电子科技大学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

实 验 报 告 书

题　　目　基于 MIPS 的 32 位流水线 CPU 设计

成　　员　朱若愚　陈家乐

指导老师　桂盛霖

目　录

一、设计目的与目标

1.1 实验内容

（1）基于 MIPS 指令集的 32 位流水线 CPU 的指令格式的拟定；

（2）基本功能部件的设计与实现；

（3）流水线 CPU 各主要功能部件的设计与实现；

（4）流水线 CPU 的封装；

（5）对流水线 CPU 进行仿真测试（相比单周期 CPU，着重体现数据冒险，控制冒险的解决方案）。

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。

（2）额外支持 xori、xor、sll、srl、sra、lui、jal 和 jr 八条指令，即完全实现教材上的全部二十条指令。

（3）支持解决 lw 指令、一般寄存器读写等的数据冒险，实现内部前推以及暂停流水线的功能；支持 beq 条件分支指令及其下一条指令存在的控制冒险，实现阻塞三个时钟周期保证正确分支指令的功能

二、课程设计器材

2.1 硬件平台

无

2.2 软件平台

操作系统：Win 11

开发平台：Modesilm

编程语言：Verilog

三、流水线 CPU 逻辑设计总体方案

基于 MIPS 指令集的 32 位单周期 CPU 由 INSTMEM 指令存储单元、CPU 运算与控制单元以及 DATAMEM 数据存储单元组成。在整个 CPU 进行运算处理的过程中，通常是由 INSTMEM 中取出相应的操作指令输出到 CPU 中进行相关的运算后，再决定是否输出到 DATAMEM 中进行存储，流水线 CPU 基于单周期 CPU 的数据通路划分为 5 个阶段以提高 CPU 性能。

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；流水线 CPU 通过重叠指令的执行过程来提升指令段的整体执行速度，需要在每个阶段引入流水线寄存器组，要求合理划分阶段，使得各阶段所需时间大致相同。

3.1 指令模块

流水线（Pipeline）CPU 指 CPU 按照单周期方式顺序执行多条指令和采用流水线重叠执行多条指令的处理器设计

指令的执行过程包括：取指令→分析指令→执行指令→保存结果。

流水线各级寄存器执行过程包括：指令流入→取指→译码→执行→指令流出。

3.1.1 MIPS 指令格式

本实验所选用 MIPS 的指令格式为 32 位。

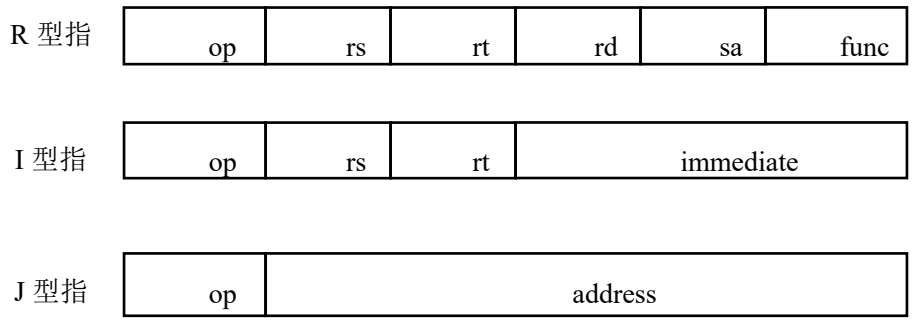

图 3-1 MIPS 指令格式

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。列出了本实验的所涉及到的 20 条 MIPS 指令。

表 3-1 MIPS 指令详细编码

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

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。

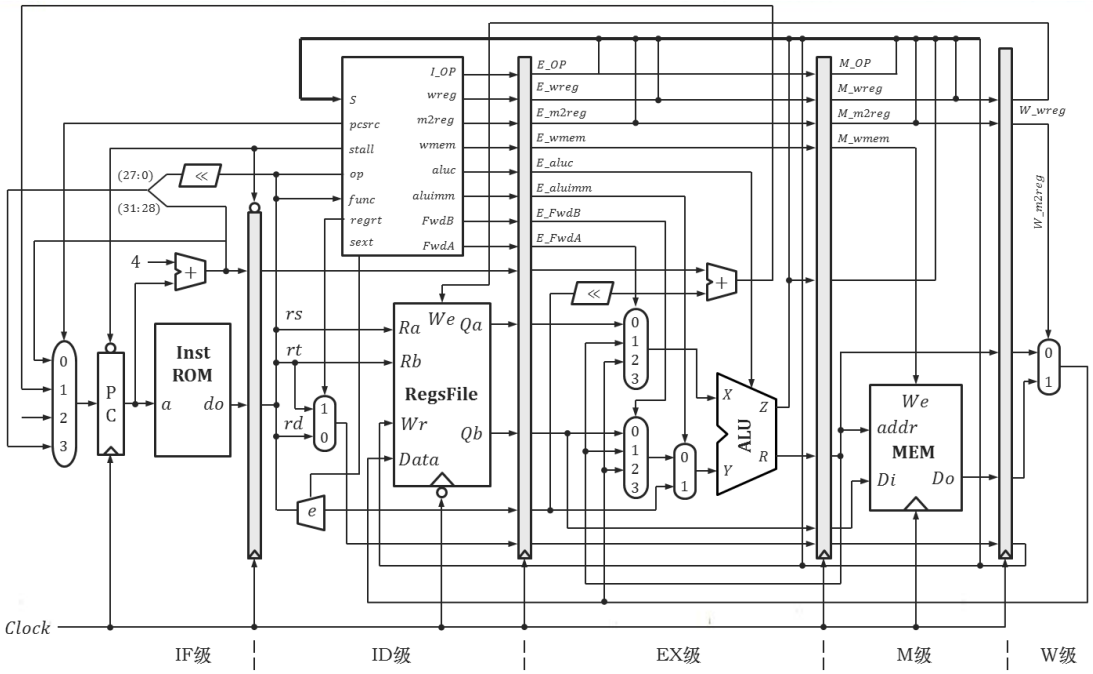

图 3-2 本实验的总体结构图

指令类型	IF	ID	EX	MEM	WB	总时间
取数（lw）	200 ps	100 ps	80 ps	200 ps	100 ps	680 ps
存数（sw）	200 ps	100 ps	80 ps	200 ps	—	580 ps
R 型（add, sub, and, or, slt）	200 ps	100 ps	80 ps	—	100 ps	480 ps
分支（beq）	200 ps	100 ps	80 ps	—	—	380 ps

图 3-3 各个指令在各部分所需的执行时间

按照执行时间需要如下设计指令处理：5 级流水线的每级所需时间按照耗时

最多的部分时间统一设定，即 200ps。把时钟周期的长度从单周期模型的 800ps 降为 200ps，每个时钟周期完成流水线中一级的执行。

根据实验原理中的流水线 CPU 总体结构图，我们可以清楚的知道除了单周期 CPU 的设计应包括 PC，PCAdd4，INSTMEM，CONUNIT，REGFILE，ALU，DATAMEM，EXT16T32 这几个核心模块，还包括 IF/ID 级流水线寄存器堆，ID/EX 级流水线寄存器堆，EX/MEM 级流水线寄存器堆，MEM/WB 级流水线寄存器堆模块使得单周期模型机的 5 个部分拓展为流水线的 5 级。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 SHIFTERL2,一个固定左移两位的移位器 SHIFT32_L2，一个四选一多路选择器 MUX4X32，一个数据扩展器 EXT16T32。为了简便，此次实验绝大部分多位二选一选择器均有三元运算符号（？：）来代替。

四、模块详细设计

4.1 PC

4.1.1 所处位置

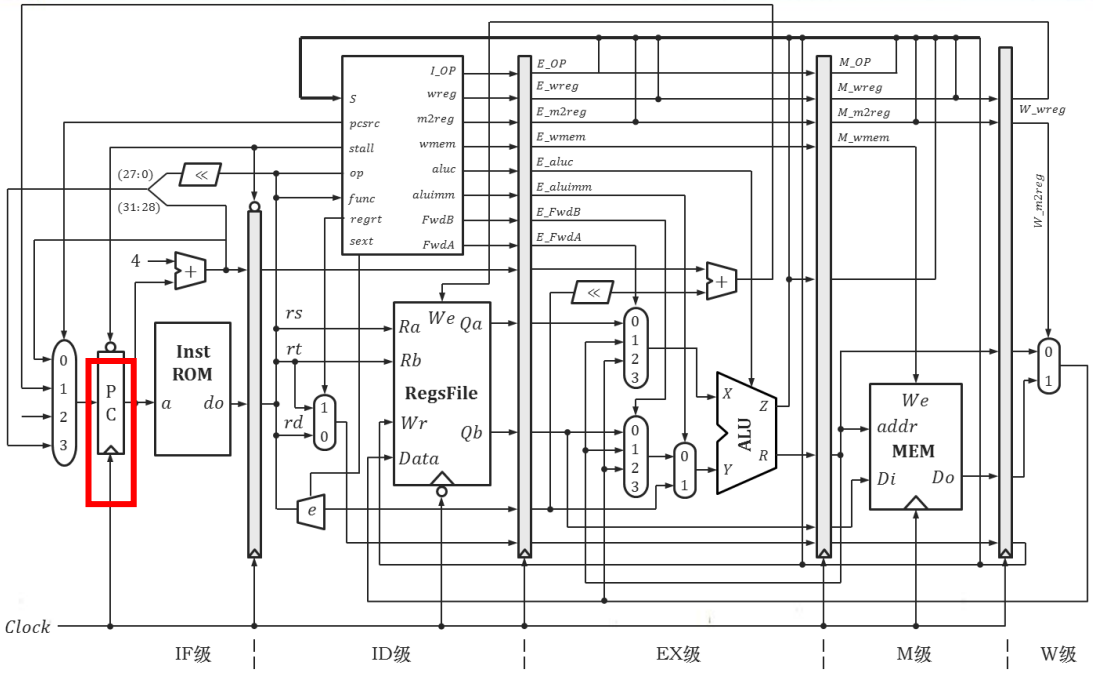

图 4-1 PC

4.1.2 模块功能

用于给出指令在指令储存器中的地址，实现数据冒险的停止写入。

4.1.3 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；为实现 lw 指令的数据冒险检

测，以及关闭 PC、IF/ID 流水线寄存器组的写使能信号和将 ID/EX 流水线寄存器组的 Clrn 端口清 0，需要在 COMUNIT 基础上增加一个输入端口 E_R2 和一个输出端口 stall，stall 端口输出高电平以及仅当 lw 指令的条件执行写停功能。

4.1.4 引脚及控制信号

Clk：时钟周期，输入信号

Clrn：控制信号，输入信号，用于 PC 寄存器清零

IF_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

IF_Addr：指令地址，输出信号

Stall：控制信号，输出信号，用于 PC 寄存器写停

4.1.5 主要实现代码

```
module PC(IF_Result,Clk,En,Clrn,IF_Addr,stall);
input [31:0]IF_Result;
input Clk,En,Clrn,stall;
output [31:0]IF_Addr;
wire [31:0]IF_Addr_n;
wire En_S;
assign En_S=En&~stall;
D_FFEC32 pc(IF_Result,Clk,En_S,Clrn,IF_Addr,IF_Addr_n);
endmodule
module D_FFEC32(D,Clk,En,Clrn,Q,Qn);
```

4.2 ADD4

4.2.1 所处位置

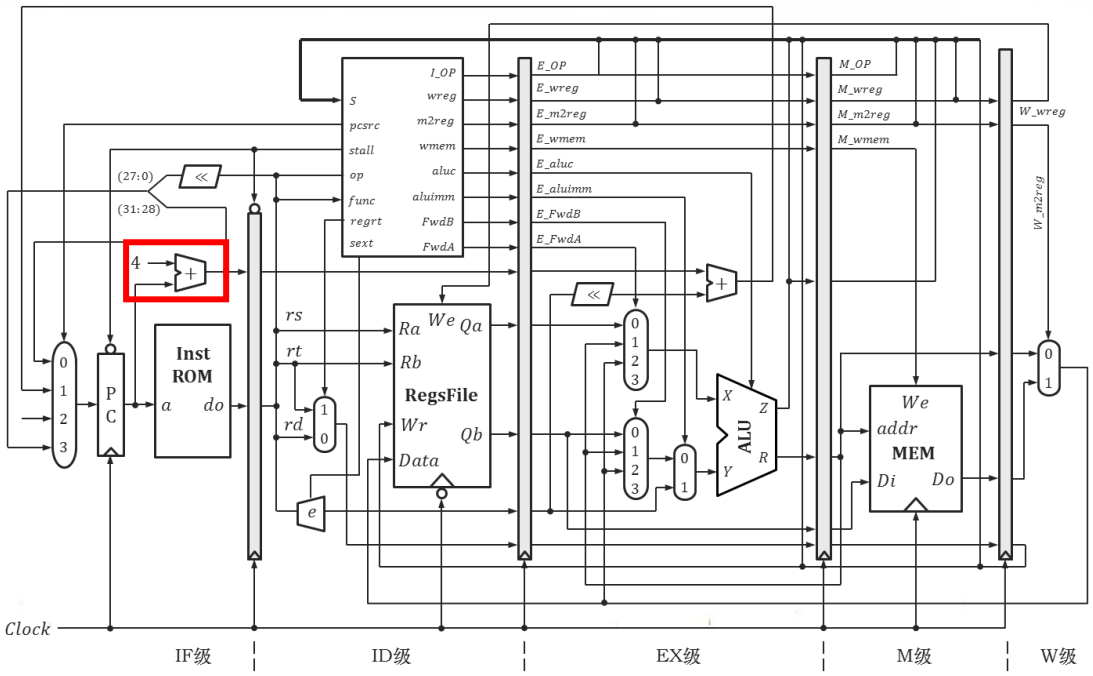

图 4-2 ADD4

4.2.2 模块功能

作为 PC 寄存器的更新信号。

4.2.3 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 IF/ID 流水线寄存器堆。

4.2.4 引脚及控制信号

IF_Addr:当前指令地址，输入端口

IF_PCadd4:下一条指令地址，输出端口

4.2.5 主要实现代码

```
module PCadd4(IF_Addr,IF_PCadd4);
input [31:0] IF_Addr;
output [31:0] IF_PCadd4;
assign IF_PCadd4=IF_Addr + 32'b00000000000000000000000000000100;
endmodule
```

