXXXXXX;
    assign Inst=Rom[Addr[6:2]];
endmodule
```

4.4 DATAMEM

16) 所处位置



17) 模块功能

数据存储器，通过控制信号，对数据寄存器进行读或者写操作，并且此处模块额外合并了输出 DB 的数据选择器，此模块同时输出写回寄存器组的数据 DB。

18) 实现思路

由于需要支持取数/存数指令，所以要在指令储存器的基础上增加写入数据的数据写入端口，写使能信号。又因为写操作在时钟信号的上升沿，所以要增加时钟信号。

19) 引脚及控制信号

引脚	控制信号	作用	状态“0”	状态“1”
Addr（输入）	R	访存地址		
Din（输入）	Qb	输入的值		
Clk（输入）	Clk	时钟周期		
We（输入）	Wmem	写使能信号	信号无效	信号有效
Dout（输出）	Dout	读取的值		

当 We 为 1 时，进行 sw 指令操作，此时 Din 端口输入信号实际为 rt，Addr 端口输入信号为 rs 和偏移量相加的地址，在时钟周期上升沿将 rt 的值写入改地址的储存单元。

当 We 为 0 时，进行 lw 指令操作，此时 Addr 端口输入信号为 rs 和偏移量相加的地址，Dout 为读取该地址储存器的内容。

20) 主要实现代码

```
module DATAMEM(Addr,Din,Clk,We,Dout);
input[31:0]Addr,Din;
input Clk,We;
output[31:0]Dout;
reg[31:0]Ram[31:0];
assign Dout=Ram[Addr[6:2]];
always@(posedge Clk)begin
if(We)Ram[Addr[6:2]]<=Din;
end
integer i;
initial begin
for(i=0;i<32;i=i+1)
Ram[i]=0;
end
endmodule
```

4.5 SHIFTER32_L2

21) 所处位置

22) 模块功能

一个固定左移两位的移位器

23) 实现思路

使用 32 位移位器 SHIFTER32，固定左移两位即可

24) 引脚及控制信号

EXTIMM：指令中的偏移量，输入信号

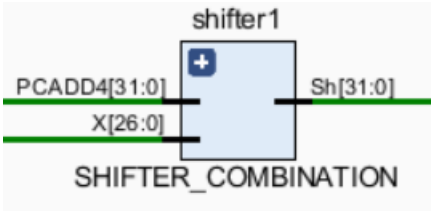
EXTIMML2：偏移量左移后的结果，输出信号

25) 主要实现代码

```
module SHIFTER32_L2(X,Sh);
input [31:0] X;
output [31:0] Sh;
parameter z=2'b00;
assign Sh={X[29:0],z};
endmodule
```

4.6 SHIFTER_COMBINATION

26) 所处位置



27) 模块功能

J 指令中用以产生跳转的目标地址

28) 实现思路

跳转的目标地址采用拼接的方式形成，最高 4 位为 PC+4 的最高 4 位，中间 26 位为 J 型指令的 26 位立即数字段，最低两位为 0.

29) 引脚及控制信号

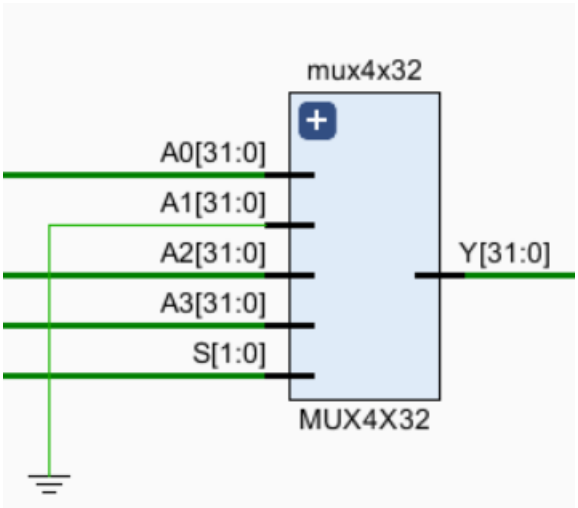
Inst[26:0]：指令编码的低 26 位字段，输入信号。
PCadd4：PC+4 的 32 位字段，输入信号。
InstL2：32 位转移目标地址，输出信号。

30) 主要实现代码

```
module SHIFTER_COMBINATION(X,PCADD4,Sh);
input [26:0] X;
input [31:0] PCADD4;
output [31:0] Sh;
parameter z=2'b00;
assign Sh={PCADD4[3:0],X[26:0],z};
endmodule
```

4.7 MUX4X32

31) 所处位置



32) 模块功能

实现目标地址的选择

33) 实现思路

目标地址可能是 PC+4，也可能是 beq 和 bne 的跳转地址或是 J 型跳转地址，所以采用一个 32 位四选一多路选择器

34) 引脚及控制信号

PCadd4：PC+4 的地址，输入信号
0：空位，输入信号
mux4x32_2：beq 和 bne 指令的跳转地址，输入信号

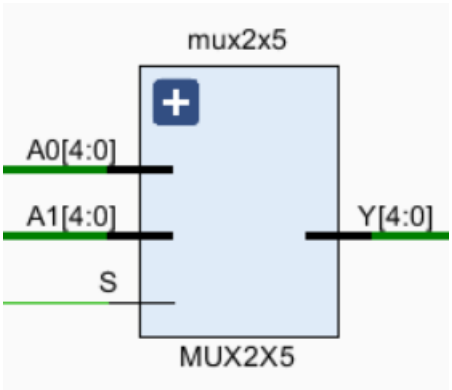
InstL2：J 指令的跳转地址，输入信号
Pcsrc：对地址进行选择的控制信号，输入信号
Result：目标地址，输出信号

35) 主要实现代码

