**目 录**

1. 设计目的与目标
2. 课程设计器材

三、CPU逻辑设计总体方案

四、模块详细设计

五、仿真模拟分析

六、结论和体会

1. **设计目的与目标**
   1. **实验内容**
2. 本实例所设计CPU的指令格式的拟定；
3. 基本功能部件的设计与实现；
4. CPU各主要功能部件的设计与实现；
5. CPU的封装；
6. 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. **实验要求**
7. 至少支持add、sub、and、or、addi、andi、ori、lw、sw、beq、bne和j十二条指令。
8. **课程设计器材**

**2.1 硬件平台**

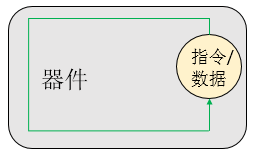
无

**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。
4. **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**

* 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**

指令0

指令1

指令5

指令2

指令4

1个时钟周期

Clock

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**

26

31

25

21

20

16

15

11

10

6

5

0

op

rs

rt

rd

sa

func

R型指令

26

31

25

21

20

16

15

0

op

rs

rt

immediate

I型指令

26

31

25

0

op

address

J型指令

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。

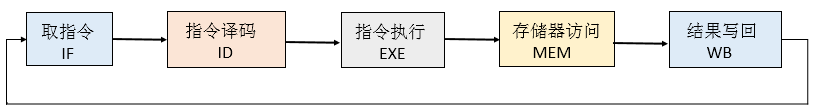
I型指令的低16位是立即数，计算时需扩展到32位，依指令的不同需进行零扩展和符号扩展。