4.3 INSTMEM

4.3.1 所处位置

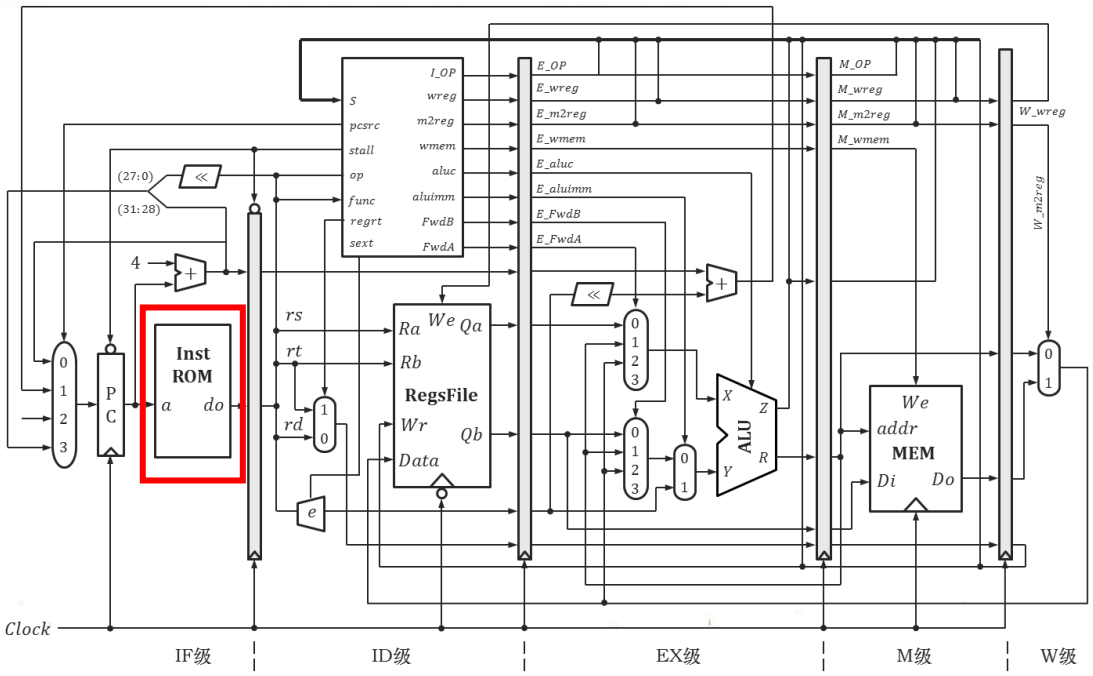

图 4-3 INSTMEM

4.3.2 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。

4.3.3 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，且本次实验着重对各种冒险进行编写指令。

4.3.4 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

4.3.5 主要实现代码

```
module INSTMEM(Addr,Inst);
    input[31:0]Addr;
    output[31:0]Inst;
    wire[31:0]Rom[31:0];
    assign Rom[5'h00]=32'b001101_00000_00011_0000_0000_0000_1011;//ori $3,$0,12
$3=11
   assign Rom[5'h01]=32'b001000_00000_00010_0000_0000_0000_1100;//addi $2,$0,12
```

```
$2=12
    assign Rom[5'h02]=32'b000000_00010_00011_10100_00000_100100;//and $2,$3,$20
$20=8
    assign Rom[5'h03]=32'b100011_10100_01101_0000000000001100;//lw $13 12($20)
$13=5
    assign Rom[5'h04]=32'b000000_01101_00011_00110_00000_100010;//sub $13,$3,$6
$6=fffffa
    assign Rom[5'h05]=32'b001100_01101_01000_1111_1111_1111_1111;//andi
$8,$2,65535 $8=5
    assign Rom[5'h06]=32'b001000_00000_00001_0000_0000_0000_1000;//addi $1,$0,6
$1=8
    assign Rom[5'h07]=32'b000100_00001_10100_0000000000000010;///beq $1 $7 2
    assign Rom[5'h08]=32'b001000_00000_00100_0000_0000_0000_0110;//addi $4,$0,6
$4=6
    assign Rom[5'h09]=32'b001000_00000_00101_0000_0000_0000_0110;//addi $5,$0,6
$5=6
    assign Rom[5'h0A]=32'b001000_00000_00111_0000_0000_0000_0110;//addi $7,$0,6
$7=6
    assign Inst=Rom[Addr[6:2]];
endmodule
```

4.4 DATAMEM

4.4.1 所处位置

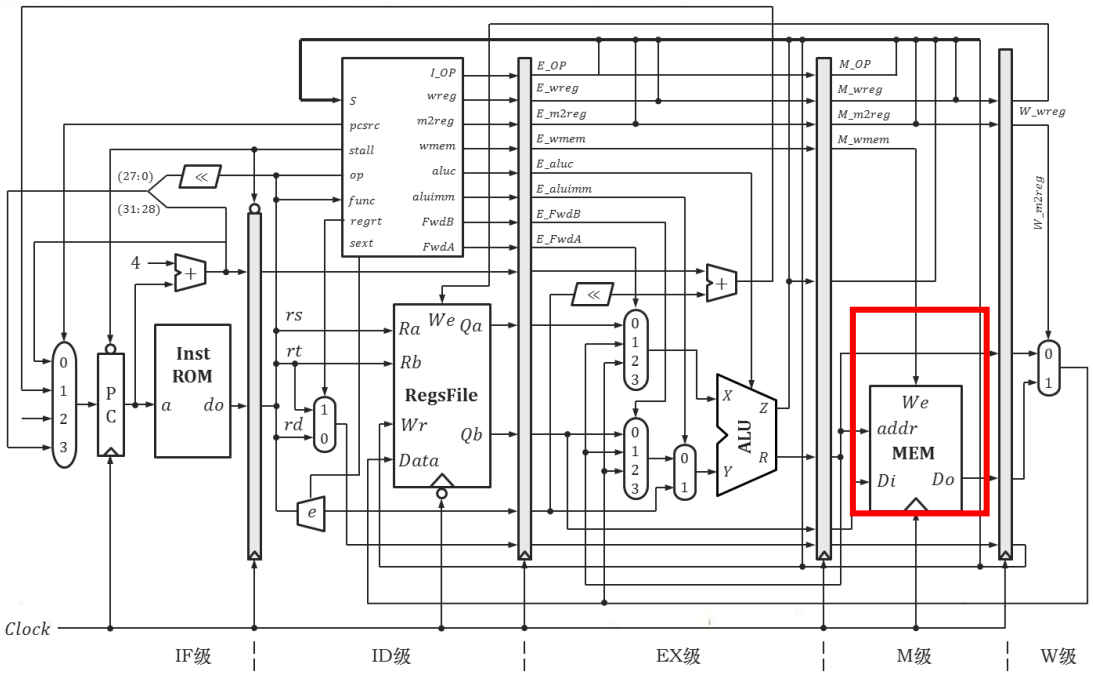

图 4-4 DATAMEM

4.4.2 模块功能

数据存储器，通过经过 EX/MEM 流水线寄存器的控制信号，对数据寄存器进行读或者写操作，并且此处模块输出 Dout 的数据选择器接入 MEM/WB 级流水线寄存器堆。

4.4.3 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。

4.4.4 引脚及控制信号

表 4-1 DATAMEM 引脚功能

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，或者当数据冲突调用读取相关寄存器，对应 FwdA，FwdB 值进行内部前推。

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，当 lw 指令处于 WB 级的时钟周期内，下条指令执行 R 型指令，E_FwdA=2，E_FwdB=0，E_aluimm= 0，W_m2reg=1，regrt=0，并且阻塞一个时钟周期。

4.4.5 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

4.5.1 所处位置

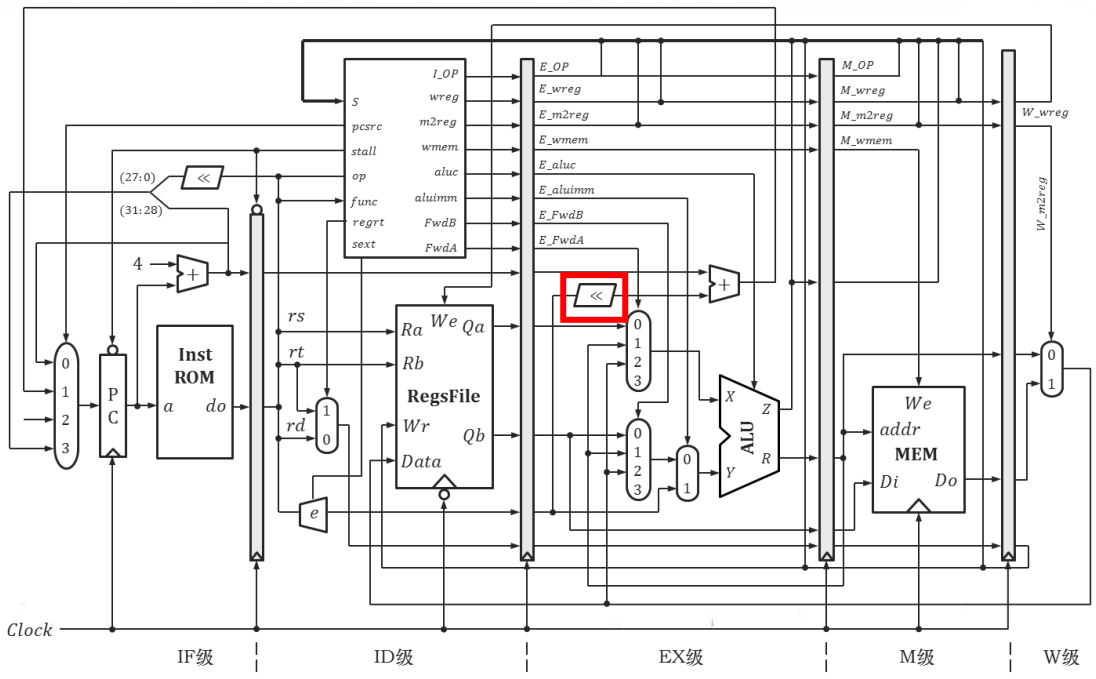

图 4-5 SHIFTER32_L2

4.5.2 模块功能

一个固定左移两位的移位器

4.5.3 实现思路

使用 32 位移位器 SHIFTER32，固定左移两位即可

4.5.4 引脚及控制信号

in：指令中的偏移量，输入信号

out：偏移量左移后的结果，输出信号

4.5.5 主要实现代码

```
module SHIFTER32_L2(in,out);
input [31:0]in;
output [31:0]out;
assign out=(in<<2);
endmodule
```

4.6 SHIFTER26_L2

4.6.1 所处位置

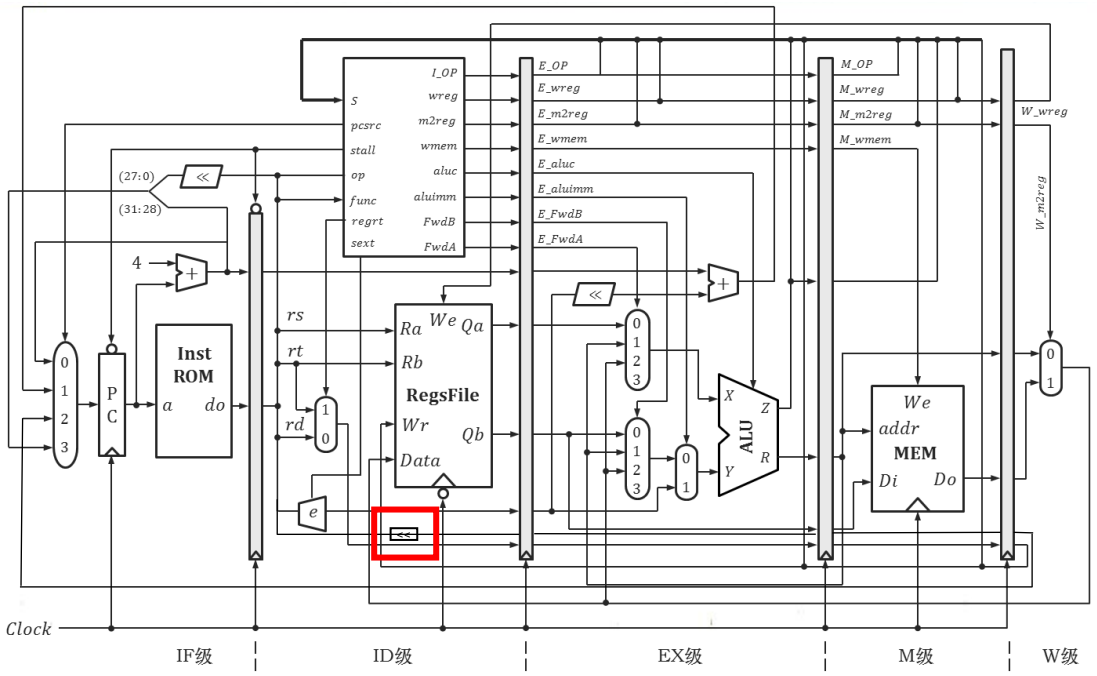

图 4-6 SHIFTER26_L2

4.6.2 模块功能

J 指令中用以产生跳转的目标地址

4.6.3 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.

4.6.4 引脚及控制信号

in：指令编码的低 26 位字段，输入信号。

add：PC+4 的 32 位字段，输入信号。

out：32 位转移目标地址，输出信号。

4.6.5 主要实现代码

```
module SHIFTER26_L2(in,add,out);
input [25:0] in;
input [3:0]add;
output [31:0]out;
wire[27:0] internal;
assign internal=(in<<2);
assign out={add,internal};
endmodule
```

4.7 MUX4X32

4.7.1 所处位置

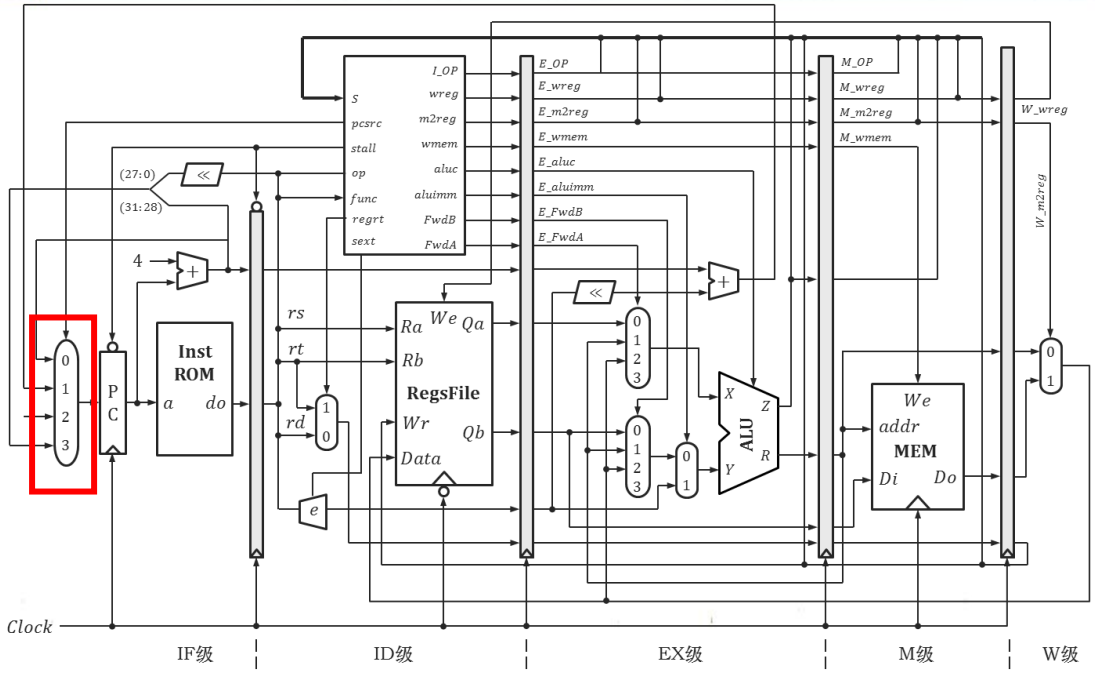

图 4-7 MUX4X32

4.7.2 模块功能

实现目标地址的选择

4.7.3 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，且增加了额外八条指令，因此采用一个 32 位四选一多路选择器