```
module MUX4X32 (A0, A1, A2, A3, S, Y);
input [31:0] A0, A1, A2, A3;
input [1:0] S;
output [31:0] Y;
function [31:0] select;
input [31:0] A0, A1, A2, A3;
input [1:0] S;
case(S)
2'b00: select = A0;
2'b01: select = A1;
2'b10: select = A2;
2'b11: select = A3;
endcase
endfunction
assign Y = select (A0, A1, A2, A3, S);
endmodule
```

4.8 MUX2X5

36) 所处位置



37) 模块功能

R 型指令和 I 行指令的 Wr 信号不同，所以需要一个 5 位二选一选择器进行选择。

38) 实现思路

R 型指令 Wr 选择 rd 信号，I 型指令 Wr 选择 rt 信号。

39) 引脚及控制信号

Inst[15:11],：R 型指令的 rd 信号，输入信号

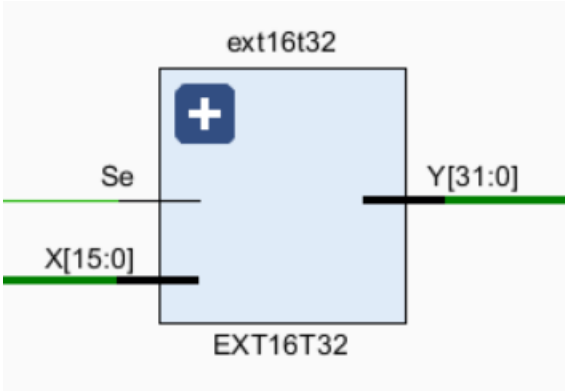
nst[20:16]：I 型指令的 rt 信号，输入信号
Regrt：选择指令的控制信号，输入信号
Wr：Wr 信号，输出信号

40) 主要实现代码

```
module MUX2X5(A0,A1,S,Y);
input [4:0] A0,A1;
input S;
output [4:0] Y;
function [4:0] select;
input [4:0] A0,A1;
input S;
case(S)
0:select=A0;
1:select=A1;
endcase
endfunction
assign Y=select(A0,A1,S);
endmodule
```

4.9 EXT16T32

41) 所处位置



42) 模块功能

I 指令的 addi 需要对立即数进行符号拓展，andi 和 ori 需要对立即数进行零扩展，所以需要一个扩展模块。

43) 实现思路

采用一个 16 位扩展成 32 位的扩展模块 EXT16T32，实现零扩展和符号扩展

44) 引脚及控制信号

Inst[15:0]：I 型指令的立即数字段，输入信号。
Se：选择零扩展或是符号扩展的控制模块，输入信号。

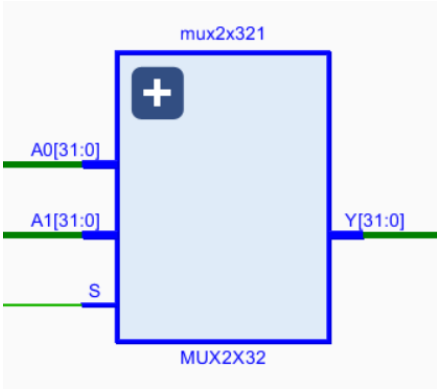
EXTIMM：扩展后的立即数，输出信号。

45）主要实现代码

```
module EXT16T32 (X, Se, Y);
input [15:0] X;
input Se;
output [31:0] Y;
wire [31:0] E0, E1;
wire [15:0] e = {16{X[15]}};
parameter z = 16'b0;
assign E0 = {z, X};
assign E1 = {e, X};
MUX2X32 i(E0, E1, Se, Y);
endmodule
```

4.10 MUX2X32

46）所处位置



47）模块功能 1

ALU 的 Y 端输入信号种类根据指令的不同而不同

48）实现思路 1

在执行 R 型指令时，ALU 的 Y 端输入信号可能来自 Qb，在执行 I 型指令的 addi，andi 和 ori 指令时时，ALU 的 Y 端输入信号来自 EXT16T32，所以需要一个二选一选择器。

49）引脚及控制信号 1

EXTIMM：来自 EXT16T32 的信号，输入信号。
Qb：来自 REGFLE 中 Qb 端口的信号，输入信号。
Aluqb：控制信号。
Y：，输入 ALU 进行后续计算的信号，输出信号。

50) 模块功能 2

对写入寄存器的数据进行选择

51) 实现思路 2

在 lw 指令中，需要将 DATAMEM 中选中储存器的值保存到 REGFILE 的寄存器中，而其他会更新 REGFILE 的指令的更新信号来自于 ALU 的 R 输出端。所以需要一个二选一选择器进行选择。

52) 引脚及控制信号 2

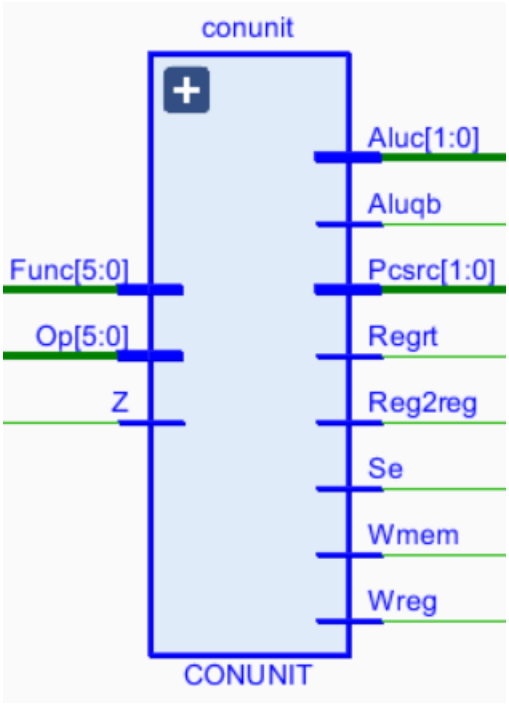
Dout: DATAMEM 的输出值，输入信号
R：ALU 的输出值，输入信号
Reg2reg：控制信号
D：：写入 R 寄存器堆 D 端的信号，输出信号

53) 主要实现代码