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。

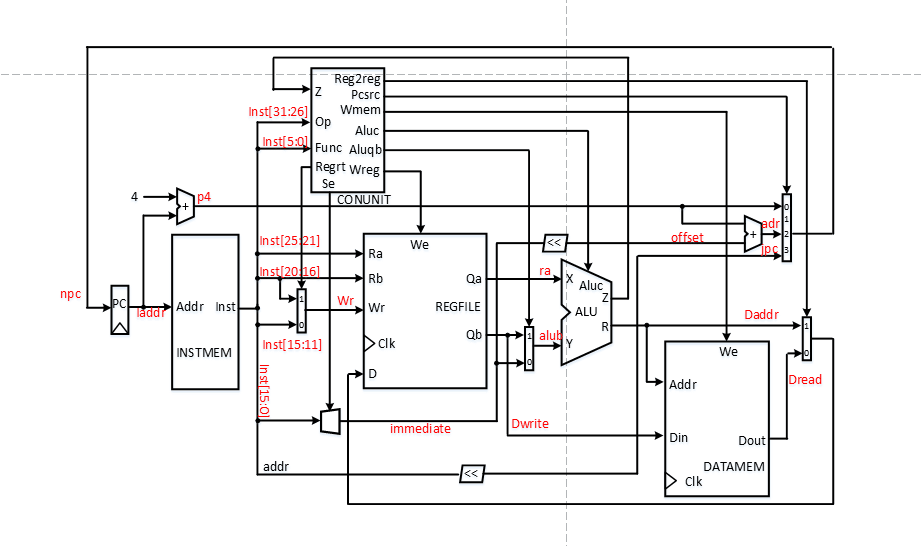


**图3-3 单周期CPU指令处理过程**

* 1. **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。

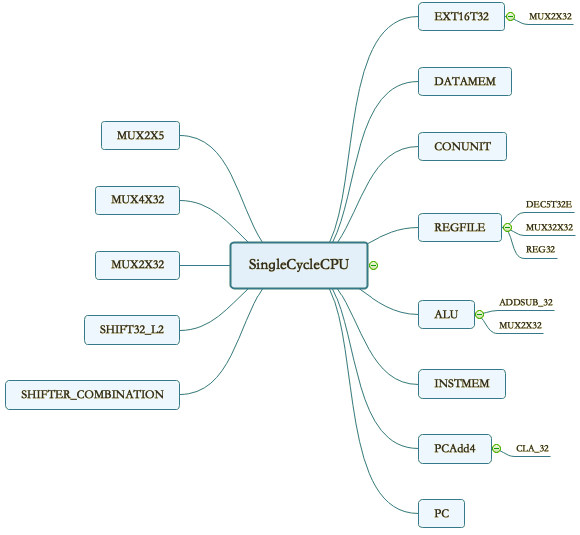
**3.2.1总体结构图**



**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**

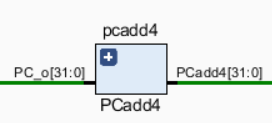
**** 根据实验原理中的单周期CPU总体结构图，我们可以清楚的知道单周期CPU的设计应包括PC，PCAdd4，INSTMEM，CONUNIT，REGFILE，ALU，DATAMEM， EXT16T32这几个核心模块，其中PCAdd4模块需要用到32位加法器CLA\_32。此外还需要左移处理模块SHIFTER\_COMBINATION,一个固定左移两位的移位器SHIFT32\_L2，一个四选一多路选择器MUX4X32，两个32位二选一多路选择器MUX2X32，一个5位二选一多路选择器MUX2X5，一个数据扩展器EXT16T32。其中为了运行整个CPU还需要加入一个顶层模块（SingleCycleCPU）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**

1. **模块详细设计**

**4.1 PCAdd4**

1. **所处位置**



1. **模块功能**

作为PC寄存器的更新信号。

1. **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。

1. **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

1. **主要实现代码**
2. **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**

1. **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**

1. **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);**

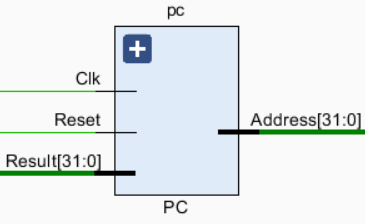
**nand getCout(Cout, tmp\_1\_4\_0, tmp\_1\_4\_1, tmp\_1\_4\_2, tmp\_1\_4\_3,tmp\_1\_4\_4);**

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

**endmodule**

**4.2 PC**

1. **所处位置**



1. **模块功能**

用于给出指令在指令储存器中的地址**。**

1. **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。

1. **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

1. **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;**

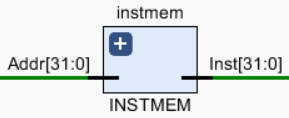
**end**

**end**

**endmodule**

**4.3 INSTMEM**

1. **所处位置**



1. **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。

1. **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。

1. **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

1. **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;

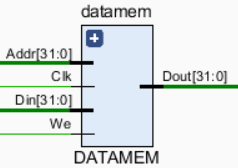
assign Rom[5'h1F]=32'hXXXXXXXX;

assign Inst=Rom[Addr[6:2]];

endmodule

**4.4 DATAMEM**

1. **所处位置**



1. **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。

1. **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。

1. **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。

1. **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**

1. **所处位置**



1. **模块功能**

一个固定左移两位的移位器

1. **实现思路**

使用32位移位器SHIFTER32，固定左移两位即可

1. **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

1. **主要实现代码**

**module SHIFTER32\_L2(X,Sh);**

**input [31:0] X;**

**output [31:0] Sh;**

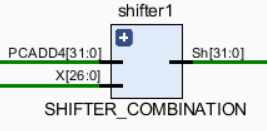
**parameter z=2'b00;**

**assign Sh={X[29:0],z};**

**endmodule**

**4.6 ﻿** **SHIFTER\_COMBINATION**

1. **所处位置**



1. **模块功能**

J指令中用以产生跳转的目标地址

1. **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.