4.7.4 引脚及控制信号

A0：PC+4 的地址，输入信号

A1：beq 和 bne 指令的跳转地址，输入信号

A2:Jr 的跳转地址，由 REGFILE 直接输出，输入信号

A3：J 指令的跳转地址，输入信号

S：对地址进行选择的控制信号，输入信号

Y：目标地址，输出信号

4.7.5 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 EXT16T32

4.8.1 所处位置

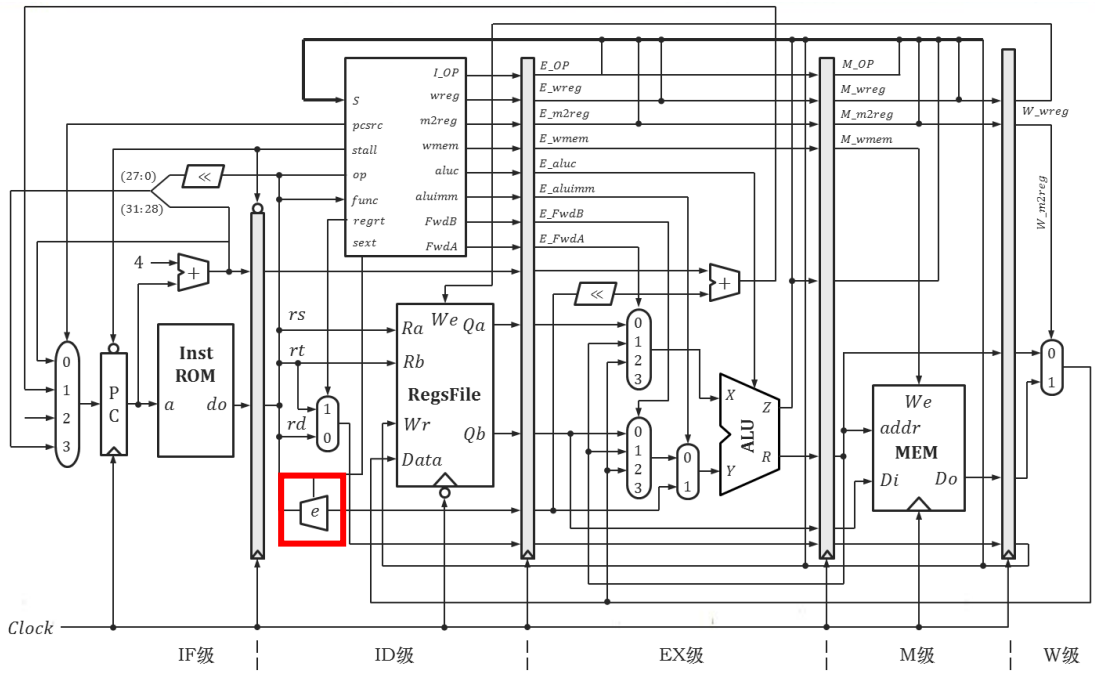

图 4-8 EXT16T32

4.8.2 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。

4.8.3 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

4.8.3 引脚及控制信号

X：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。

Y：扩展后的立即数，输出信号。

4.8.5 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
```

4.9 CONUNIT

4.9.1 所处位置

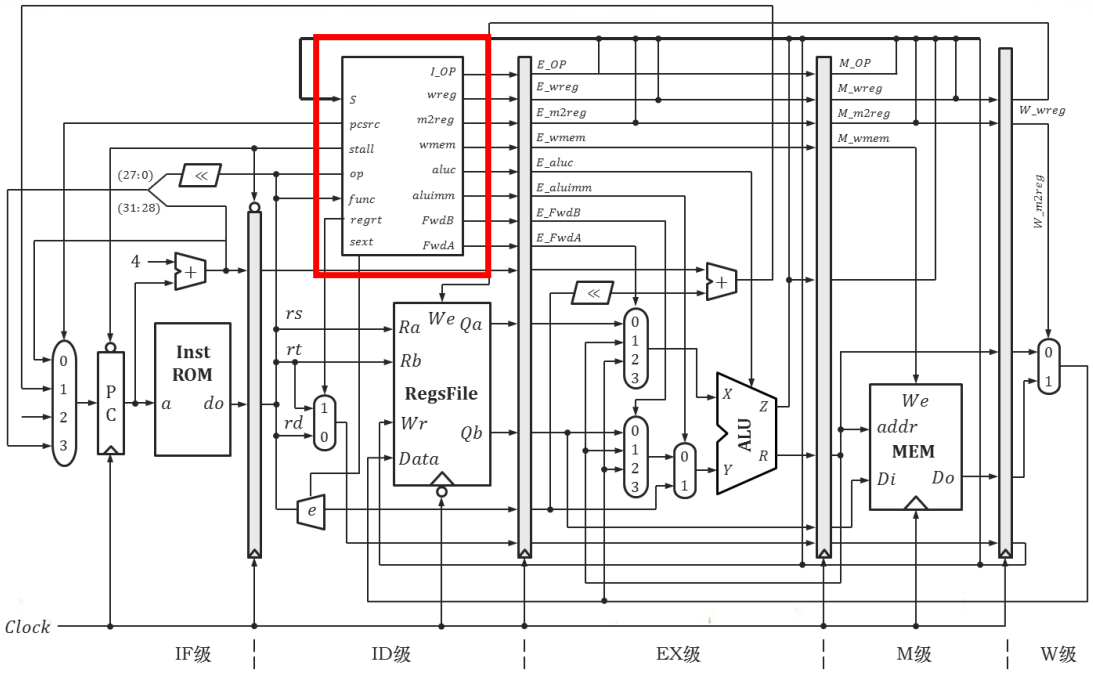

图 4-9 CONUNIT

4.9.2 模块功能

控制器是作为流水线 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。

4.9.3 实现思路

根据 Inst 输入到 CONUNIT 的 Op 段和 Func 段，ID/EX 级流水线寄存器输入到 CONUNIT 的 E_Op,E_Func,E_Wreg ,E_Reg2reg,E_Rd 段，EX/MEM 级流水线寄存器输入到 CONUNIT 的 M_Wreg，M_Rd,通过运算决定各个选择器的选择信号。控制部件 CONUNIT 新增输出端口 Condep，输出取反后连接到流水线寄存器组 IF/ID 和 PC，判断 J 型指令跳转的写停。

4.9.4 引脚

Inst[31:26]：Op，E_Op，输入信号。

Inst[5:0]：Func,E_Func，输入信号。

E_Rd,M_Rd：输入信号，检测 lw 指令的数据冒险。

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。

E_Wreg,M_Wreg： 输入信号。

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。

E_Reg2reg，Reg2reg：控制 REHFILE 更新值的来源。

此外，为了实现额外的八条指令，新增两个信号 shift 和 j 用于控制。

j：控制 jal、jr 指令的写入，输出信号。

shift：控制位移指令输入到 ALU，输出信号。

FwdA,FwdB：控制输入 ALU 的输入信号，选择内部前推的寄存器返回值。

stall,condep：输出信号，用于流水线寄存器的 Clrn 端口低电平，使得内容清零，改为两条空指令。

4.9.5 主要实现代码

```
module
CONUNIT(Z,E_Op,Op,E_Func,Func,Rs,Rt,Regrt,Se,E_Rd,M_Rd,E_Reg2reg,E_Wreg,M_Wreg,
Wreg,Reg2reg,Pcsrc,Wmem,Aluc,Aluqb,shift,j,FwdA,FwdB,stall,condep);
input [5:0]Op,Func,E_Op,E_Func;
input Z;
input E_Wreg,M_Wreg,E_Reg2reg;
input [4:0]E_Rd,M_Rd,Rs,Rt;
output Regrt,Se,Wreg,Aluqb,Wmem,Reg2reg,stall,condep;
output [1:0]Pcsrc;
output [3:0]Aluc;
output shift;
output j;
output reg [1:0]FwdA,FwdB;
wire i_add = (Op == 6'b000000 & Func == 6'b100000)?1:0;
wire i_sub = (Op == 6'b000000 & Func == 6'b100010)?1:0;
wire i_and = (Op == 6'b000000 & Func == 6'b100100)?1:0;
wire i_or  = (Op == 6'b000000 & Func == 6'b100101)?1:0;
wire i_xor = (Op == 6'b000000 & Func == 6'b100110)?1:0;
wire i_sll = (Op == 6'b000000 & Func == 6'b000000)?1:0;
wire i_srl = (Op == 6'b000000 & Func == 6'b000010)?1:0;
wire i_sra = (Op == 6'b000000 & Func == 6'b000011)?1:0;
wire i_jr  = (Op == 6'b000000 & Func == 6'b001000)?1:0;
```

```
wire E_jr  = (E_Op == 6'b000000 & E_Func == 6'b001000)?1:0;
wire i_addi = (Op == 6'b001000)?1:0;
wire i_andi = (Op == 6'b001100)?1:0;
wire i_ori  = (Op == 6'b001101)?1:0;
wire i_xori = (Op == 6'b001110)?1:0;
wire i_lw   = (Op == 6'b100011)?1:0;
wire i_sw   = (Op == 6'b101011)?1:0;
wire i_beq  = (Op == 6'b000100)?1:0;
wire i_bne  = (Op == 6'b000101)?1:0;
wire i_lui  = (Op == 6'b001111)?1:0;
wire i_j    = (Op == 6'b000010)?1:0;
wire i_jal  = (Op == 6'b000011)?1:0;
wire E_j    = (E_Op == 6'b000010)?1:0;
wire E_jal  = (E_Op == 6'b000011)?1:0;
wire E_beq = (E_Op == 6'b000100)?1:0;
wire E_bne = (E_Op == 6'b000101)?1:0;
wire E_Inst = i_add|i_sub|i_and|i_or|i_sw|i_beq|i_bne;
assign Wreg =
i_add|i_sub|i_and|i_or|i_xor|i_sll|i_srl|i_sra|i_addi|i_andi|i_ori|i_or|i_xori|
i_lw|i_lui|i_jal;
assign Regrt =
i_addi|i_andi|i_ori|i_xori|i_lw|i_sw|i_lui|i_beq|i_bne|i_j|i_jal;
assign Reg2reg  =
i_add|i_sub|i_and|i_or|i_xor|i_sll|i_srl|i_sra|i_addi|i_andi|i_ori|i_xori|i_sw|
i_beq|i_bne|i_j|i_jal;
assign Aluqb = i_add | i_sub | i_and | i_or | i_xor | i_sll | i_srl | i_sra |
i_beq | i_bne |i_j;
assign Se   = i_addi | i_lw | i_sw | i_beq | i_bne;
assign Aluc[3] = i_sra;
assign Aluc[2] = i_xor |i_lui | i_sll | i_srl | i_sra |i_xori;
assign Aluc[1] = i_and | i_or | i_lui | i_srl | i_sra | i_andi | i_ori;
assign Aluc[0] = i_sub | i_ori | i_or | i_sll | i_srl |i_sra| i_beq | i_bne;
assign Wmem = i_sw;
assign Pcsrc[0] = E_jal | E_j | (E_beq&Z) | (E_bne&~Z);
assign Pcsrc[1] = E_j | E_jr | E_jal;
assign shift = i_sll | i_srl | i_sra;
assign j = i_jal | i_jr;
always@(E_Rd,M_Rd,E_Wreg,M_Wreg,Rs,Rt,i_add,i_sub,i_and,i_or,i_sw,i_beq,i_bne)b
```