```
module MUX2X32(A0,A1,S,Y);
input [31:0] A0,A1;
input S;
output [31:0] Y;
function [31:0] select;
input [31:0] A0,A1;
input S;
case(S)
0:select=A0;
1:select=A1;
endcase
endfunction
assign Y=select(A0,A1,S);
endmodule
```

4.11 CONUNIT

54) 所处位置



55) 模块功能

控制器是作为 CPU 控制信号产生的器件，通过通过解析 op 得到该指令的各种控制信号，使其他器件有效或无效。

56) 实现思路

参照引脚和控制信号设计

57) 引脚

Inst[31:26]：Op，输入信号。
Inst[5:0]：Func，输入信号。
Z：零标志信号，对 Pcsrc 有影响，输入信号。
Regrt：控制输入寄存器的 Wr 端口，输出信号。
Se：控制扩展模块，输出信号。
Wreg：控制寄存器端的写使能信号，输出信号。
Aluqb：控制 ALU 的 Y 端口的输入值，输出信号。
Aluc：控制 ALU 的计算种类，输出信号。
Wmem：控制数据存储器的写使能信号，输出信号。
Pcsrc：控制目标指令地址，输出信号。
Reg2reg：控制 REHFILE 更新值的来源。

58) 控制信号

输入端口				输出端口							
Op [5:0]	Func [5:0]	备注	Z	Regrt	Se	Wreg	Aluqb	Aluc [1:0]	Wmem	Pcsrc [1:0]	Reg2reg
000000	100000	add	X	0	0	1	1	00	0	00	1
000000	100010	sub	X	0	0	1	1	01	0	00	1
000000	100100	and	X	0	0	1	1	10	0	00	1
000000	100101	or	X	0	0	1	1	11	0	00	1
001000	-	addi	X	1	1	1	0	00	0	00	1
001100	-	andi	X	1	0	1	0	10	0	00	1
001101	-	ori	X	1	0	1	0	11	0	00	1
100011	-	lw	X	1	1	1	0	00	0	00	0
101011	-	sw	X	1	1	0	0	00	1	00	1
000100	-	beq	0	1	1	0	1	01	0	00	1
000100	-	beq	1	1	1	0	1	01	0	10	1
000101	-	bne	0	1	1	0	1	01	0	10	1
000101	-	bne	1	1	1	0	1	01	0	00	1
000010	-	j	X	1	0	0	1	00	0	11	1

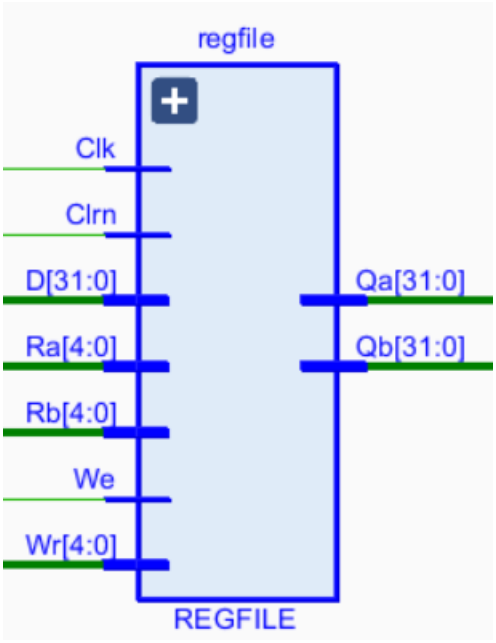
59) 主要实现代码

```
module
CONUNIT(Op,Func,Z,Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Pcsrc,Reg2reg);
input[5:0]Op,Func;
input Z;
output Regrt,Se,Wreg,Aluqb,Wmem,Reg2reg;
output[1:0]Pcsrc,Aluc;
wire R_type=~|Op;
wire
I_add=R_type&Func[5]&~Func[4]&~Func[3]&~Func[2]&~Func[1]&~Func[0];
```

```
wire
I_sub=R_type&Func[5]&~Func[4]&~Func[3]&~Func[2]&Func[1]&~Func[0];
wire
I_and=R_type&Func[5]&~Func[4]&~Func[3]&Func[2]&~Func[1]&~Func[0];
wire I_or=R_type&Func[5]&~Func[4]&~Func[3]&Func[2]&~Func[1]&Func[0];
wire I_addi=~Op[5]&~Op[4]&Op[3]&~Op[2]&~Op[1]&~Op[0];
wire I_andi=~Op[5]&~Op[4]&Op[3]&Op[2]&~Op[1]&~Op[0];
wire I_ori=~Op[5]&~Op[4]&Op[3]&Op[2]&~Op[1]&Op[0];
wire I_lw=Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0];
wire I_sw=Op[5]&~Op[4]&Op[3]&~Op[2]&Op[1]&Op[0];
wire I_beq=~Op[5]&~Op[4]&~Op[3]&Op[2]&~Op[1]&~Op[0];
wire I_bne=~Op[5]&~Op[4]&~Op[3]&Op[2]&~Op[1]&Op[0];
wire I_J=~Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&~Op[0];
assign Regrt=I_addi|I_andi|I_ori|I_lw|I_sw|I_beq|I_bne|I_J;
assign Se=I_addi|I_lw|I_sw|I_beq|I_bne;
assign Wreg=I_add|I_sub|I_and|I_or|I_addi|I_andi|I_ori|I_lw;
assign Aluqb=I_add|I_sub|I_and|I_or|I_beq|I_bne|I_J;
assign Aluc[1]=I_and|I_or|I_andi|I_ori;
assign Aluc[0]=I_sub|I_or|I_ori|I_beq|I_bne;
assign Wmem=I_sw;
assign Pcsrc[1]=I_beq&Z|I_bne&~Z|I_J;
assign Pcsrc[0]=I_J;
assign
Reg2reg=I_add|I_sub|I_and|I_or|I_addi|I_andi|I_ori|I_sw|I_beq|I_bne|
I_J;
endmodule
```

4.12 REGFILE

60) 所处位置



61）模块功能

给出要读取的两个寄存器编号和要写入的寄存器编号，然后由 Qa 和 Qb 端口更新 Ra 和 Rb 端口的输入编号分别输入其值。

62）实现思路

由 32 个寄存器组成，增加两个端口用于接收要读取的两个寄存器编号，另一个端口用于接收要写入的寄存器的编号。且在时钟上升沿将 D 写入。

63）引脚及控制信号

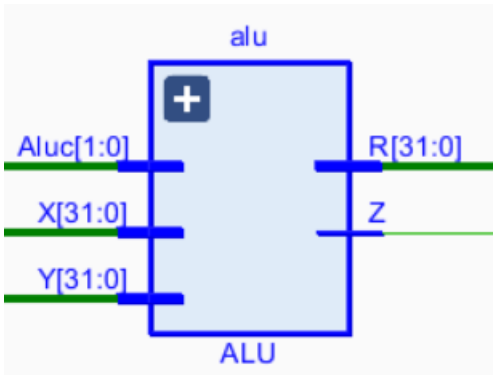
Inst[25:21]：读取寄存器编号 1，输入信号。
Inst[20:16]：读取寄存器编号 2 或立即数，输入信号。
D：寄存器更新值，输入信号。
Wr：写入寄存器编号 3，输入信号。
Wreg：写使能信号，为 0 的时候不能写入，D 值不更新，为 1 的时候能写入，D 值更新，输入信号。
Clk：时钟周期，输入信号。
Reset：清零信号，输入信号。
Qa：输出寄存器 1 的值，输入信号。
Qb：输出寄存器 2 的值，输入信号。

64）主要实现代码