1. **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。

1. **主要实现代码**

**module SHIFTER\_COMBINATION(X,PCADD4,Sh);**

**input [26:0] X;**

**input [31:0] PCADD4;**

**output [31:0] Sh;**

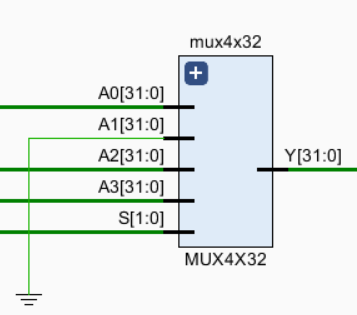
**parameter z=2'b00;**

**assign Sh={PCADD4[3:0],X[26:0],z};**

**endmodule**

**4.7 ﻿** **MUX4X32**

1. **所处位置**



1. **模块功能**

实现目标地址的选择

1. **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

1. **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

1. **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**

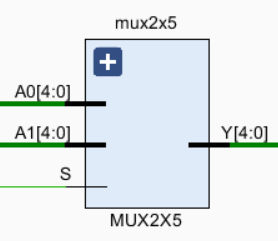
**endfunction**

**assign Y = select (A0, A1, A2, A3, S);**

**endmodule**

**4.8 ﻿ MUX2X5**

1. **所处位置**



1. **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。

1. **实现思路**

R型指令Wr选择rd信号，I型指令Wr选择rt信号。

1. **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

1. **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**

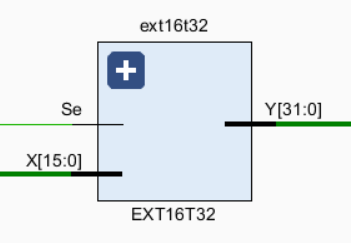
**endfunction**

**assign Y=select(A0,A1,S);**

**endmodule**

**4.9 ﻿** **EXT16T32**

1. **所处位置**



1. **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。

1. **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

1. **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。

1. **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};**

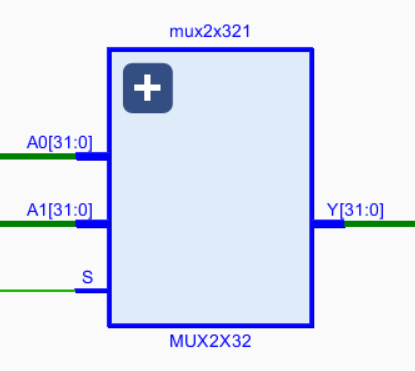
**assign E1 = {e, X};**

**MUX2X32 i(E0, E1, Se, Y);**

**endmodule**

**4.10 MUX2X32**

1. **所处位置**



1. **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

1. **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。

1. **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。

1. **模块功能2**

对写入寄存器的数据进行选择

1. **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。

1. **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

1. **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**

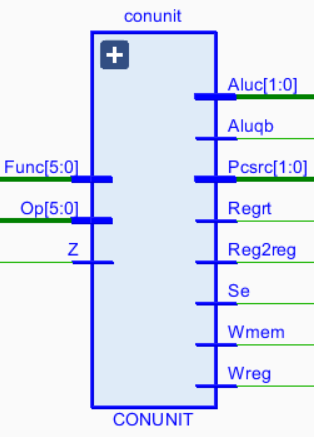
**endfunction**

**assign Y=select(A0,A1,S);**

**endmodule**

**4.11 CONUNIT**

1. **所处位置**



1. **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。

1. **实现思路**

参照引脚和控制信号设计

1. **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。

1. **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 |

1. **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;**

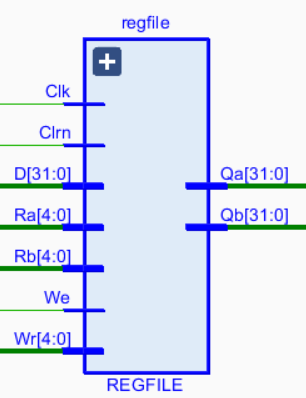
**assign Pcsrc[0]=I\_J;**

**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**

1. **所处位置**



1. **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。

1. **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。

1. **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。

1. **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\_reg32,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,Q6\_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\_reg32,Q26\_reg32,Q25\_reg32,Q24\_reg32,Q23\_reg32,Q22\_reg32,Q21\_reg32,Q20\_reg32,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,Q6\_reg32,Q5\_reg32,Q4\_reg32,Q3\_reg32,Q2\_reg32,Q1\_reg32,Q0\_reg32);**