```
egin
FwdA=2'b00;
if((Rs==E_Rd)&(E_Rd!=0)&(E_Wreg==1))begin
FwdA=2'b10;
end else begin
if((Rs==M_Rd)&(M_Rd!=0)&(M_Wreg==1))begin
FwdA=2'b01;
end
end
end
always@(E_Rd,M_Rd,E_Wreg,M_Wreg,Rs,Rt,i_add,i_sub,i_and,i_or,i_sw,i_beq,i_bne,i
_sll,i_xor,i_srl,i_sra)begin
FwdB=2'b00;
if((Rt==E_Rd)&((i_add==1)|(i_sub==1)|(i_and==1)|(i_or==1)|(i_sw==1)|(i_beq==1)|
(i_bne==1)|(i_xor==1)|(i_sra==1)|(i_srl==1)|(i_sll==1))&(E_Rd!=0)&(E_Wreg==1))b
egin
FwdB=2'b10;
end else begin
if((Rt==M_Rd)&((i_add==1)|(i_sub==1)|(i_and==1)|(i_or==1)|(i_sw==1)|(i_beq==1)|
(i_bne==1)|(i_xor==1)|(i_sra==1)|(i_srl==1)|(i_sll==1))&(M_Rd!=0)&(M_Wreg==1))b
egin
FwdB=2'b01;
end
end
end
assign stall=((Rs==E_Rd)|(Rt==E_Rd))&(E_Reg2reg==0)&(E_Rd!=0)&(E_Wreg==1);
assign condep=(E_jal)|(E_jr)|(E_beq&Z)|(E_bne&~Z);
endmodule
```

4.10 REGFILE

4.10.1 所处位置

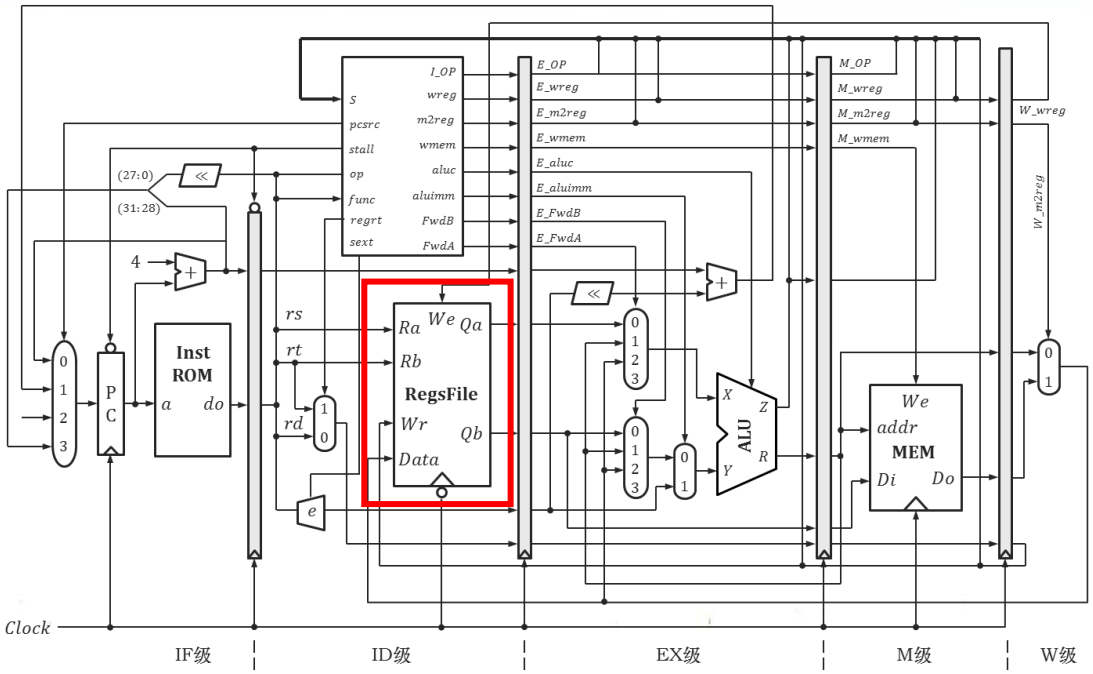

图 4-10 REGFILE

4.10.2 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。

4.10.3 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。

4.10.4 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。

4.10.5 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_reg
32,Q24_reg32,Q23_reg32,Q22_reg32,Q21_reg32,Q20_reg32,Q19_reg32,Q18_reg32,Q17_re
g32,Q16_reg32,Q15_reg32,Q14_reg32,Q13_reg32,Q12_reg32,Q11_reg32,Q10_reg32,Q9_re
g32,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_reg3
2,Q9_reg32,Q8_reg32,Q7_reg32,Q6_reg32,Q5_reg32,Q4_reg32,Q3_reg32,Q2_reg32,Q1_re
g32,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_reg3
2,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_reg3
2,Rb,Qb);
endmodule

module
REG32(D,En,Clk,Clrn,Q31,Q30,Q29,Q28,Q27,Q26,Q25,Q24,Q23,Q22,Q21,Q20,Q19,Q18,Q17
,Q16,Q15,Q14,Q13,Q12,Q11,Q10,Q9,Q8,Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0);
input [31:0]D,En;
```

```
input Clk,Clrn;
output
[31:0]Q31,Q30,Q29,Q28,Q27,Q26,Q25,Q24,Q23,Q22,Q21,Q20,Q19,Q18,Q17,Q16,Q15,Q14,Q
13,Q12,Q11,Q10,Q9,Q8,Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0;
wire[31:0]
Q31n,Q30n,Q29n,Q28n,Q27n,Q26n,Q25n,Q24n,Q23n,Q22n,Q21n,Q20n,Q19n,Q18n,Q17n,Q16n
,Q15n,Q14n,Q13n,Q12n,Q11n,Q10n,Q9n,Q8n,Q7n,Q6n,Q5n,Q4n,Q3n,Q2n,Q1n,Q0n;
D_FFEC32 q31(D,Clk,En[31],Clrn,Q31,Q31n);
D_FFEC32 q30(D,Clk,En[30],Clrn,Q30,Q30n);
D_FFEC32 q29(D,Clk,En[29],Clrn,Q29,Q29n);
D_FFEC32 q28(D,Clk,En[28],Clrn,Q28,Q28n);
D_FFEC32 q27(D,Clk,En[27],Clrn,Q27,Q27n);
D_FFEC32 q26(D,Clk,En[26],Clrn,Q26,Q26n);
D_FFEC32 q25(D,Clk,En[25],Clrn,Q25,Q25n);
D_FFEC32 q24(D,Clk,En[24],Clrn,Q24,Q24n);
D_FFEC32 q23(D,Clk,En[23],Clrn,Q23,Q23n);
D_FFEC32 q22(D,Clk,En[22],Clrn,Q22,Q22n);
D_FFEC32 q21(D,Clk,En[21],Clrn,Q21,Q21n);
D_FFEC32 q20(D,Clk,En[20],Clrn,Q20,Q20n);
D_FFEC32 q19(D,Clk,En[19],Clrn,Q19,Q19n);
D_FFEC32 q18(D,Clk,En[18],Clrn,Q18,Q18n);
D_FFEC32 q17(D,Clk,En[17],Clrn,Q17,Q17n);
D_FFEC32 q16(D,Clk,En[16],Clrn,Q16,Q16n);
D_FFEC32 q15(D,Clk,En[15],Clrn,Q15,Q15n);
D_FFEC32 q14(D,Clk,En[14],Clrn,Q14,Q14n);
D_FFEC32 q13(D,Clk,En[13],Clrn,Q13,Q13n);
D_FFEC32 q12(D,Clk,En[12],Clrn,Q12,Q12n);
D_FFEC32 q11(D,Clk,En[11],Clrn,Q11,Q11n);
D_FFEC32 q10(D,Clk,En[10],Clrn,Q10,Q10n);
D_FFEC32 q9(D,Clk,En[9],Clrn,Q9,Q9n);
D_FFEC32 q8(D,Clk,En[8],Clrn,Q8,Q8n);
D_FFEC32 q7(D,Clk,En[7],Clrn,Q7,Q7n);
D_FFEC32 q6(D,Clk,En[6],Clrn,Q6,Q6n);
D_FFEC32 q5(D,Clk,En[5],Clrn,Q5,Q5n);
D_FFEC32 q4(D,Clk,En[4],Clrn,Q4,Q4n);
D_FFEC32 q3(D,Clk,En[3],Clrn,Q3,Q3n);
D_FFEC32 q2(D,Clk,En[2],Clrn,Q2,Q2n);
D_FFEC32 q1(D,Clk,En[1],Clrn,Q1,Q1n);
```

```
assign Q0=0;
endmodule

module D_FFEC32(D,Clk,En,Clrn,Q,Qn);
 input [31:0]D;
 input Clk,En,Clrn;
 output [31:0]Q,Qn;
 D_FFEC d0(D[0],Clk,En,Clrn,Q[0],Qn[0]);
 D_FFEC d1(D[1],Clk,En,Clrn,Q[1],Qn[1]);
 D_FFEC d2(D[2],Clk,En,Clrn,Q[2],Qn[2]);
 D_FFEC d3(D[3],Clk,En,Clrn,Q[3],Qn[3]);
 D_FFEC d4(D[4],Clk,En,Clrn,Q[4],Qn[4]);
 D_FFEC d5(D[5],Clk,En,Clrn,Q[5],Qn[5]);
 D_FFEC d6(D[6],Clk,En,Clrn,Q[6],Qn[6]);
 D_FFEC d7(D[7],Clk,En,Clrn,Q[7],Qn[7]);
 D_FFEC d8(D[8],Clk,En,Clrn,Q[8],Qn[8]);
 D_FFEC d9(D[9],Clk,En,Clrn,Q[9],Qn[9]);
 D_FFEC d10(D[10],Clk,En,Clrn,Q[10],Qn[10]);
 D_FFEC d11(D[11],Clk,En,Clrn,Q[11],Qn[11]);
 D_FFEC d12(D[12],Clk,En,Clrn,Q[12],Qn[12]);
 D_FFEC d13(D[13],Clk,En,Clrn,Q[13],Qn[13]);
 D_FFEC d14(D[14],Clk,En,Clrn,Q[14],Qn[14]);
 D_FFEC d15(D[15],Clk,En,Clrn,Q[15],Qn[15]);
 D_FFEC d16(D[16],Clk,En,Clrn,Q[16],Qn[16]);
 D_FFEC d17(D[17],Clk,En,Clrn,Q[17],Qn[17]);
 D_FFEC d18(D[18],Clk,En,Clrn,Q[18],Qn[18]);
 D_FFEC d19(D[19],Clk,En,Clrn,Q[19],Qn[19]);
 D_FFEC d20(D[20],Clk,En,Clrn,Q[20],Qn[20]);
 D_FFEC d21(D[21],Clk,En,Clrn,Q[21],Qn[21]);
 D_FFEC d22(D[22],Clk,En,Clrn,Q[22],Qn[22]);
 D_FFEC d23(D[23],Clk,En,Clrn,Q[23],Qn[23]);
 D_FFEC d24(D[24],Clk,En,Clrn,Q[24],Qn[24]);
 D_FFEC d25(D[25],Clk,En,Clrn,Q[25],Qn[25]);
 D_FFEC d26(D[26],Clk,En,Clrn,Q[26],Qn[26]);
 D_FFEC d27(D[27],Clk,En,Clrn,Q[27],Qn[27]);
 D_FFEC d28(D[28],Clk,En,Clrn,Q[28],Qn[28]);
 D_FFEC d29(D[29],Clk,En,Clrn,Q[29],Qn[29]);
 D_FFEC d30(D[30],Clk,En,Clrn,Q[30],Qn[30]);
```

```
 D_FFEC d31(D[31],Clk,En,Clrn,Q[31],Qn[31]);
 endmodule

module D_FFEC(D,Clk,En,Clrn,Q,Qn);
input D,Clk,En,Clrn;
output Q,Qn;
wire Y0,Y_C;
MUX2X1 m0(Q,D,En,Y0);
and i0(Y_C,Y0,Clrn);
D_FF d0(Y_C,Clk,Q,Qn);
endmodule

module D_FF(D,Clk,Q,Qn);
input D,Clk;
output Q,Qn;
wire Clkn,Q0,Qn0;
not i0(Clkn,Clk);
D_Latch d0(D,Clkn,Q0,Qn0);
D_Latch d1(Q0,Clk,Q,Qn);
endmodule

module D_Latch(D,En,Q,Qn);
input D,En;
output Q,Qn;
wire Sn,Rn,Dn;
not i0(Dn,D);
nand i1(Sn,D,En);
nand i2(Rn,En,Dn);
nand i3(Q,Sn,Qn);
nand i4(Qn,Q,Rn);
endmodule

module D_FF(D,Clk,Q,Qn);
input D,Clk;
output Q,Qn;
wire Clkn,Q0,Qn0;
not i0(Clkn,Clk);
D_Latch d0(D,Clkn,Q0,Qn0);
```

```
D_Latch d1(Q0,Clk,Q,Qn);
endmodule

module
MUX32X32(Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10,Q11,Q12,Q13,Q14,Q15,Q16,Q17,Q18,Q19,
Q20,Q21,Q22,Q23,Q24,Q25,Q26,Q27,Q28,Q29,Q30,Q31,R,Q);
input [4:0]R;
input
[31:0]Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10,Q11,Q12,Q13,Q14,Q15,Q16,Q17,Q18,Q19,Q20
,Q21,Q22,Q23,Q24,Q25,Q26,Q27,Q28,Q29,Q30,Q31;
output [31:0]Q;
function [31:0] select;
input
[31:0]Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10,Q11,Q12,Q13,Q14,Q15,Q16,Q17,Q18,Q19,Q20
,Q21,Q22,Q23,Q24,Q25,Q26,Q27,Q28,Q29,Q30,Q31;
input [31:0]R;
case(R)
5'b00000:select=Q0;
5'b00001:select=Q1;
5'b00010:select=Q2;
5'b00011:select=Q3;
5'b00100:select=Q4;
5'b00101:select=Q5;
5'b00110:select=Q6;
5'b00111:select=Q7;
5'b01000:select=Q8;
5'b01001:select=Q9;
5'b01010:select=Q10;
5'b01011:select=Q11;
5'b01100:select=Q12;
5'b01101:select=Q13;
5'b01110:select=Q14;
5'b01111:select=Q15;
5'b10000:select=Q16;
5'b10001:select=Q17;
5'b10010:select=Q18;
5'b10011:select=Q19;
5'b10100:select=Q20;
```

```
5'b10101:select=Q21;
5'b10110:select=Q22;
5'b10111:select=Q23;
5'b11000:select=Q24;
5'b11001:select=Q25;
5'b11010:select=Q26;
5'b11011:select=Q27;
5'b11100:select=Q28;
5'b11101:select=Q29;
5'b11110:select=Q30;
5'b11111:select=Q31;
endcase
endfunction
assign
Q=select(Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10,Q11,Q12,Q13,Q14,Q15,Q16,Q17,Q18,Q19,
Q20,Q21,Q22,Q23,Q24,Q25,Q26,Q27,Q28,Q29,Q30,Q31,R);
endmodule

module DEC5T32E(I,En,Y);
input[4:0] I;
output reg[31:0] Y;
input En;
always @(I,En) begin
if (En) begin
case (I)
5'b00000: Y = 32'b0000_0000_0000_0000_0000_0000_0000_0001;
5'b00001: Y = 32'b0000_0000_0000_0000_0000_0000_0000_0010;
5'b00010: Y = 32'b0000_0000_0000_0000_0000_0000_0000_0100;
5'b00011: Y = 32'b0000_0000_0000_0000_0000_0000_0000_1000;
5'b00100: Y = 32'b0000_0000_0000_0000_0000_0000_0001_0000;
5'b00101: Y = 32'b0000_0000_0000_0000_0000_0000_0010_0000;
5'b00110: Y = 32'b0000_0000_0000_0000_0000_0000_0100_0000;
5'b00111: Y = 32'b0000_0000_0000_0000_0000_0000_1000_0000;
5'b01000: Y = 32'b0000_0000_0000_0000_0000_0001_0000_0000;
5'b01001: Y = 32'b0000_0000_0000_0000_0000_0010_0000_0000;
5'b01010: Y = 32'b0000_0000_0000_0000_0000_0100_0000_0000;
5'b01011: Y = 32'b0000_0000_0000_0000_0000_1000_0000_0000;
5'b01100: Y = 32'b0000_0000_0000_0000_0001_0000_0000_0000;
```

```
5'b01101: Y = 32'b0000_0000_0000_0000_0010_0000_0000_0000;
5'b01110: Y = 32'b0000_0000_0000_0000_0100_0000_0000_0000;
5'b01111: Y = 32'b0000_0000_0000_0000_1000_0000_0000_0000;
5'b10000: Y = 32'b0000_0000_0000_0001_0000_0000_0000_0000;
5'b10001: Y = 32'b0000_0000_0000_0010_0000_0000_0000_0000;
5'b10010: Y = 32'b0000_0000_0000_0100_0000_0000_0000_0000;
5'b10011: Y = 32'b0000_0000_0000_1000_0000_0000_0000_0000;
5'b10100: Y = 32'b0000_0000_0001_0000_0000_0000_0000_0000;
5'b10101: Y = 32'b0000_0000_0010_0000_0000_0000_0000_0000;
5'b10110: Y = 32'b0000_0000_0100_0000_0000_0000_0000_0000;
5'b10111: Y = 32'b0000_0000_1000_0000_0000_0000_0000_0000;
5'b11000: Y = 32'b0000_0001_0000_0000_0000_0000_0000_0000;
5'b11001: Y = 32'b0000_0010_0000_0000_0000_0000_0000_0000;
5'b11010: Y = 32'b0000_0100_0000_0000_0000_0000_0000_0000;
5'b11011: Y = 32'b0000_1000_0000_0000_0000_0000_0000_0000;
5'b11100: Y = 32'b0001_0000_0000_0000_0000_0000_0000_0000;
5'b11101: Y = 32'b0010_0000_0000_0000_0000_0000_0000_0000;
5'b11110: Y = 32'b0100_0000_0000_0000_0000_0000_0000_0000;
5'b11111: Y = 32'b1000_0000_0000_0000_0000_0000_0000_0000;
default: Y = 32'b0000_0000_0000_0000_0000_0000_0000_0000;
endcase
end
else begin
Y = 32'b0000_0000_0000_0000_0000_0000_0000_0000;
end
end
endmodule

module MUX2X1(A0,A1,S,Y);
input A0,A1,S;
output Y;
not i0(S_n,S);
nand i1(A0_S,A0,S_n);
nand i2(A1_S,A1,S);
nand i3(Y,A0_S,A1_S);
endmodule
```

4.11 ALU

4.11.1 所处位置

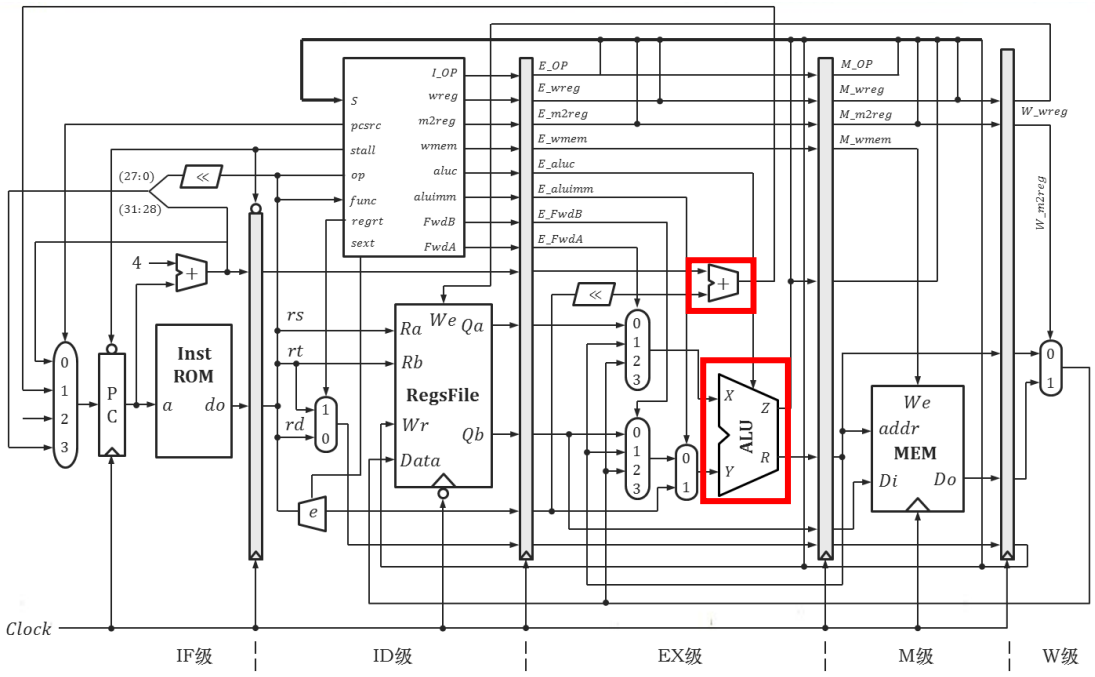

图 4-11 ALU（内含 32 位加法器）

4.11.2 模块功能

算数逻辑部件，需要实现加，减，按位与，按位或。

4.11.3 实现思路

由于新增了 lui 指令等需要 4 位控制信号控制运算类型，核心部件是 32 位加法器 ADDSUB_32。

4.11.4 引脚和控制信号

Qa：寄存器 1 的值或 Inst。

Y：寄存器 2 的值或立即数。

Aluc：4 位控制信号。

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。

4.11.5 主要实现代码

```
module ALU(X,Y,Aluc,R,Z);
input[31:0]X,Y;
input[3:0]Aluc;
output[31:0]R;
```

```
output Z;
wire[31:0]d_as,d_and,d_or,d_xor,d_lui,d_sh,d;
wire cout;
 ADDSUB_32 as32(X,Y,Aluc[0],d_as,cout);
assign d_and=X&Y;
assign d_or=X|Y;
assign d_xor=X^Y;
assign d_lui={Y[15:0],16'h0};
SHIFTER shift(Y,X[10:6],Aluc[3],Aluc[1],d_sh);
MUX6X32 select(d_and,d_or,d_xor,d_lui,d_sh,d_as,Aluc[3:0],R);
assign Z=~|R;
endmodule

module SHIFTER(X,Sa,Arith,Right,Sh);
input [31:0]X;
input [4:0]Sa;
input Arith,Right;
output [31:0]Sh;
wire [31:0]T4,T3,T2,T1,T0,S4,S3,S2,S1;
wire a=X[31]&Arith;
wire [15:0]e={16{a}};
parameter z=16'b0000000000000000;
wire [31:0]L1u,L1d,L2u,L2d,L3u,L3d,L4u,L4d,L5u,L5d;
assign L1u={X[15:0],z[15:0]};
assign L1d={e,X[31:16]};
MUX2X32 M1l(L1u,L1d,Right,T4);
MUX2X32 M1r(X,T4,Sa[4],S4);
assign L2u={S4[23:0],z[7:0]};
assign L2d={e[7:0],S4[31:8]};
MUX2X32 M2l(L2u,L2d,Right,T3);
MUX2X32 M2r(S4,T3,Sa[3],S3);
assign L3u={S3[27:0],z[3:0]};
assign L3d={e[3:0],S3[31:4]};
MUX2X32 M3l(L3u,L3d,Right,T2);
MUX2X32 M3r(S3,T2,Sa[2],S2);
assign L4u={S2[29:0],z[1:0]};
assign L4d={e[1:0],S2[31:2]};
MUX2X32 M4l(L4u,L4d,Right,T1);
```

```
MUX2X32 M4r(S2,T1,Sa[1],S1);
assign L5u={S1[30:0],z[0]};
assign L5d={e[0],S1[31:1]};
MUX2X32 M5l(L5u,L5d,Right,T0);
MUX2X32 M5r(S1,T0,Sa[0],Sh);
endmodule

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)
1'b0:select=A0;
1'b1:select=A1;
endcase
endfunction
assign Y = select(A0,A1,S);
endmodule

module MUX6X32(d_and,d_or,d_xor,d_lui,d_sh,d_as,Aluc,d);
input [31:0]d_and,d_or,d_xor,d_lui,d_sh,d_as;
input [3:0]Aluc;
output [31:0]d;
function [31:0]select;
input [31:0]d_and,d_or,d_xor,d_lui,d_sh,d_as;
input [3:0]Aluc;
case(Aluc)
4'b0000:select=d_as;
4'b0001:select=d_as;
4'b0010:select=d_and;
4'b0011:select=d_or;
4'b0100:select=d_xor;
4'b0110:select=d_lui;
4'b0101:select=d_sh;
4'b0111:select=d_sh;
```

```
4'b1111:select=d_sh;
4'b1101:select=d_sh;
endcase
endfunction
assign d=select(d_and,d_or,d_xor,d_lui,d_sh,d_as,Aluc);
endmodule
module ADDSUB_32(X, Y, Sub, S, Cout);
input [31:0] X;
input [31:0] Y;
input Sub;
output [31:0] S;
output Cout;
CLA_32 adder0 (X, Y^{32{Sub}}, Sub, S, Cout);
endmodule

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

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
```