```
module REGFILE(Ra,Rb,D,Wr,We,Clk,Clrn,Qa,Qb);
input [4:0]Ra,Rb,Wr;
input [31:0]D;
input We,Clk,Clrn;
output [31:0]Qa,Qb;
wire
[31:0]Y_mux,Q31_reg32,Q30_reg32,Q29_reg32,Q28_reg32,Q27_reg32,Q26
_reg32,Q25_reg32,Q24_reg32,Q23_reg32,Q22_reg32,Q21_reg32,Q20_reg3
2,Q19_reg32,Q18_reg32,Q17_reg32,Q16_reg32,Q15_reg32,Q14_reg32,Q13
_reg32,Q12_reg32,Q11_reg32,Q10_reg32,Q9_reg32,Q8_reg32,Q7_reg32,Q
6_reg32,Q5_reg32,Q4_reg32,Q3_reg32,Q2_reg32,Q1_reg32,Q0_reg32;

DEC5T32E dec(Wr,We,Y_mux);

REG32
A(D,Y_mux,Clk,Clrn,Q31_reg32,Q30_reg32,Q29_reg32,Q28_reg32,Q27_re
g32,Q26_reg32,Q25_reg32,Q24_reg32,Q23_reg32,Q22_reg32,Q21_reg32,Q
20_reg32,Q19_reg32,Q18_reg32,Q17_reg32,Q16_reg32,Q15_reg32,Q14_re
g32,Q13_reg32,Q12_reg32,Q11_reg32,Q10_reg32,Q9_reg32,Q8_reg32,Q7_
reg32,Q6_reg32,Q5_reg32,Q4_reg32,Q3_reg32,Q2_reg32,Q1_reg32,Q0_re
g32);

MUX32X32
select1(Q0_reg32,Q1_reg32,Q2_reg32,Q3_reg32,Q4_reg32,Q5_reg32,Q6_
reg32,Q7_reg32,Q8_reg32,Q9_reg32,Q10_reg32,Q11_reg32,Q12_reg32,Q1
3_reg32,Q14_reg32,Q15_reg32,Q16_reg32,Q17_reg32,Q18_reg32,Q19_reg
32,Q20_reg32,Q21_reg32,Q22_reg32,Q23_reg32,Q24_reg32,Q25_reg32,Q2
6_reg32,Q27_reg32,Q28_reg32,Q29_reg32,Q30_reg32,Q31_reg32,Ra,Qa);
MUX32X32
select2(Q0_reg32,Q1_reg32,Q2_reg32,Q3_reg32,Q4_reg32,Q5_reg32,Q6_
reg32,Q7_reg32,Q8_reg32,Q9_reg32,Q10_reg32,Q11_reg32,Q12_reg32,Q1
3_reg32,Q14_reg32,Q15_reg32,Q16_reg32,Q17_reg32,Q18_reg32,Q19_reg
32,Q20_reg32,Q21_reg32,Q22_reg32,Q23_reg32,Q24_reg32,Q25_reg32,Q2
6_reg32,Q27_reg32,Q28_reg32,Q29_reg32,Q30_reg32,Q31_reg32,Rb,Qb);