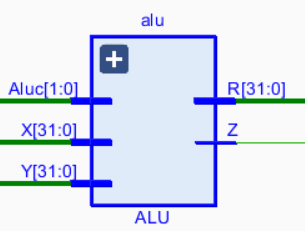
**MUX32X32 select1(Q0\_reg32,Q1\_reg32,Q2\_reg32,Q3\_reg32,Q4\_reg32,Q5\_reg32,Q6\_reg32,Q7\_reg32,Q8\_reg32,Q9\_reg32,Q10\_reg32,Q11\_reg32,Q12\_reg32,Q13\_reg32,Q14\_reg32,Q15\_reg32,Q16\_reg32,Q17\_reg32,Q18\_reg32,Q19\_reg32,Q20\_reg32,Q21\_reg32,Q22\_reg32,Q23\_reg32,Q24\_reg32,Q25\_reg32,Q26\_reg32,Q27\_reg32,Q28\_reg32,Q29\_reg32,Q30\_reg32,Q31\_reg32,Ra,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,Q13\_reg32,Q14\_reg32,Q15\_reg32,Q16\_reg32,Q17\_reg32,Q18\_reg32,Q19\_reg32,Q20\_reg32,Q21\_reg32,Q22\_reg32,Q23\_reg32,Q24\_reg32,Q25\_reg32,Q26\_reg32,Q27\_reg32,Q28\_reg32,Q29\_reg32,Q30\_reg32,Q31\_reg32,Rb,Qb);**

**Endmodule**

**4.13 ALU**

1. **所处位置**



1. **模块功能**

算数逻辑部件，需要实现加，减，按位与，按位或。

1. **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。

1. **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。

1. **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);**

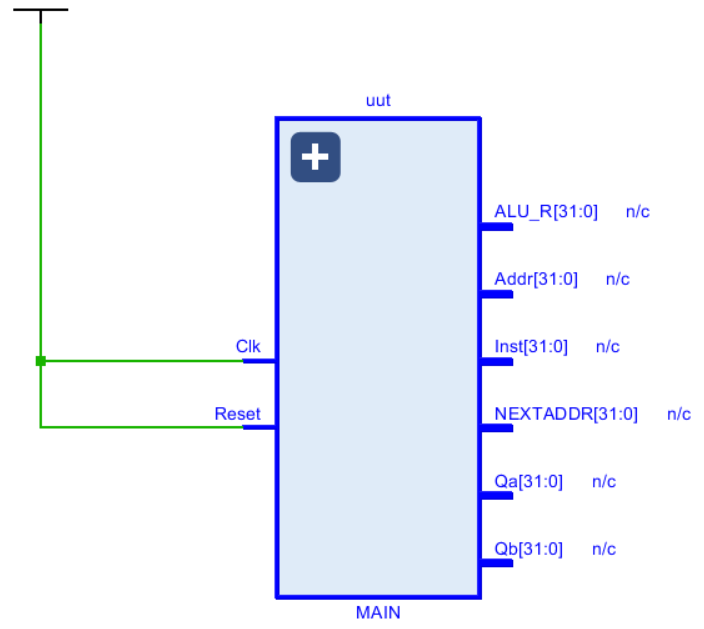
**MUX2X32 seleted(d\_as,d\_and\_or,Aluc[1],R);**