4.12 REG_ifid

4.12.1 所处位置

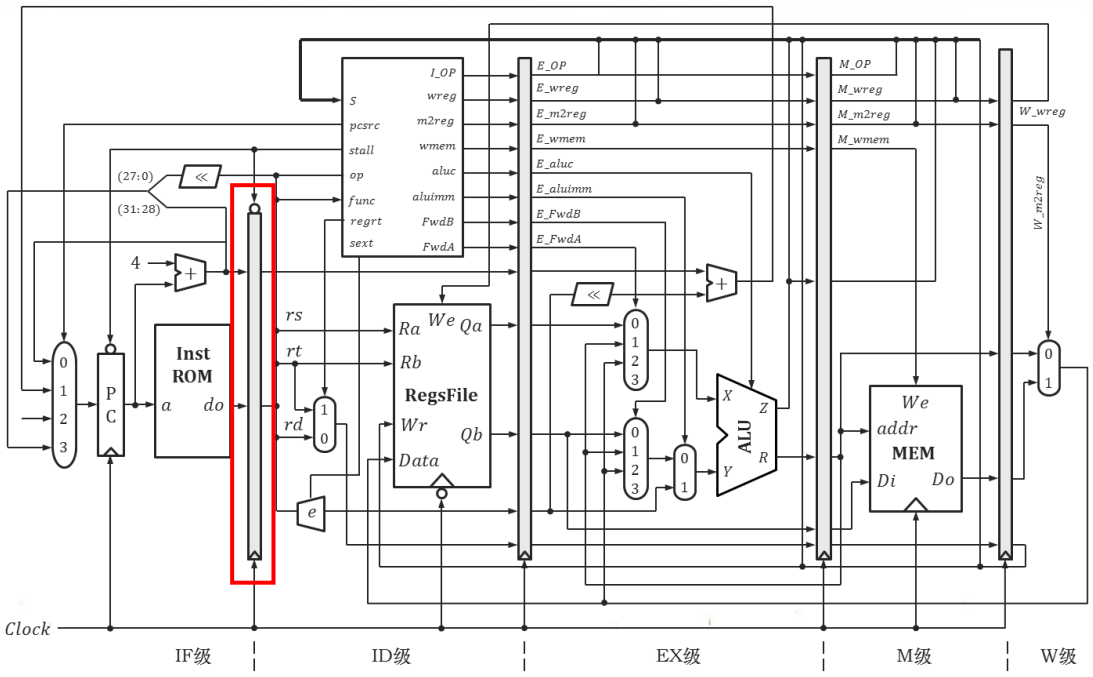

图 4-12 REG_ifid

4.12.2 模块功能

IF/ID 流水线寄存器会从程序计数器（PC）中读取指令，并将这些指令放入流水线寄存器中，在下一个时钟周期输出指令信息。

4.12.3 实现思路

将 IF 级信息接口作为输入与输出信号，根据 stall 与 condep 控制信息与时钟信号进行写入写停。

4.12.4 引脚及控制信号名

CLk：时钟周期，外部输入信号。

En：清零信号，外部输入信号。

Clrn：重置寄存器堆。

stall,condep：输入信息，指令控制 En，Clrn 高低电平

IF_PCadd4，IF_Inst（D0，D1）：输入信号。

ID_PCadd4，ID_Inst（Q0，Q1）：输出信号。

4.12.5 主要实现代码

```
module REG_ifid(D0,D1,Q0,Q1,En,Clk,Clrn,stall,condep);
input [31:0]D0,D1;
input En,Clk,Clrn;
input stall,condep;
```

```
output [31:0]Q0,Q1;
wire En_S,Clrn_C;
wire [31:0]Q0n,Q1n;
assign En_S=En&~stall;
assign Clrn_C=Clrn&~condep;
D_FFEC32 q0(D0,Clk,En_S,Clrn_C,Q0,Q0n);
D_FFEC32 q1(D1,Clk,En_S,Clrn_C,Q1,Q1n);
Endmodule
```

4.13 REG_idex

4.13.1 所处位置

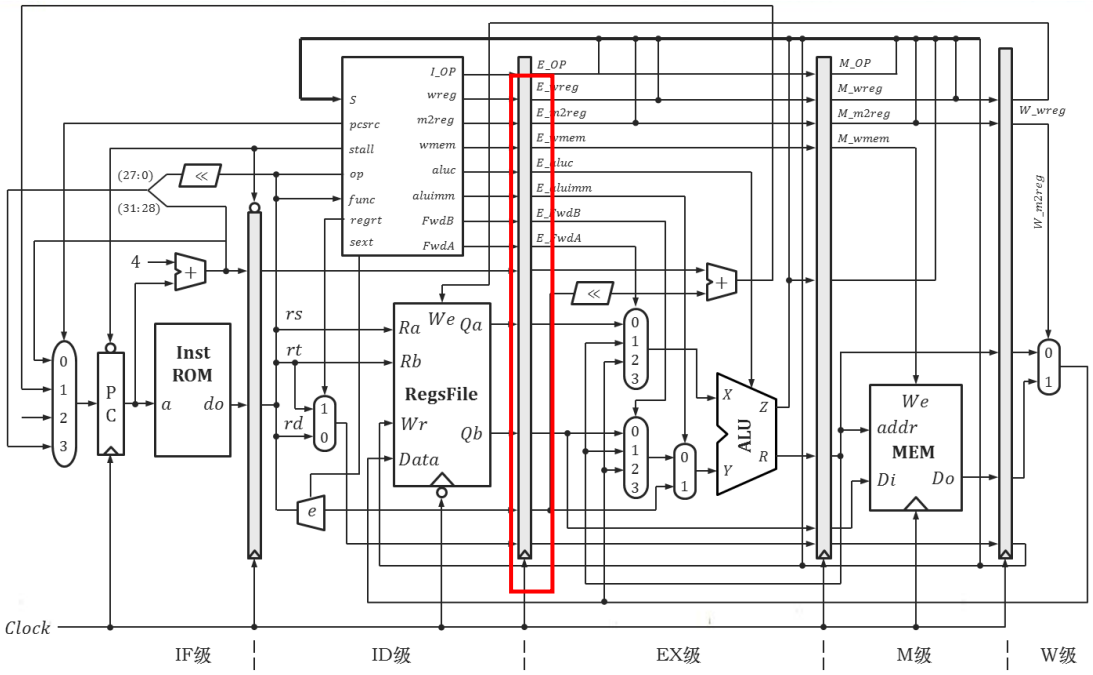

图 4-13 REG_ idex

4.13.2 模块功能

ID/EX 流水线寄存器会从 CONUNIT，RegFile 中读取指令，并将这些指令放入流水线寄存器中，在下一个时钟周期输出指令信息。

4.13.3 实现思路

将 ID 级信息接口作为输入与输出信号，根据 stall 与 condep 控制信息与时钟信号进行写入写停。

4.13.4 引脚及控制信号名

CLk:时钟周期，外部输入信号。

En：清零信号，外部输入信号。

Clrn：重置寄存器堆。

stall,condep：输入信息，指令控制 En，Clrn 高低电平

FwdA,FwdB：输入信息，用于 EX 级信号选择

Aluc：4 位控制信号

ID_Wr：输入信号，根据指令类型为 rt 或 rd。

其余信号均仅做简单寄存，下个周期输出。

4.13.5 主要实现代码

```
ModuleREG_idex(Wreg,Reg2reg,Wmem,Op,Aluc,Aluqb,ID_PCadd4,ID_Inst,ID_Inst_2,j,ID
_InstL2,ID_PC,ID_Qa,ID_Qb,ID_EXTIMM,ID_Wr,En,Clk,Clrn,E_Wreg,E_Reg2reg,E_Wmem,E
_Op,E_Aluc,E_Aluqb,EX_Inst,E_R1,E_R2,E_I,E_Rd,FwdA,FwdB,E_FwdA,E_FwdB,EX_PC,EX_
InstL2,EX_PCadd4,E_j,E_Func,stall,condep);
input [31:0]ID_EXTIMM,ID_PC,ID_PCadd4,ID_InstL2,ID_Inst,ID_Qa,ID_Qb;
input [5:0]Op,ID_Inst_2;
input [4:0]ID_Wr;
input [3:0]Aluc;
input [1:0]FwdA,FwdB;
input Wreg,Reg2reg,Wmem,Aluqb,j;
input En,Clk,Clrn,condep,stall;
output [31:0] EX_PCadd4,EX_InstL2,EX_Inst,E_R1,E_R2,E_I,EX_PC;
output [5:0] E_Op,E_Func;
output [4:0]E_Rd;
output [3:0]E_Aluc;
output [1:0]E_FwdA,E_FwdB;
output E_Wreg,E_Reg2reg,E_Wmem,E_Aluqb,E_j;
wire [31:0] Qn15,Qn6,Qn7,Qn8,Qn9,Qn13,Qn14;
wire [5:0] Qn3,Qn17;
wire [4:0]Qn10;
wire [3:0]Qn4;
wire [1:0]Qn11,Qn12;
wire Qn0,Qn1,Qn2,Qn5,Qn16;
wire Clrn_SC;
assign Clrn_SC=Clrn&~stall&~condep;
D_FFEC q0(Wreg,Clk,En,Clrn_SC,E_Wreg,Qn0);
D_FFEC q1(Reg2reg,Clk,En,Clrn_SC,E_Reg2reg,Qn1);
D_FFEC q2(Wmem,Clk,En,Clrn_SC,E_Wmem,Qn2);
```

```
D_FFEC q5(Aluqb, Clk, En, Clrn_SC, E_Aluqb, Qn5);
D_FFEC q16(j, Clk, En, Clrn_SC, E_j, Qn16);
D_FFEC2 q11(FwdA, Clk, En, Clrn_SC, E_FwdA, Qn11);
D_FFEC2 q12(FwdB, Clk, En, Clrn_SC, E_FwdB, Qn12);
D_FFEC4 q4(Aluc, Clk, En, Clrn_SC, E_Aluc, Qn4);
D_FFEC5 q10(ID_Wr, Clk, En, Clrn_SC, E_Rd, Qn10);
D_FFEC6 q3(Op, Clk, En, Clrn_SC, E_Op, Qn3);
D_FFEC32 q6(ID_Inst, Clk, En, Clrn_SC, EX_Inst, Qn6);
D_FFEC32 q7(ID_Qa, Clk, En, Clrn_SC, E_R1, Qn7);
D_FFEC32 q8(ID_Qb, Clk, En, Clrn_SC, E_R2, Qn8);
D_FFEC32 q9(ID_EXTIMM, Clk, En, Clrn_SC, E_I, Qn9);
D_FFEC32 q13(ID_PC, Clk, En, Clrn_SC, EX_PC, Qn13);
D_FFEC32 q14(ID_InstL2, Clk, En, Clrn_SC, EX_InstL2, Qn14);
D_FFEC32 q15(ID_PCadd4, Clk, En, Clrn_SC, EX_PCadd4, Qn15);
D_FFEC6 q17(ID_Inst_2, Clk, En, Clrn_SC, E_Func, Qn17);
Endmodule
```

4.14 REG_exmem

4.14.1 所处位置

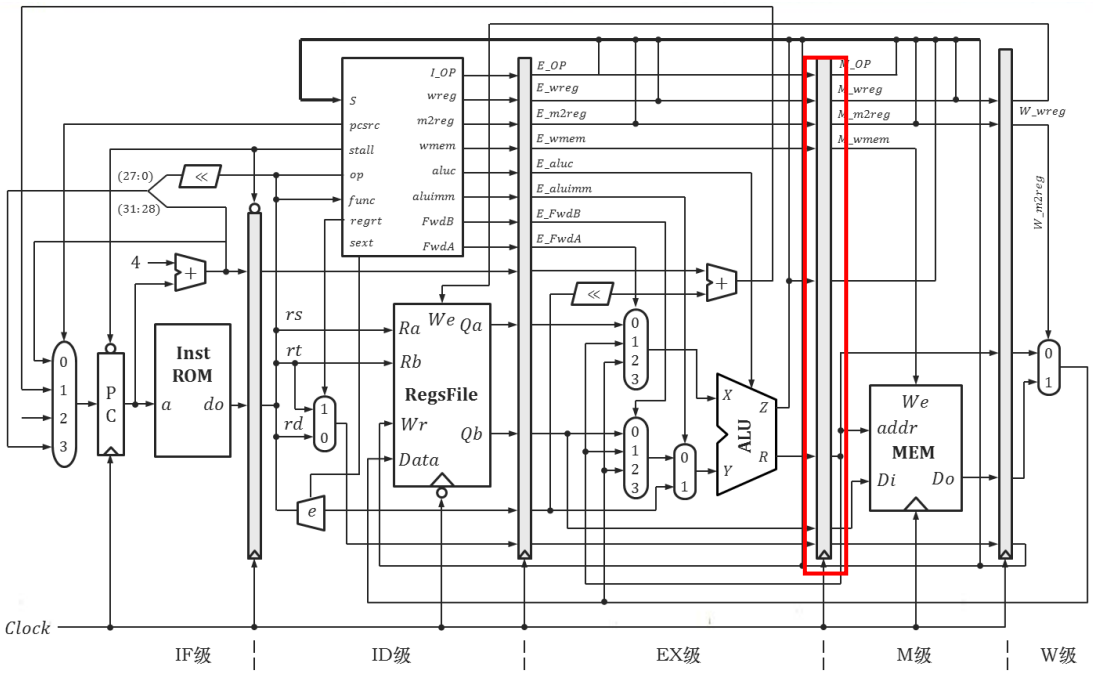

图 4-14 REG_ exmem

4.14.2 模块功能

EX/MEM 流水线寄存器会从 ALU，ID/EX 流水线寄存器中读取指令，并将这些指令放入流水线寄存器中，在下一个时钟周期输出指令信息。

4.14.3 实现思路

将 EX 级信息接口作为输入与输出信号

4.14.4 引脚及控制信号名

CLk:时钟周期，外部输入信号。

En：清零信号，外部输入信号。

Clrn：重置寄存器堆。

EX_PCadd4,E_R,E_R2, E_Wreg,E_Reg2reg,E_Wmem,E_j, E_Rd：输入信号

M_PCadd4,M_R,M_S, M_Rd,M_Wreg,M_Reg2reg,M_Wmem,M_j：输出信号

4.14.5 主要实现代码

```
ModuleREG_exmem(E_Wreg,E_Reg2reg,E_Wmem,E_R,E_R2,E_Rd,E_j,EX_PCadd4,En,Clk,Clrn
,M_Wreg,M_Reg2reg,M_Wmem,M_R,M_S,M_Rd,M_PCadd4,M_j);
input [31:0]EX_PCadd4,E_R,E_R2;
input [4:0]E_Rd;
input E_Wreg,E_Reg2reg,E_Wmem,E_j;
input En,Clk,Clrn;
output [31:0] M_PCadd4,M_R,M_S;
output [4:0]M_Rd;
output M_Wreg,M_Reg2reg,M_Wmem,M_j;
wire [31:0] Qn3,Qn6,Qn7;
wire [4:0]Qn8;
wire Qn0,Qn1,Qn2,Qn4;
D_FFEC q0(E_Wreg,Clk,En,Clrn,M_Wreg,Qn0);
D_FFEC q1(E_Reg2reg,Clk,En,Clrn,M_Reg2reg,Qn1);
D_FFEC q2(E_Wmem,Clk,En,Clrn,M_Wmem,Qn2);
D_FFEC32 q3(EX_PCadd4,Clk,En,Clrn,M_PCadd4,Qn3);
D_FFEC q4(E_j,Clk,En,Clrn,M_j,Qn4);
D_FFEC32 q6(E_R,Clk,En,Clrn,M_R,Qn6);
D_FFEC32 q7(E_R2,Clk,En,Clrn,M_S,Qn7);
D_FFEC5 q8(E_Rd,Clk,En,Clrn,M_Rd,Qn8);
Endmodule
```