Endmodule
```

4.13 ALU

65) 所处位置



66) 模块功能

算数逻辑部件，需要实现加，减，按位与，按位或。

67) 实现思路

需要 2 位控制信号控制运算类型，核心部件是 32 位加法器 ADDSUB_32。

68) 引脚和控制信号

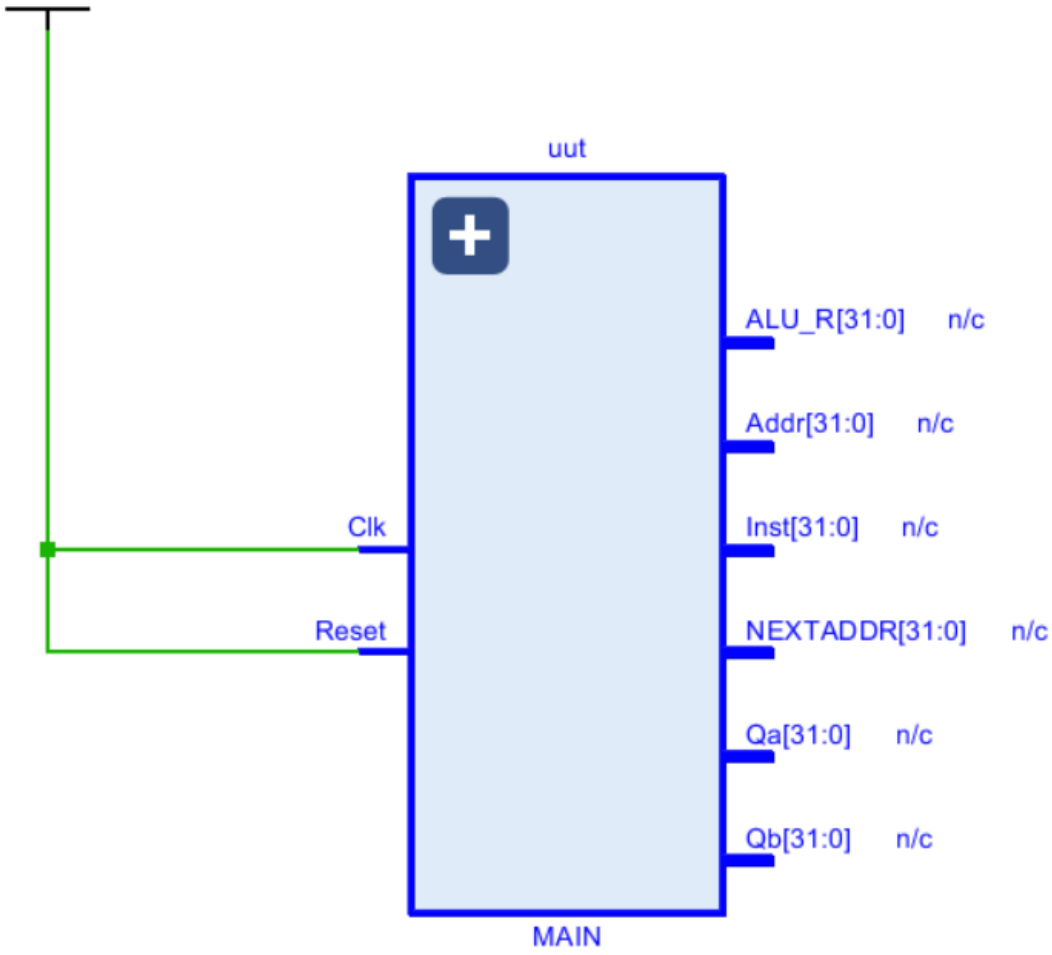
Qa：寄存器 1 的值。
Y：寄存器 2 的值或立即数。
Aluc：控制信号。
R：输入寄存器端口 D 的计算结果，输出信号。
Z：当值为 1 时代表两个输入信号值相等，当值为 0 时代表两个输入信号不等，输出信号。

69) 主要实现代码

```
module ALU(X,Y,Aluc,R,Z);//ALU 代码
input [31:0]X,Y;
input [1:0]Aluc;
output [31:0]R;
output Z;
wire[31:0]d_as,d_and,d_or,d_and_or;
ADDSUB_32 as(X,Y,Aluc[0],d_as);
assign d_and=X&Y;
assign d_or=X|Y;
MUX2X32 select1(d_and,d_or,Aluc[0],d_and_or);
MUX2X32 seleted(d_as,d_and_or,Aluc[1],R);
assign Z=~|R;
endmodule
```

4.14 SingleCycleCPU

70) 所处位置



71) 模块功能

实现 CPU 的封装，设计输出信号使得在方正时便于观察其波形图

72) 实现思路

调用各个下层模块并将他们的输入和输出连接到一起。

73) 引脚及控制信号名

CLk:时钟周期，外部输入信号。

Reset：清零信号，外部输入信号。

74) 主要实现代码

```
module MAIN(Clk,Reset,Addr,Inst,Qa,Qb,ALU_R,NEXTADDR,D);
input Clk,Reset;
output [31:0] Inst,NEXTADDR,ALU_R,Qb,Qa,Addr,D;

wire
[31:0]Result,PCadd4,EXTIMM,InstL2,EXTIMML2,D,Y,Dout,mux4x32_2,R;
wire Z,Regrt,Se,Wreg,Aluqb,Reg2reg,Cout,Wmem;
wire [1:0]Aluc,Pcsrc;
```

```
wire [4:0]Wr;

PC pc(Clk,Reset,Result,Addr);
PCadd4 pcadd4(Addr,PCadd4);
INSTMEM instmem(Addr,Inst);

CONUNIT
conunit(Inst[31:26],Inst[5:0],Z,Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Pcsrc,
Reg2reg);
MUX2X5 mux2x5(Inst[15:11],Inst[20:16],Regrt,Wr);
EXT16T32 ext16t32(Inst[15:0],Se,EXTIMM);
SHIFTER_COMBINATION shifter1(Inst[26:0],PCadd4,InstL2);
SHIFTER32_L2 shifter2(EXTIMM,EXTIMML2);
REGFILE regfile(Inst[25:21],Inst[20:16],D,Wr,Wreg,Clk,Reset,Qa,Qb);
MUX2X32 mux2x321(EXTIMM,Qb,Aluqb,Y);
ALU alu(Qa,Y,Aluc,R,Z);
DATAMEM datamem(R,Qb,Clk,Wmem,Dout);
MUX2X32 mux2x322(Dout,R,Reg2reg,D);
CLA_32 cla_32(PCadd4,EXTIMML2,0,mux4x32_2, Cout);
MUX4X32 mux4x32(PCadd4,0,mux4x32_2,InstL2,Pcsrc,Result);
assign NEXTADDR=Result;
assign ALU_R=R;
endmodule
```

4.15 Test

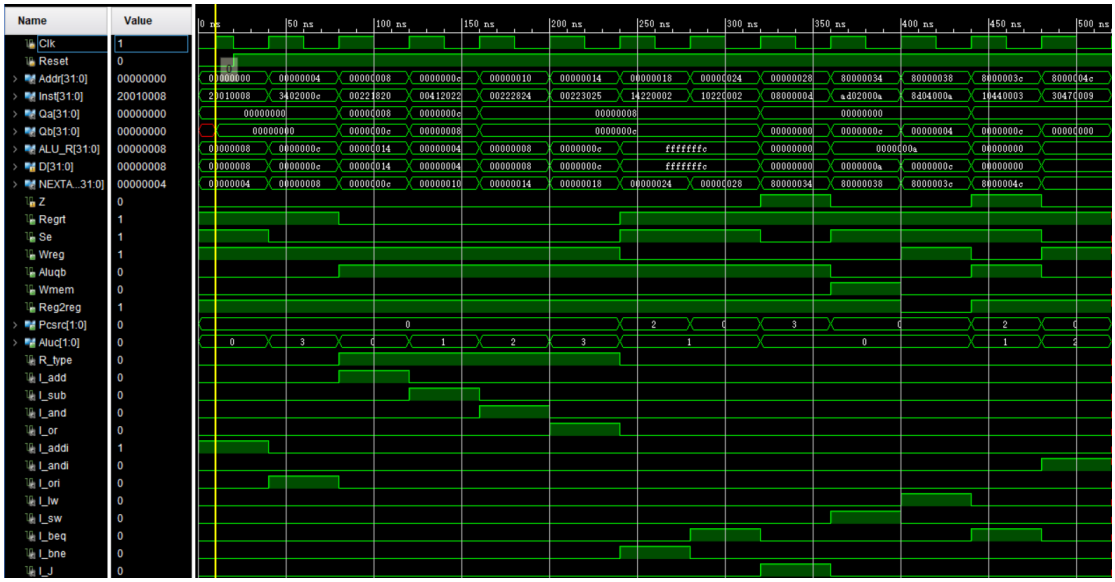
主要实现代码：