**assign Z=~|R;**

**endmodule**

**4.14 ﻿SingleCycleCPU**

1. **所处位置**



1. **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

1. **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。

1. **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。

1. **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;**

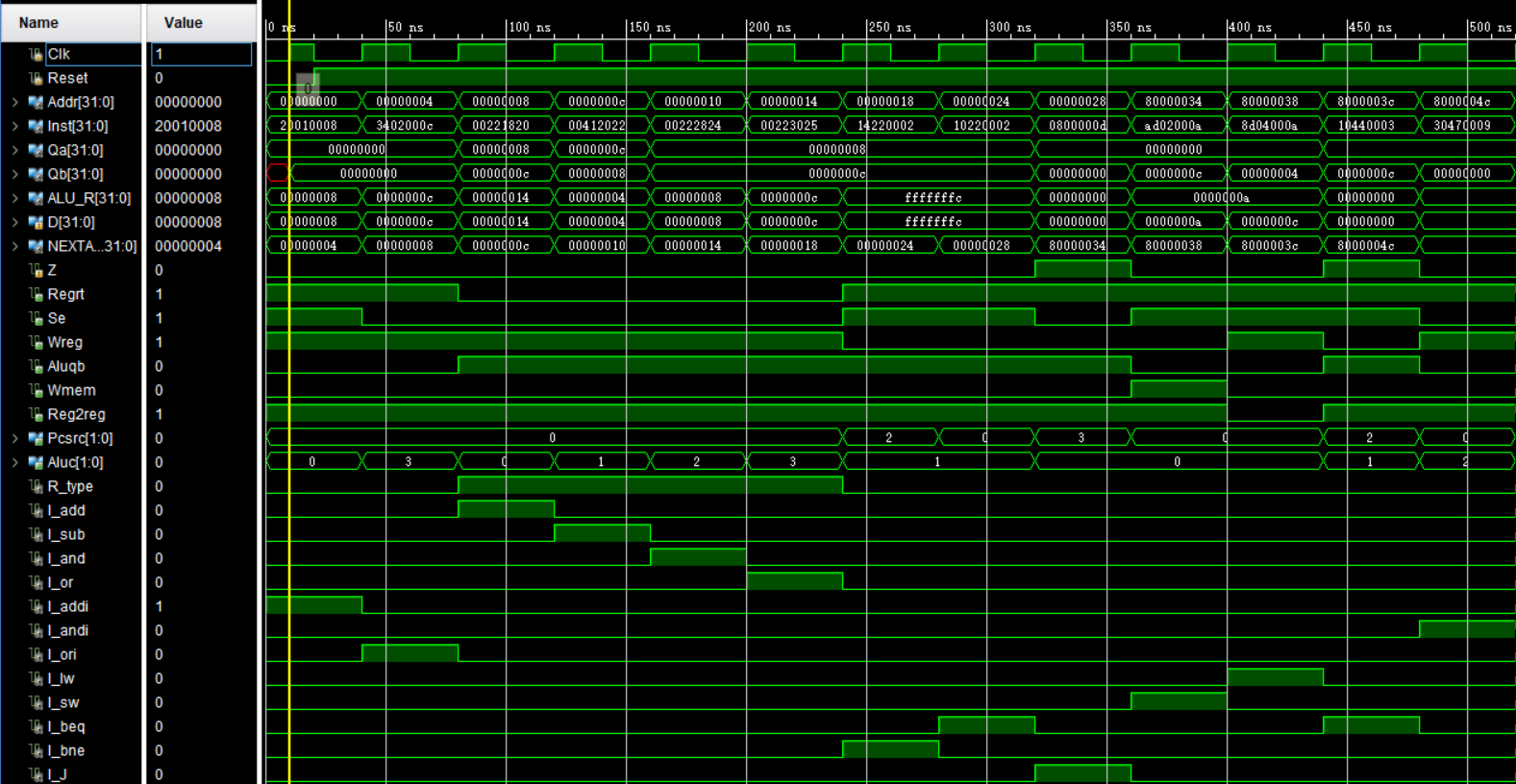
**forever #20 Clk=~Clk;**

**end**

**endmodule**

1. **仿真模拟分析**

**5.1 仿真波形图**



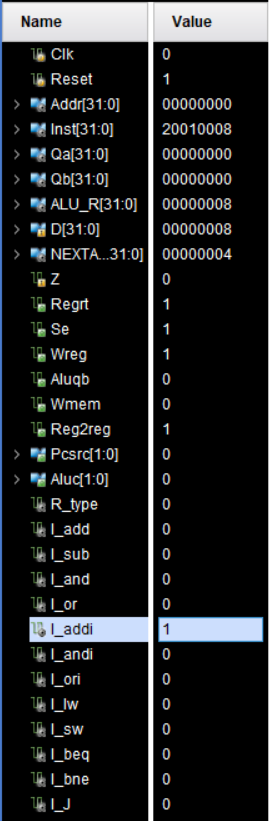
**5.1 指令代码分析**

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



1. 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

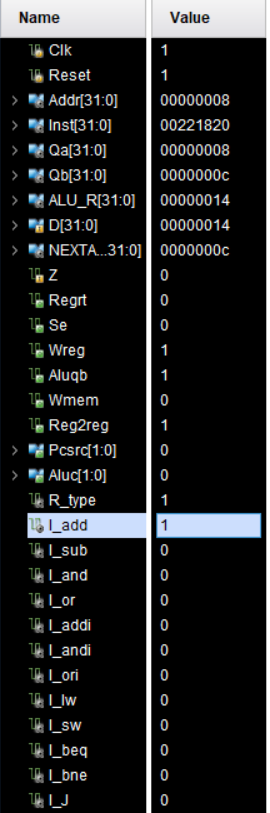