4.15 REG_memwb

4.15.1 所处位置

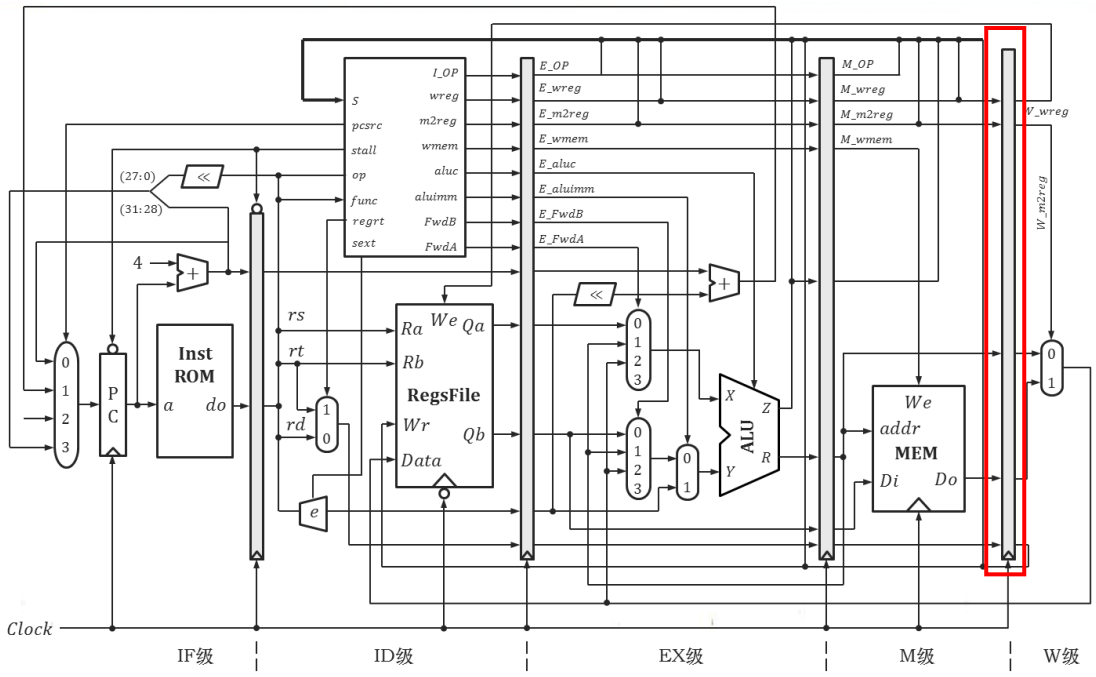

图 4-15 REG_ memwb

4.15.2 模块功能

MEM/WB 流水线寄存器会从 DATAMEM，EX/MEM 流水线寄存器中读取指令，并将这些指令放入流水线寄存器中，在下一个时钟周期输出指令信息。

4.15.3 实现思路

将 MEM 级信息接口作为输入与输出信号

4.15.4 引脚及控制信号名

CLk:时钟周期，外部输入信号。

En：清零信号，外部输入信号。

Clrn：重置寄存器堆。

M_Wreg,M_Reg2reg,M_j,M_R,Dout,M_PCadd4,M_Rd：输入信号

W_Wreg,W_Reg2reg,W_j,W_Dout,W_C,W_PCadd4, W_Wr：输出信号

4.15.5 主要实现代码

```
moduleREG_memwb(M_j, M_PCadd4, M_Wreg, M_Reg2reg, M_R, Dout, M_Rd, En, Clk, Clrn, W_Wreg,
W_Reg2reg, W_Dout, W_C, W_Wr, W_PCadd4, W_j);
input M_Wreg, M_Reg2reg, M_j;
```

```
input [31:0] M_R,Dout,M_PCadd4;
input [4:0] M_Rd;
input En,Clk,Clrn;
output W_Wreg,W_Reg2reg,W_j;
output [31:0] W_Dout,W_C,W_PCadd4;
output [4:0] W_Wr;
wire Qn0,Qn1,Qn6;
wire [31:0] Qn2,Qn3,Qn5;
wire [4:0] Qn4;
D_FFEC q0(M_Wreg,Clk,En,Clrn,W_Wreg,Qn0);
D_FFEC q1(M_Reg2reg,Clk,En,Clrn,W_Reg2reg,Qn1);
D_FFEC32 q2(M_R,Clk,En,Clrn,W_Dout,Qn2);
D_FFEC32 q3(Dout,Clk,En,Clrn,W_C,Qn3);
D_FFEC5 q4(M_Rd,Clk,En,Clrn,W_Wr,Qn4);
D_FFEC32 q5(M_PCadd4,Clk,En,Clrn,W_PCadd4,Qn5);
D_FFEC q6(M_j,Clk,En,Clrn,W_j,Qn6);
Endmodule
```

4.16 CPU

4.16.1 所处位置

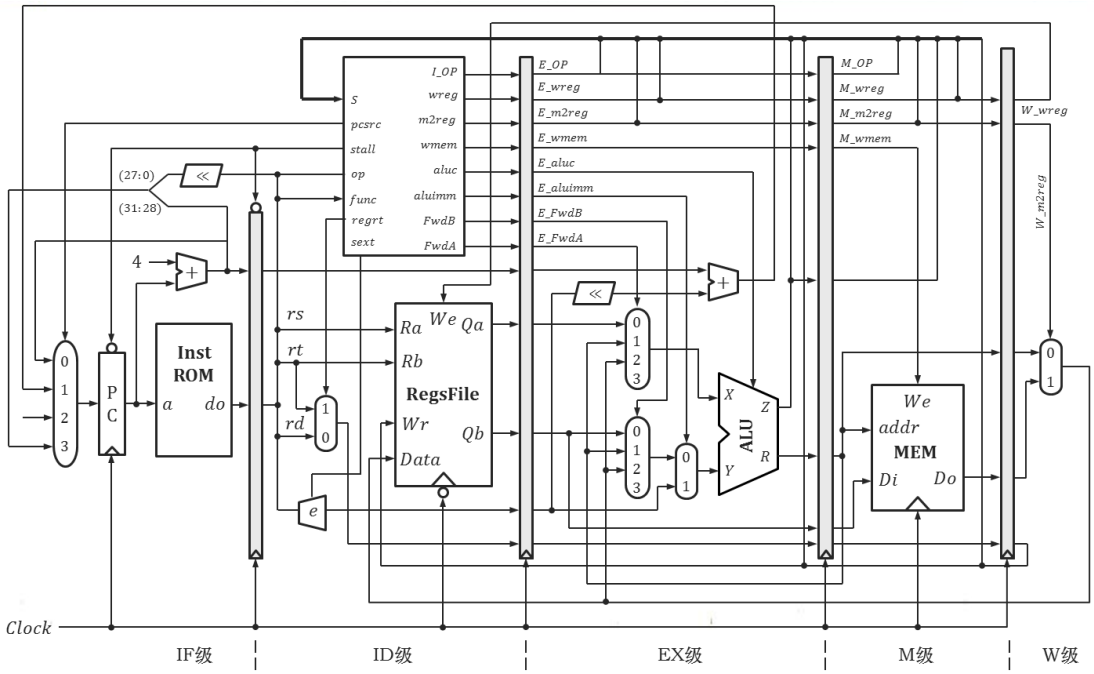

图 4-16 CPU

4.16.2 模块功能

实现整个流水线 CPU 的封装。

4.16.3 实现思路

调用各个部分模块并将他们的输入和输出连接到一起。

4.16.4 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。

Clrn：重置寄存器堆。

IF_ADDR,EX_R,EX_X,EX_Y：输出信号

4.16.5 主要实现代码

```
module CPU(Clk,En,Clrn,IF_ADDR,EX_X,EX_Y,EX_R);
input Clk,En,Clrn;
output[31:0] IF_ADDR,EX_R,EX_X,EX_Y;
wire[31:0]IF_Result,IF_Addr,IF_PCadd4,IF_Inst,Dread,D1,ID_Qa,ID_Qb,ID_PCadd4,ID
_Inst,ID_InstL2,EX_InstL2;
wire[31:0]E_R1,E_R2,E_I,X,Y,E_R,EX_PC,EX_Inst,M_R,M_S,Dout,W_Dout,W_C,ID_EXTIMM
,Alu_X,E_NUM,ID_EXTIMM_L2,ID_PC,EX_PCadd4,M_PCadd4,W_PCadd4;
wire [5:0] E_Op,E_Func;
```

```
wire [4:0] ID_Wr,ID_Wr_1,W_Wr,E_Rd,M_Rd;
wire [3:0]Aluc,E_Aluc;
wire [1:0]Pcsrc,FwdA,FwdB,E_FwdA,E_FwdB;
wire Regrt,Se,Wreg,Aluqb,Reg2reg,Wmem,Z,shift,j,Clkn,E_j,M_j,W_j;
wire
E_Wreg,E_Reg2reg,E_Wmem,E_Aluqb,Cout,M_Wreg,M_Reg2reg,M_Wmem,W_Wreg,W_Reg2reg,s
tall,condep;
not i0(Clkn,Clk);
MUX4X32 mux4x32(IF_PCadd4,EX_PC,E_R1,EX_InstL2,Pcsrc,IF_Result);
PC pc(IF_Result,Clk,En,Clrn,IF_Addr,stall);
PCadd4 pcadd4(IF_Addr,IF_PCadd4);
INSTMEM instmem(IF_Addr,IF_Inst);
REG_ifid
reg_ifid(IF_PCadd4,IF_Inst,ID_PCadd4,ID_Inst,En,Clk,Clrn,stall,condep);
CONUNIT
conunit(Z,E_Op,ID_Inst[31:26],E_Func,ID_Inst[5:0],ID_Inst[25:21],ID_Inst[20:16]
,Regrt,Se,E_Rd,M_Rd,E_Reg2reg,E_Wreg,M_Wreg,Wreg,Reg2reg,Pcsrc,Wmem,Aluc,Aluqb,
shift,j,FwdA,FwdB,stall,condep);
assign ID_Wr_1=Regrt?ID_Inst[20:16]:ID_Inst[15:11];
assign ID_Wr=j?31:ID_Wr_1;
EXT16T32 ext16t32(ID_Inst[15:0],Se,ID_EXTIMM);
REGFILE
regfile(ID_Inst[25:21],ID_Inst[20:16],Dread,W_Wr,W_Wreg,Clkn,Clrn,ID_Qa,ID_Qb);
SHIFTER32_L2 shifter2(ID_EXTIMM,ID_EXTIMM_L2);
SHIFTER26_L2 shifter1(ID_Inst[25:0],ID_PCadd4[31:28],ID_InstL2);
CLA_32 cla_32(ID_PCadd4,ID_EXTIMM_L2,0,ID_PC,Cout);
MUX2X32 mux2x32_1(D1,W_PCadd4,W_j,Dread);
REG_idex
reg_idex(Wreg,Reg2reg,Wmem,ID_Inst[31:26],Aluc,Aluqb,ID_PCadd4,ID_Inst,ID_Inst[
5:0],j,ID_InstL2,ID_PC,ID_Qa,ID_Qb,ID_EXTIMM,ID_Wr,En,Clk,Clrn,E_Wreg,E_Reg2reg
,E_Wmem,E_Op,E_Aluc,E_Aluqb,EX_Inst,E_R1,E_R2,E_I,E_Rd,FwdA,FwdB,E_FwdA,E_FwdB,
EX_PC,EX_InstL2,EX_PCadd4,E_j,E_Func,stall,condep);
MUX4X32 mux4x32_ex_1(E_R1,D1,M_R,0,E_FwdA,Alu_X);
MUX4X32 mux4x32_ex_2(E_R2,D1,M_R,0,E_FwdB,E_NUM);
MUX2X32 mux2x32_2(E_I,E_NUM,E_Aluqb,Y);
MUX2X32 mux2x32_3(Alu_X,EX_Inst,shift,X);
ALU alu(X,Y,E_Aluc,E_R,Z);
REG_exmem
```

```
reg_exmem(E_Wreg,E_Reg2reg,E_Wmem,E_R,E_R2,E_Rd,E_j,EX_PCadd4,En,Clk,Clrn,M_Wre
g,M_Reg2reg,M_Wmem,M_R,M_S,M_Rd,M_PCadd4,M_j);
DATAMEM datamem(M_R,M_S,Clk,M_Wmem,Dout);
REG_memwb
reg_memwb(M_j,M_PCadd4,M_Wreg,M_Reg2reg,M_R,Dout,M_Rd,En,Clk,Clrn,W_Wreg,W_Reg2
reg,W_Dout,W_C,W_Wr,W_PCadd4,W_j);
MUX2X32 mux2x32_4(W_C,W_Dout,W_Reg2reg,D1);
assign IF_ADDR=IF_Addr;
assign EX_R=E_R;
assign EX_X=X;
assign EX_Y=Y;
endmodule
```

4.17 CPU_END_test

4.17.1 模块功能

堆整个封装完毕的流水线 cpu 进行测试。

4.17.2 主要实现代码

```
`timescale 10ns/10ns
module CPU_test;
reg Clk,En,Clrn;
wire[31:0] IF_ADDR,EX_R,EX_X,EX_Y;
initial begin
Clk=0;
Clrn=0;
En=1;
#10
Clrn<=1;
end
always #10 Clk=~Clk;
CPU CPU_test(
.Clk(Clk),
.En(En),
```

```
.Clrn(Clrn),
.IF_ADDR(IF_ADDR),
.EX_R(EX_R),
.EX_X(EX_X),
.EX_Y(EX_Y));
endmodule
```