```
module TEST;
reg Clk;
reg Reset;
wire [31:0] Addr,Inst,Qa,Qb,ALU_R,NEXTADDR;


MAIN uut(
.Clk(Clk),
.Reset(Reset),
.Addr(Addr),
.Inst(Inst),
.Qa(Qa),
.Qb(Qb),
.ALU_R(ALU_R),
.NEXTADDR(NEXTADDR),
.D(D)
```

```
);

initial begin
Clk=0;Reset=0;
#10;
Clk=1;Reset=0;
#10;
Reset=1;
Clk=0;
forever #20 Clk=~Clk;
end

endmodule
```

五、 仿真模拟分析

5.1 仿真波形图



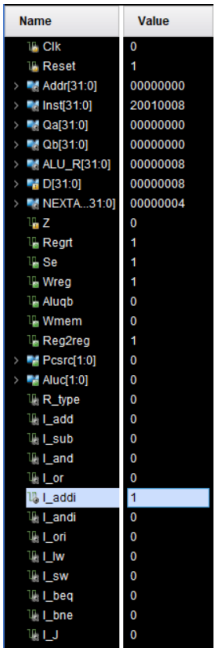
5.1 指令代码分析

1） assign Rom[5'h00]=32'h20010008;

二进制源码：001000 00000 00001 00000 00000 001000

指令含义：addi $1,$0,8

结果：$1=$0+8=8



2） assign Rom[5'h01]=32'h3402000C;

二进制源码：001101 00000 00010 00000 00000 001100

指令含义：ori $2,$0,12

结果：$2=12

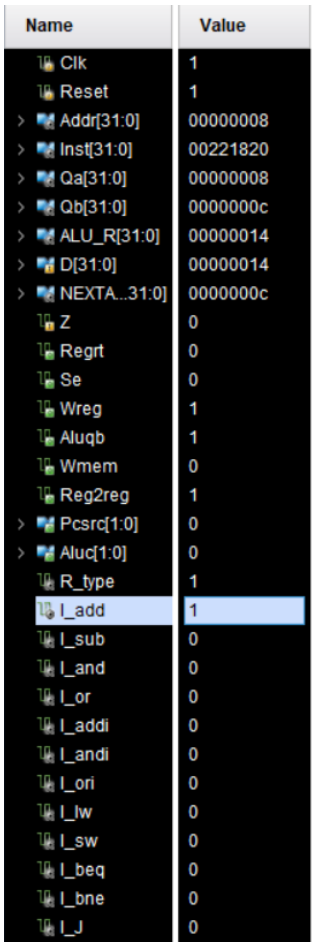
Name	Value
Clk	1
Reset	1
Addr[31:0]	00000004
Inst[31:0]	3402000c
Qa[31:0]	00000000
Qb[31:0]	00000000
ALU_R[31:0]	0000000c
D[31:0]	0000000c
NEXTA...31:0]	00000008
Z	0
Regrt	1
Se	0
Wreg	1
Aluqb	0
Wmem	0
Reg2reg	1
Pcsrc[1:0]	0
Aluc[1:0]	3
R_type	0
I_add	0
I_sub	0
I_and	0
I_or	0
I_addi	0
I_andi	0
I_ori	1
I_lw	0
I_sw	0
I_beq	0
I_bne	0
I_J	0

3) assign Rom[5'h02]=32'h00221820;

二进制源码：000000 00001 00010 00011 00000 100000

指令含义：add $3,$1,$2

结果：$3=8+12=20

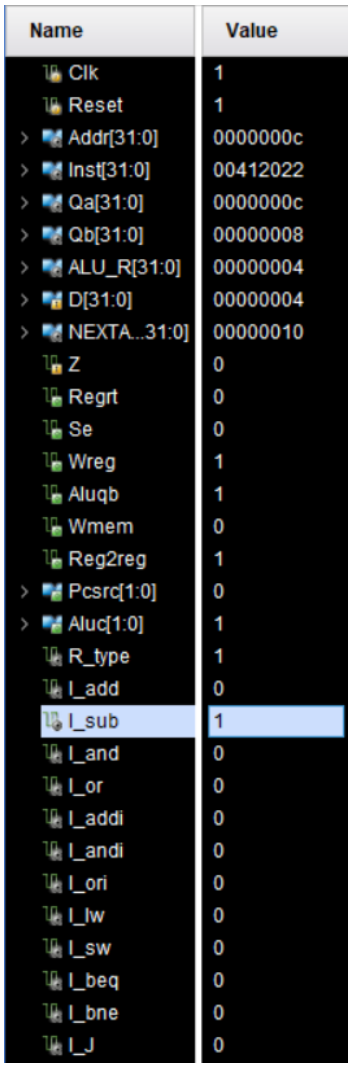


4) `assign Rom[5'h03]=32'h00412022;`

二进制源码：000000 00010 00001 00100 00000 100010

指令含义：sub $4,$2,$1

结果：$4=12-8=4



5) assign Rom[5'h04]=32'h00222824;

二进制源码：000000 00001 00010 00101 00000 100100

指令含义：and $5,$1,$2

结果：$5=b1000&b1100=b1000=h8

6) assign Rom[5'h05]=32'h00223025;

二进制源码：000000 00001 00010 00110 00000 100101

指令含义：or $1,$2,$6

结果：$6=b1000|b1100=b1100=12=hc

7） assign Rom[5'h06]=32'h14220002;

二进制源码：000101 00001 00010 0000000000000010

指令含义：bne $1,$2,2