1. assign Rom[5'h02]=32'h00221820;

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

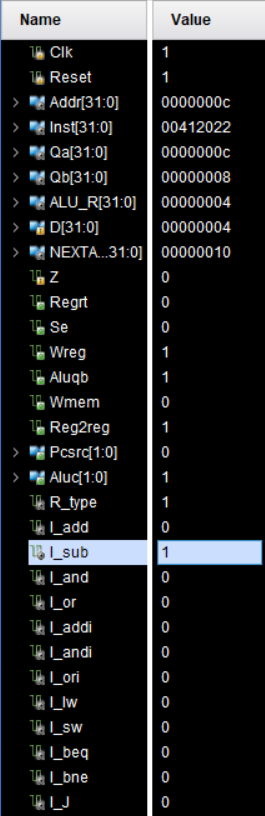


1. assign Rom[5'h03]=32'h00412022;

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

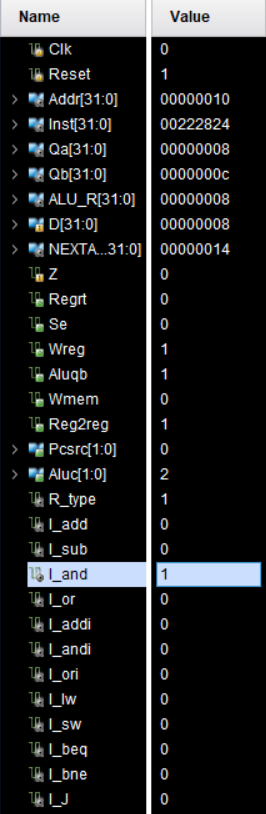


1. assign Rom[5'h04]=32'h00222824;

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

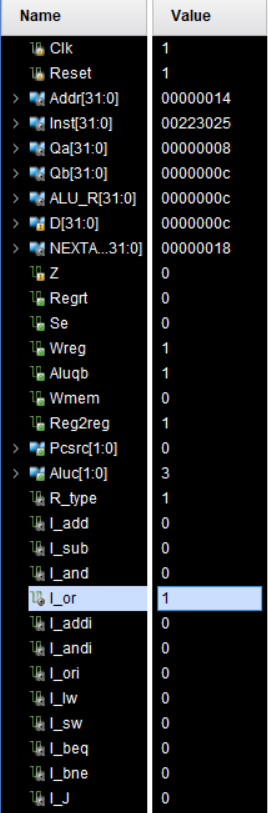


1. assign Rom[5'h05]=32'h00223025;

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



1. 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。

）



1. assign Rom[5'h07]=32'hXXXXXXXX;
2. assign Rom[5'h08]=32'hXXXXXXXX;
3. 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