五、仿真模拟分析

5.1 仿真波形图

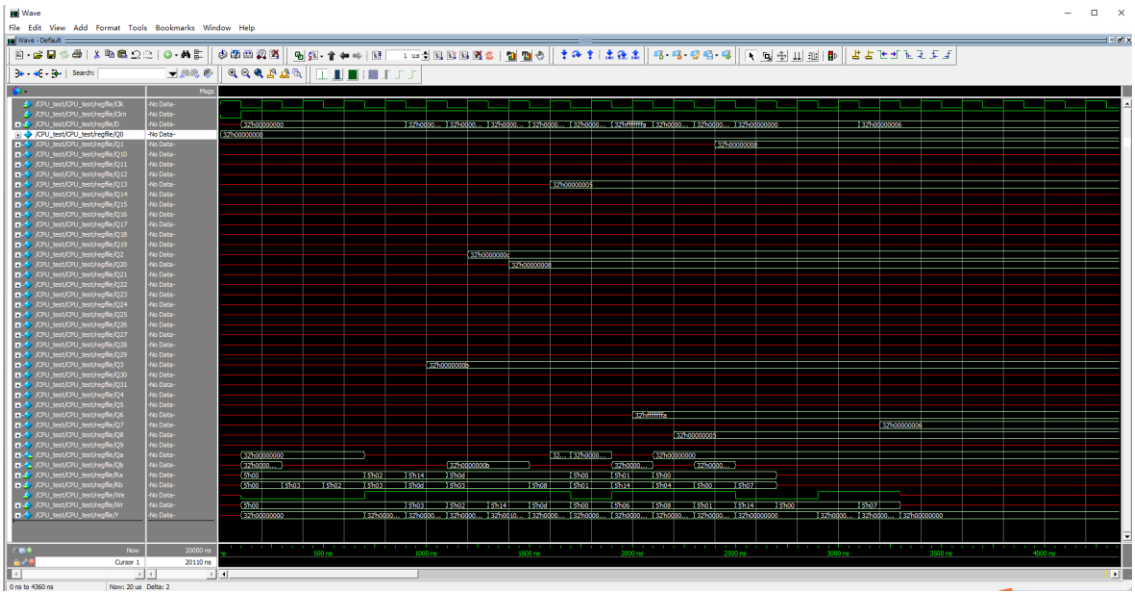

图 5-1 寄存器波形图

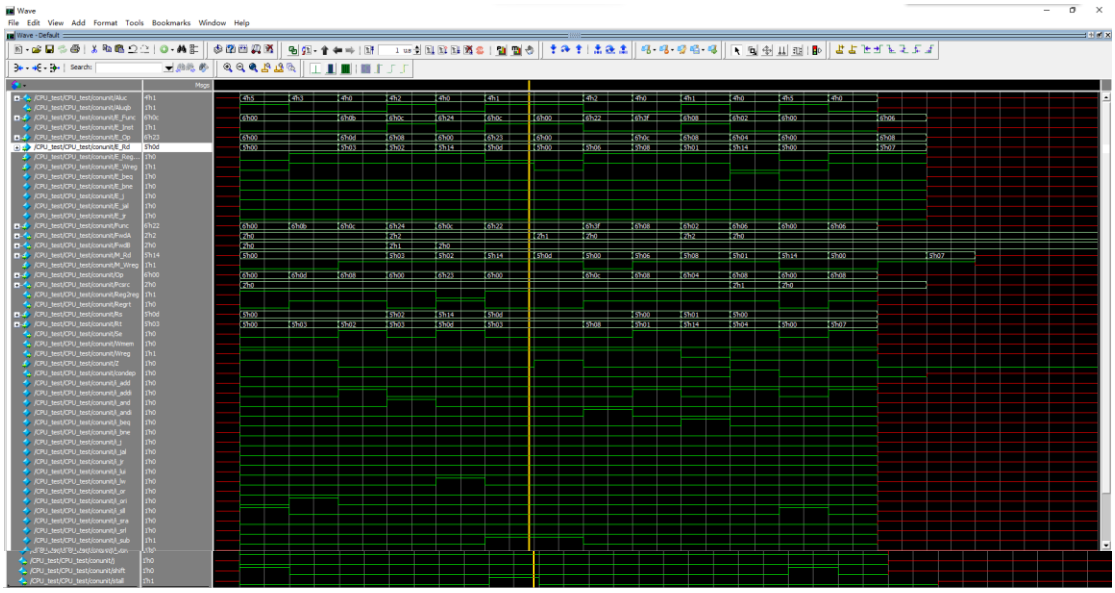

图 5-2 控制信号波形图

5.2 指令代码分析

5.2.1

assign Rom[5'h00]=32'b001101_00000_00011_0000_0000_0000_1011;指令含义：ori $3,$0,11

结果：$3=11

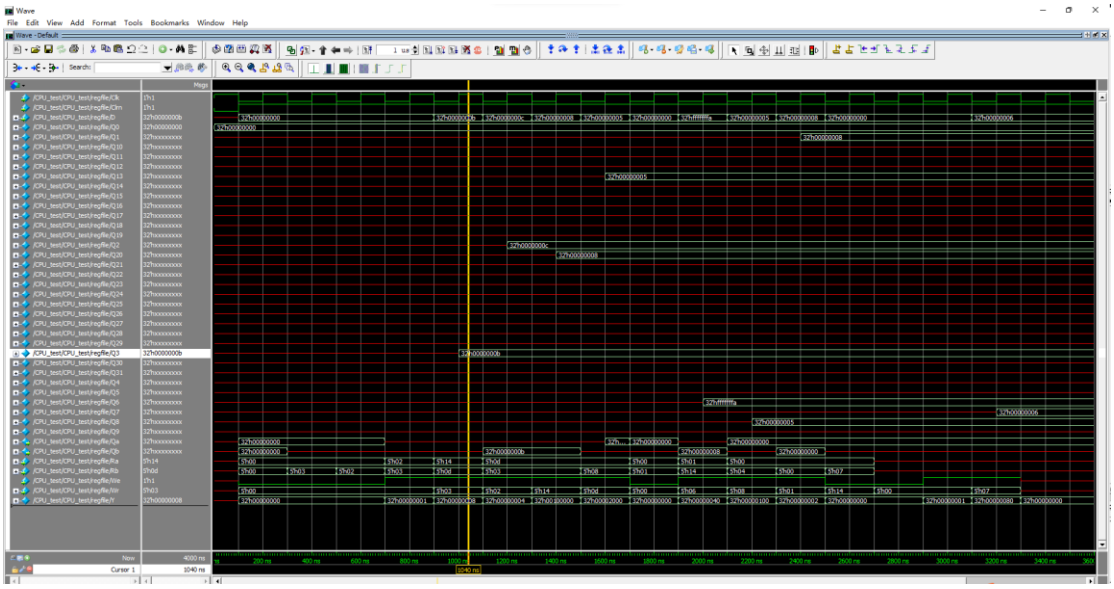

图 5-3 ori 寄存器结果

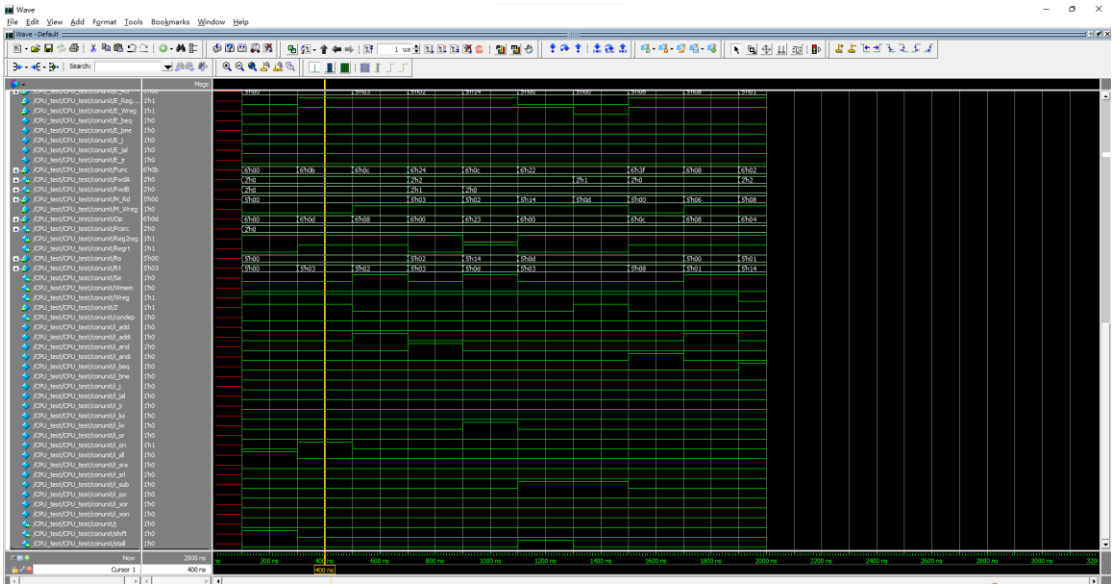

图 5-4 ori 控制信号

Ori 控制指令相比$3 赋值为 11 写入寄存器提前 3 个周期，说明指令运行到时钟周期 WB 段与原先 CONUNIT 所在的 ID 级相差 3 个时钟周期。

5.2.2

assign Rom[5'h01]=32'b001000_00000_00010_0000_0000_0000_1100;

指令含义：addi $2,$0,12

结果：$2=12

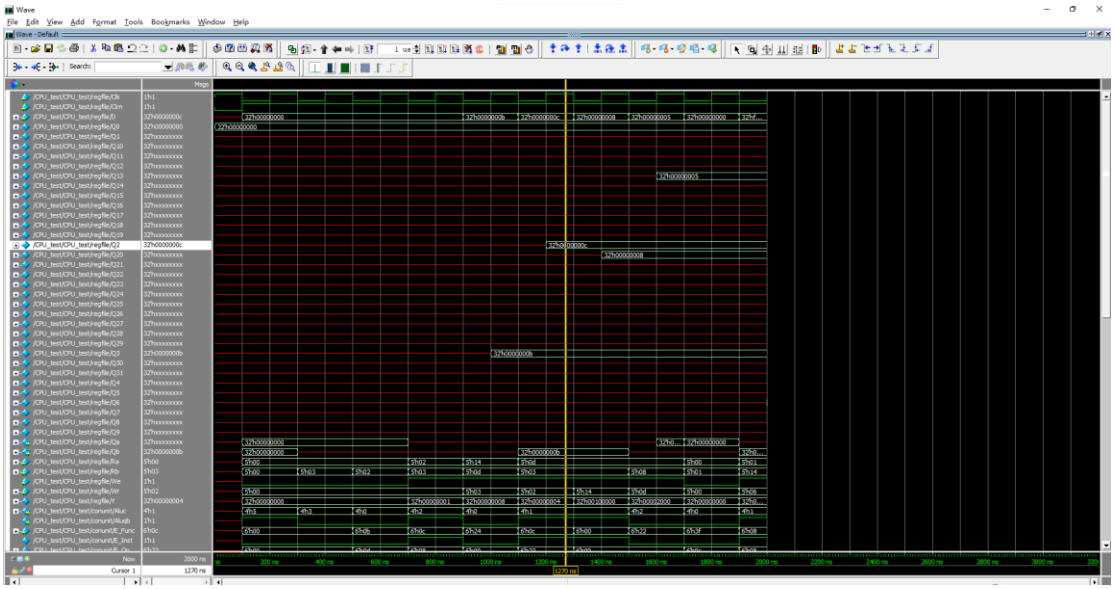

图 5-5 addi 寄存器结果

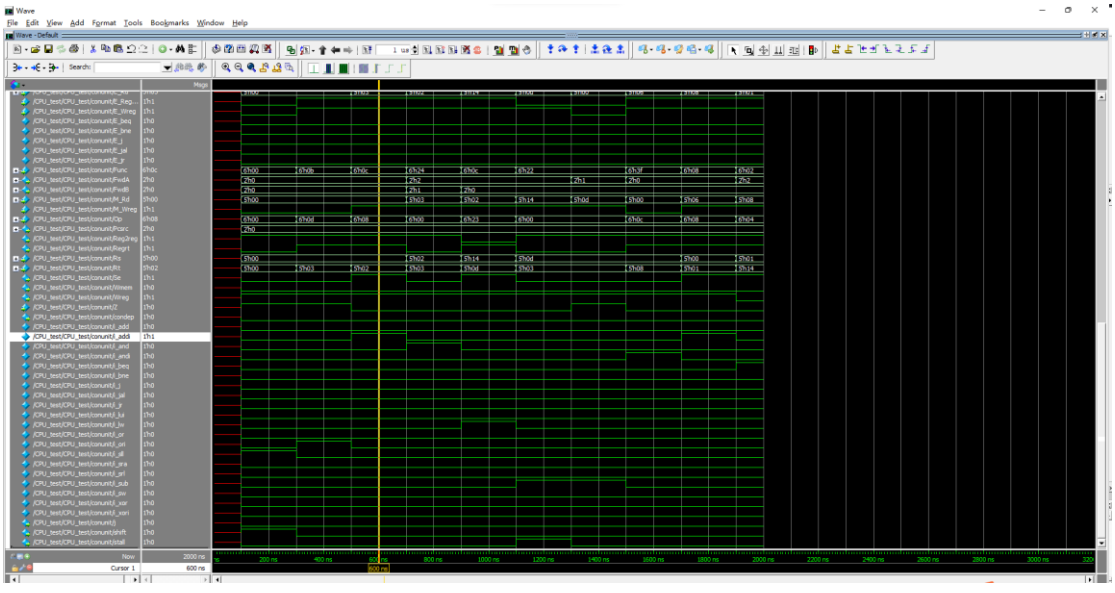

图 5-6 addi 控制信号

5.2.3

assign Rom[5'h02]=32'b000000_00010_00011_10100_00000_100100;

指令含义：and $2,$3,$20

结果：$20=8

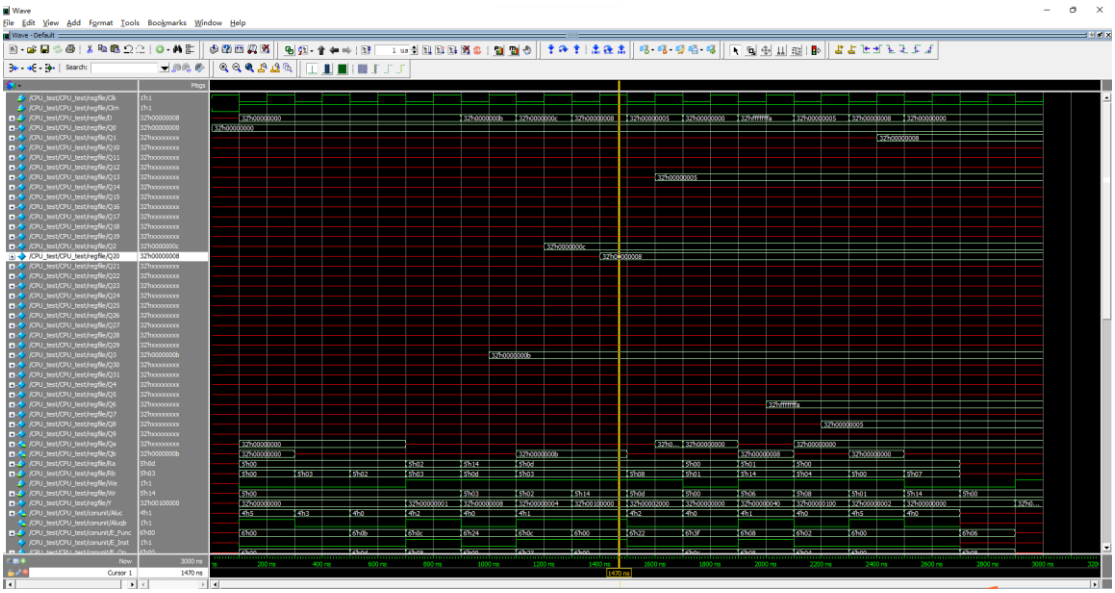

图 5-7 and 寄存器结果

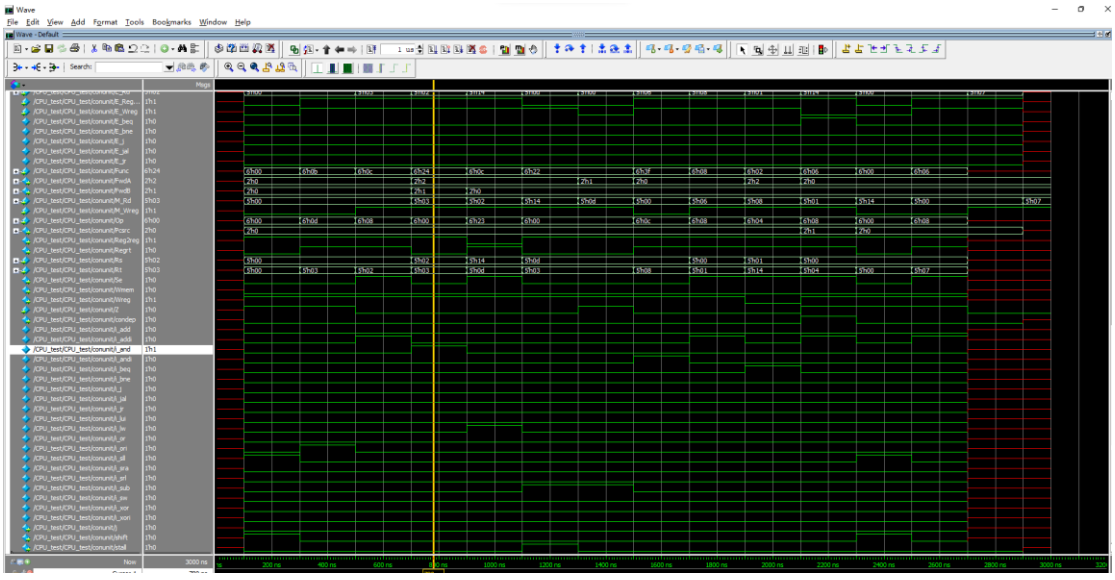

图 5-8 and 控制信号

$2 的调用本应该造成数据冲突，由于 MEM 级和 EX 级间、WB 级和 EX 级之间增加数据路径，使得运算结果能前推到 EX 级作为 AIU 的输入作为运算，保证第二条指令和第三条指令的计算结果是正确的，实现内部前推。