结果：跳转到本地址+2 的地址位置

（1：偏移量扩展到 32 位。

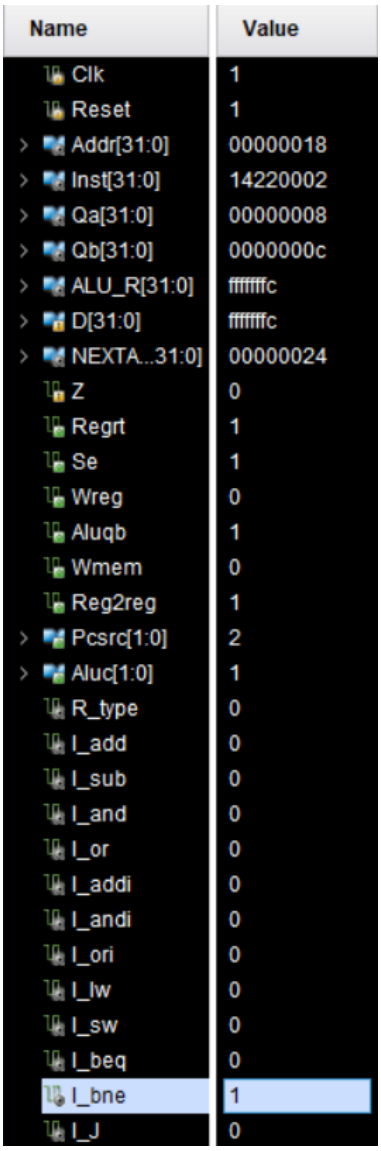
2：左移两位。

3：加 b1000。

4：加 PC 地址得到 Addr。

5：取 Addr[6:2]作为下一个地址。

）



8) assign Rom[5'h07]=32'hXXXXXXXX;

9) assign Rom[5'h08]=32'hXXXXXXXX;

10) assign Rom[5'h09]=32'h10220002;

二进制源码：000100 00001 00010 0000000000000010

指令含义：beq $1,$2,2

结果：没有跳转

11) assign Rom[5'h0A]=32'h0800000D;

二进制源码：000010 00000000000000000000001101

指令含义：J 0D

结果：跳转到第 14 条指令的地址位置

12) assign Rom[5'h0B]=32'hXXXXXXXX;

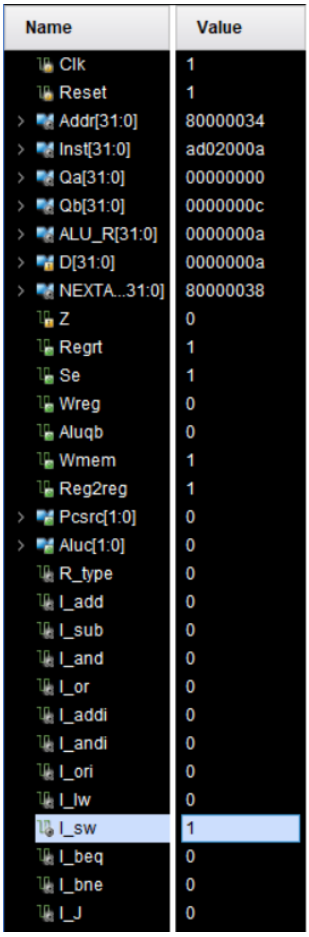
13) assign Rom[5'h0C]=32'hXXXXXXXX;

14) assign Rom[5'h0D]=32'hAD02000A;

二进制源码：101011 01000 00010 0000000000001010

指令含义：sw $2 10($8)

结果：将$2 的值写入 10+$8 的储存单元。



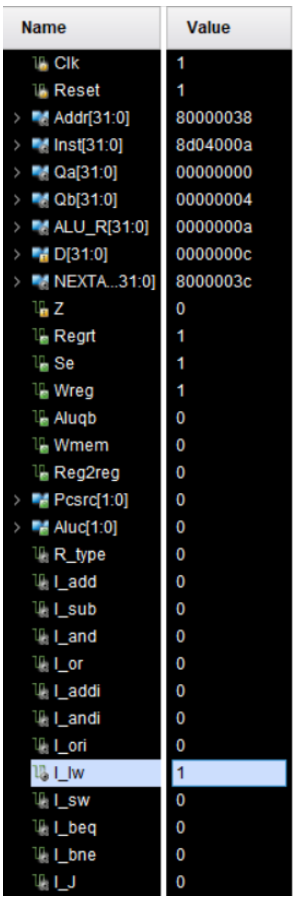
Name	Value
Clk	1
Reset	1
Addr[31:0]	80000034
Inst[31:0]	ad02000a
Qa[31:0]	00000000
Qb[31:0]	0000000c
ALU_R[31:0]	0000000a
D[31:0]	0000000a
NEXTA...31:0]	80000038
Z	0
Regrt	1
Se	1
Wreg	0
Aluqb	0
Wmem	1
Reg2reg	1
Pcsrc[1:0]	0
Aluc[1:0]	0
R_type	0
I_add	0
I_sub	0
I_and	0
I_or	0
I_addi	0
I_andi	0
I_ori	0
I_lw	0
I_sw	1
I_beq	0
I_bne	0
I_J	0

15) assign Rom[5'h0E]=32'h8D04000A;

二进制源码：100011 01000 00100 0000000000001010

指令含义：lw $4 10($8) 数据储存器中$8 和偏移量相加的地址读取到$4。

结果： $4=$2=12

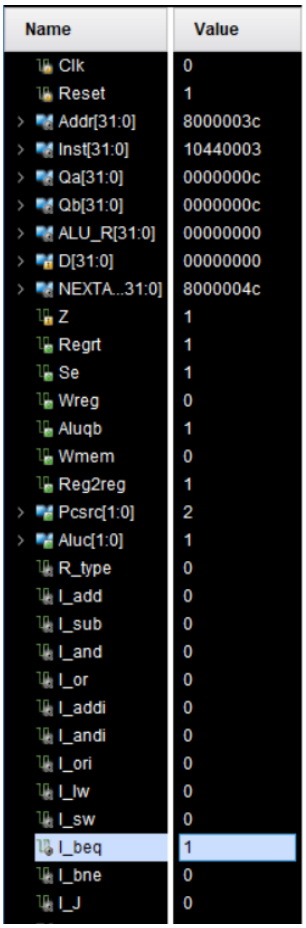


16) assign Rom[5'h0F]=32'h10440003;

二进制源码：000100 00010 00100 0000000000000011

指令含义：beq $2,$4,3

结果：跳转到本地址+3 的地址位置



17) assign Rom[5'h10]=32'hXXXXXXXX;

18) assign Rom[5'h11]=32'hXXXXXXXX;

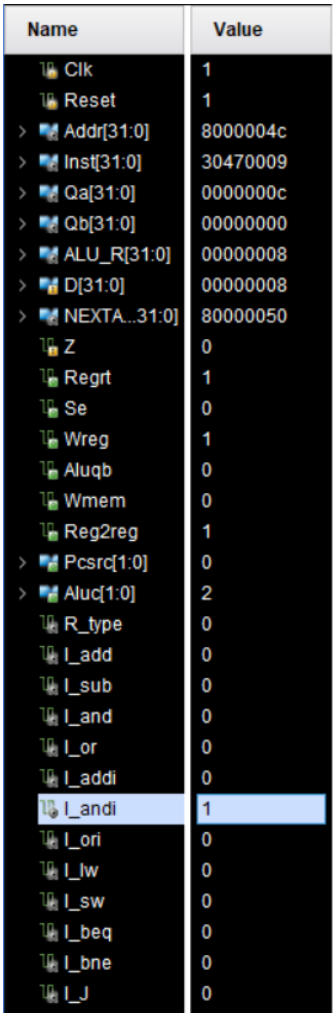
19) assign Rom[5'h12]=32'hXXXXXXXX;

20) assign Rom[5'h13]=32'h30470009;

二进制源码：001100 00010 00111 0000000000001001

指令含义：andi $2,9,$7

结果：$7=b1100&b1001=b1000=8