1. assign Rom[5'h0A]=32'h0800000D;

二进制源码：000010 00000000000000000000001101

指令含义：J OD

结果：跳转到第14条指令的地址位置

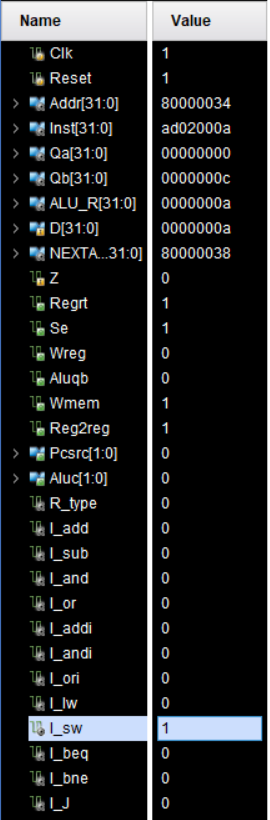


1. assign Rom[5'h0B]=32'hXXXXXXXX;
2. assign Rom[5'h0C]=32'hXXXXXXXX;
3. assign Rom[5'h0D]=32'hAD02000A;

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。



1. 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

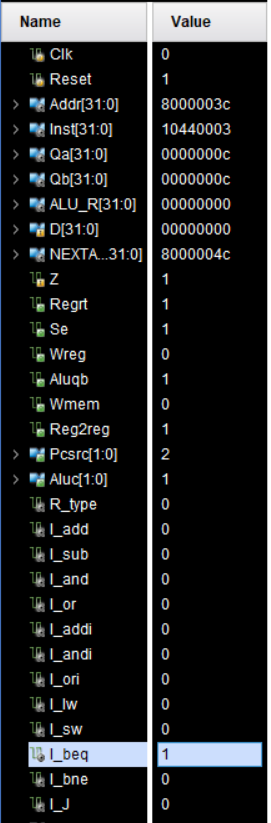


1. assign Rom[5'h0F]=32'h10440003;

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

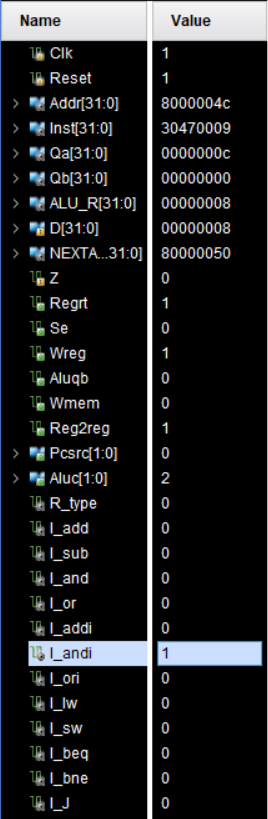


1. assign Rom[5'h10]=32'hXXXXXXXX;
2. assign Rom[5'h11]=32'hXXXXXXXX;
3. assign Rom[5'h12]=32'hXXXXXXXX;
4. assign Rom[5'h13]=32'h30470009;

二进制源码：001100 00010 00111 0000000000001001

指令含义：andi $2,9,$7

结果：$7=b1100&b1001=b1000=8



1. assign Rom[5'h14]=32'hXXXXXXXX;
2. assign Rom[5'h15]=32'hXXXXXXXX;
3. assign Rom[5'h16]=32'hXXXXXXXX;
4. assign Rom[5'h17]=32'hXXXXXXXX;
5. assign Rom[5'h18]=32'hXXXXXXXX;
6. assign Rom[5'h19]=32'hXXXXXXXX;
7. assign Rom[5'h1A]=32'hXXXXXXXX;
8. assign Rom[5'h1B]=32'hXXXXXXXX;
9. assign Rom[5'h1C]=32'hXXXXXXXX;
10. assign Rom[5'h1D]=32'hXXXXXXXX;
11. assign Rom[5'h1E]=32'hXXXXXXXX;
12. 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

1. **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。