```
21) assign Rom[5'h14]=32'hXXXXXXXX;
22) assign Rom[5'h15]=32'hXXXXXXXX;
23) assign Rom[5'h16]=32'hXXXXXXXX;
24) assign Rom[5'h17]=32'hXXXXXXXX;
25) assign Rom[5'h18]=32'hXXXXXXXX;
26) assign Rom[5'h19]=32'hXXXXXXXX;
27) assign Rom[5'h1A]=32'hXXXXXXXX;
28) assign Rom[5'h1B]=32'hXXXXXXXX;
29) assign Rom[5'h1C]=32'hXXXXXXXX;
30) assign Rom[5'h1D]=32'hXXXXXXXX;
31) assign Rom[5'h1E]=32'hXXXXXXXX;
32) assign Rom[5'h1F]=32'hXXXXXXXX;
```

图 6-1 单周期 CPU 仿真波形图

6.2 仿真结果分析

结合指令储存器，主要观察 Qa，Qb，ALU_R 三个变量（ALU 计算部件的 X，Y，R 端）

指令地址	指令代码	含义	Qa	Qb	R
00000000	20010008	addi $1,$0,8 $1=8	$0(0）	8(立即数)	$1(8)
00000004	3402000C	ori $2,$0,12 $2=12	$0(0)	12(立即数)	$2(12)
00000008	00221820	add $3,$1,$2 $3=20	$1(8)	$2(12)	$3(20)
0000000C	00412022	sub $4,$2,$1 $4=4	$2(12)	$1(8)	$4(4)
00000010	00222824	and $5,$1,$2	$1(8)	$2(12)	$5(8)
00000014	00223025	or $6,$1,$2	$1(8)	$2(12)	$6(C)
00000018	14220002	bne $1,$2,2，不等转移	$1(8)	$2(12)	
00000024	10220002	beq $1,$2,2，相等转移	$1(8)	$2(12)	
00000028	0800000D	J 0D ，跳转到 0D(34)			
00000034	AD02000A	sw $2 10($8) memory[$8+10]=12	$8	10	a
00000038	8D04000A	lw $4 10($8) $4=12	$8	10	a
0000003C	20810000	addi $1,$4,0 $1=12	$4(4)	0	$1(4)

表 6-1 仿真分析

六、 结论和体会

6.1 体会感悟

通过此次 10 周的 CPU 设计实验，让我们对 CPU 内部组成以及指令在 CPU 部件上如何运作有了一个更深的理解。在实验过程中，我们遇到了各种问题，一开始老师布置下来的 CPU 任务的时候，完全是懵的，因为 CPU 器件和指令运算只在课本上学习，从来没有真正实践过，现在需要自己设计 CPU 的各个部件，而且要将指令在器件上运行，感觉很复杂。但在接下来的日子，我们没有因为

不会而放弃，而是努力专心去设计好每个部件，对每个部件的功能进行模拟仿真，确保这一部件模块不出错，在设计过程中，感觉慢慢可以理清思路，也明白了下一步需要设计的东西通过此次实验，让我们对 CPU 有了更深的理解，而不只是纸上谈兵。

6.2 对本实验过程及方法、手段的改进建议

1.过程中，应该每个部件都分别进行调试之后再组装在一起。
2.各部件尽量再拆分为更小的部件组合而成。