5.2.4

assign Rom[5'h03]= 32'b100011_10100_01101_0000000000001100;

指令含义：lw $13 12($20)

结果：$13=5

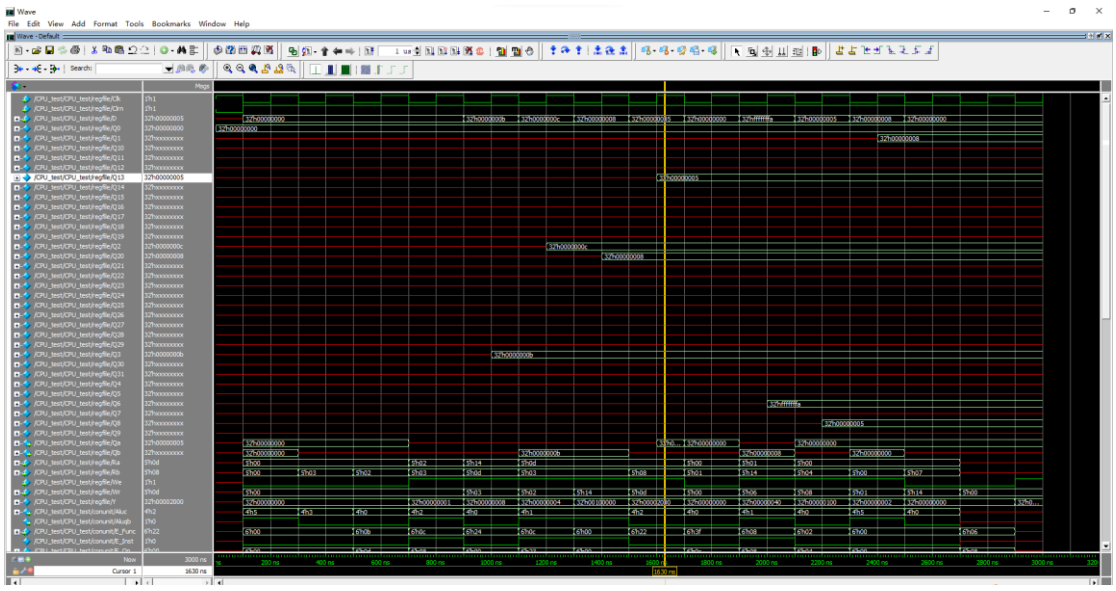

图 5-9 lw 寄存器结果

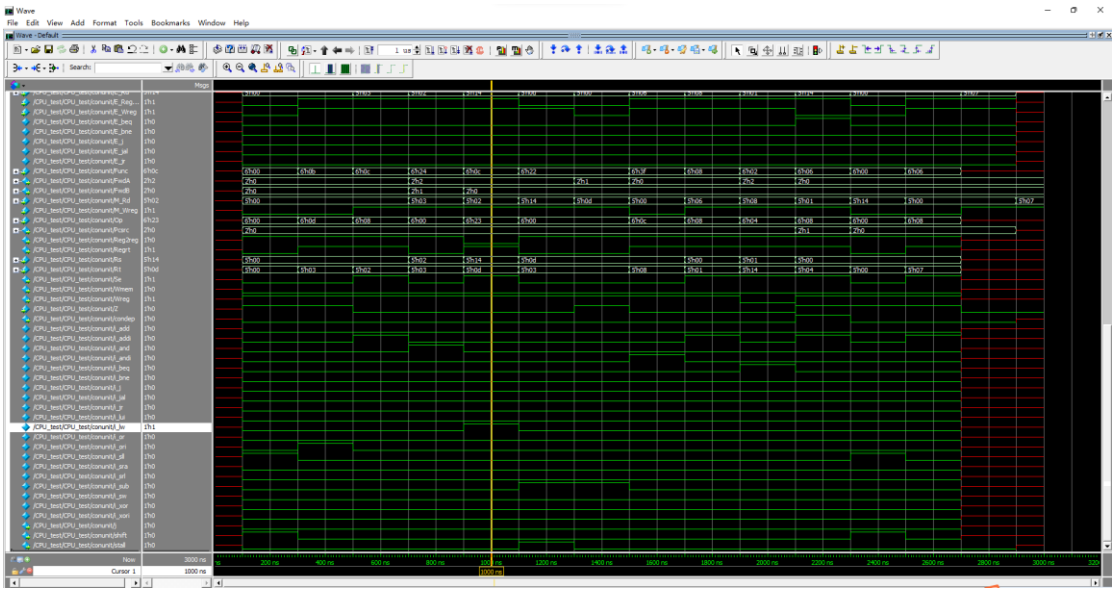

图 5-10 lw 控制信号

5.2.5

```
assign Rom[5'h04]= 32'b000000_01101_00011_00110_00000_100010;
```

指令含义：sub $13,$3,$6

结果：$6=fffffa

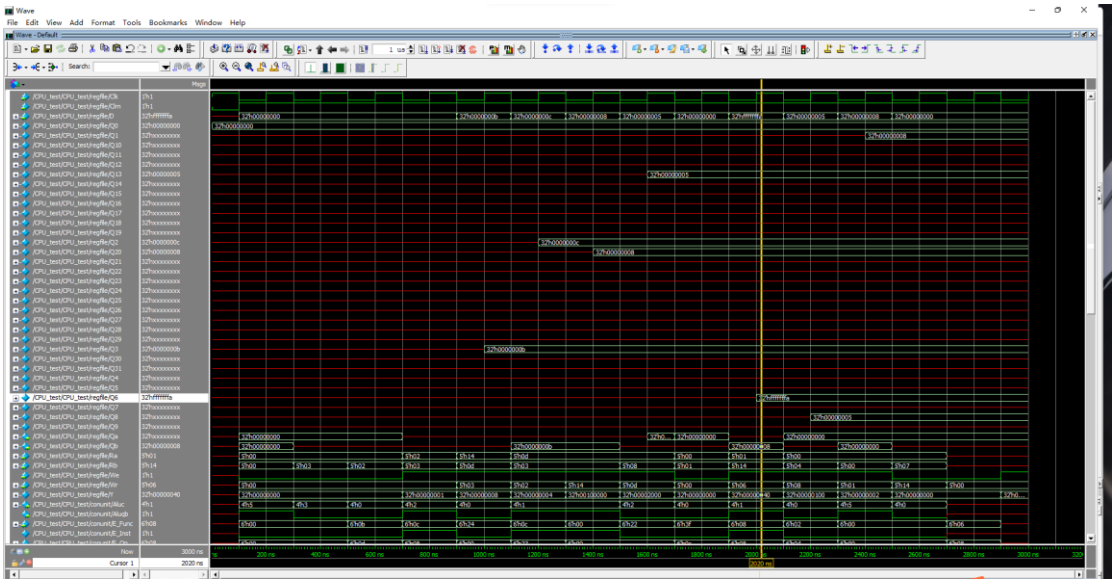

图 5-11 sub 寄存器结果

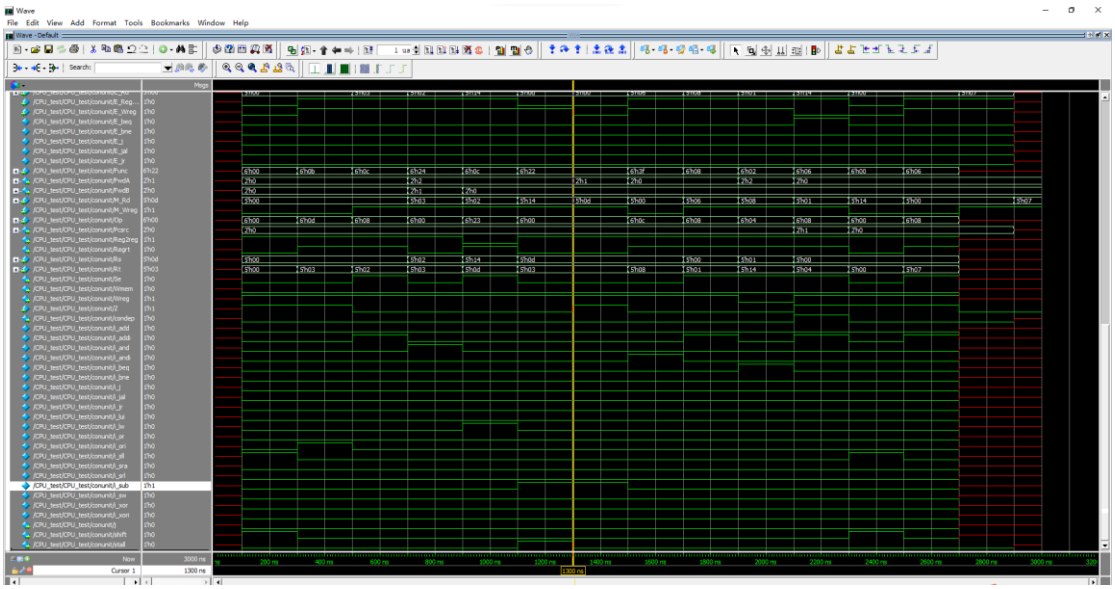

图 5-12 sub 控制信号

lw 指令的数据冒险：通过 Regreg 信号可以区分 lw 指令和其他指令

暂停流水线：PC 寄存器的输出维持不变，IF 仍为 sub 指令；将 ID/EX 流水线寄存器的 Clrn 端口信号清 0，使 EX 信号变为空指令。

由波形图可知，sub 指令延迟一个时钟周期

5.2.6

```
assign Rom[5'h05]=32'b001100_01101_01000_1111_1111_1111_1111;
```

指令含义：andi $8,$2,65535

结果：$8=5

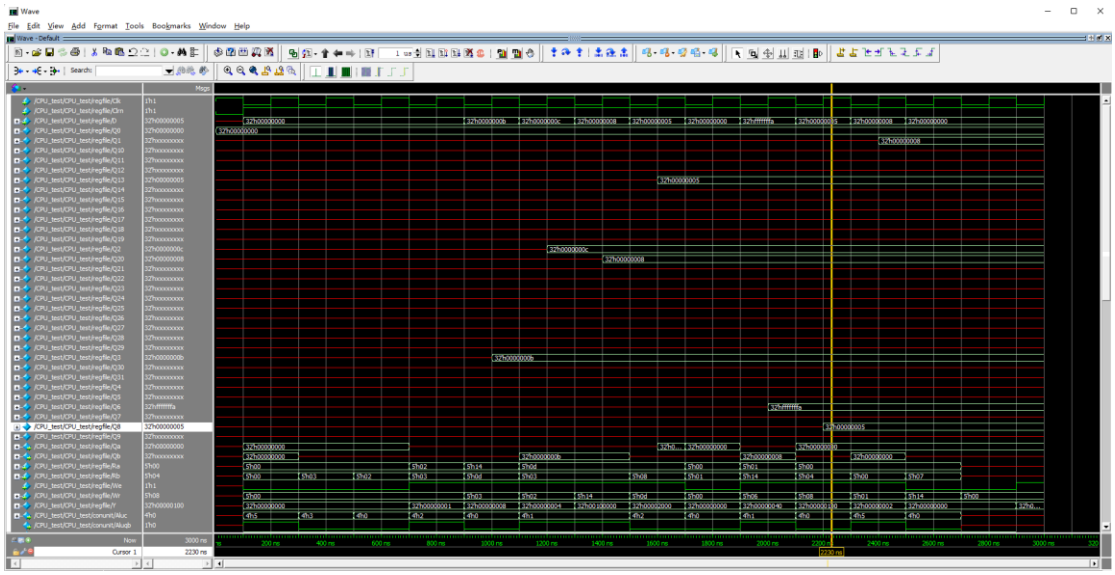

图 5-13 andi 寄存器结果

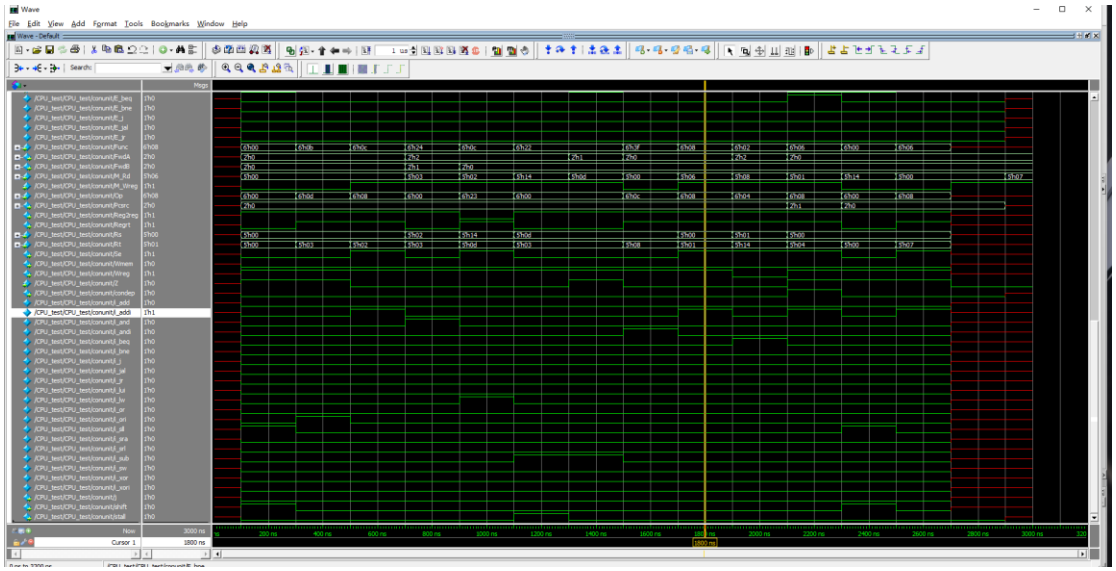

图 5-14 andi 控制信号

5.2.7

```
assign Rom[5'h06]= 32'b000100_00001_10100_0000000000000010;
```

指令含义：beq $1 $7 2

结果：跳转到下两条指令

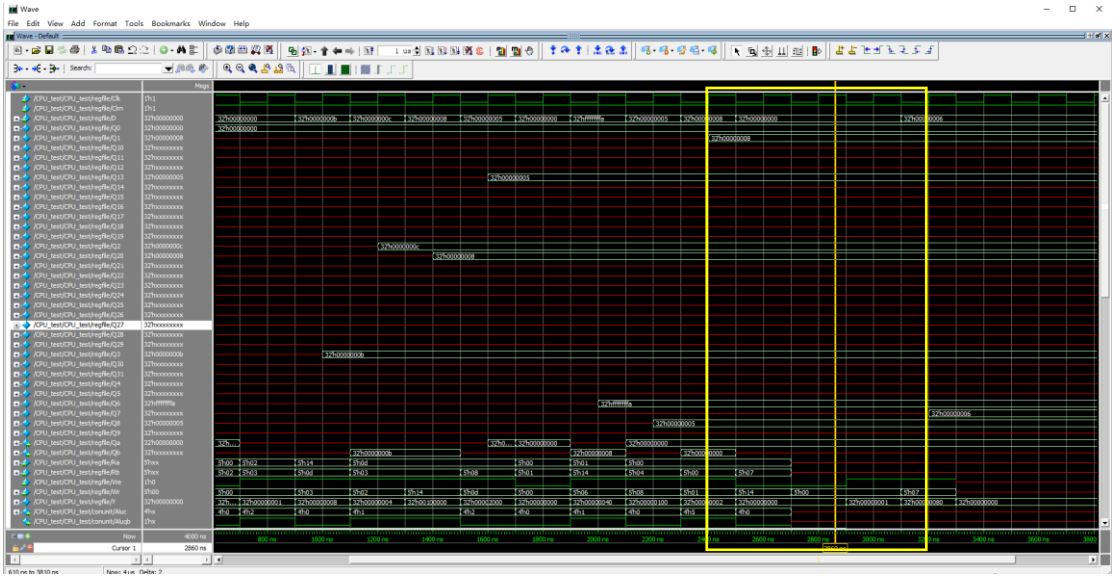

图 5-15 beq 寄存器结果

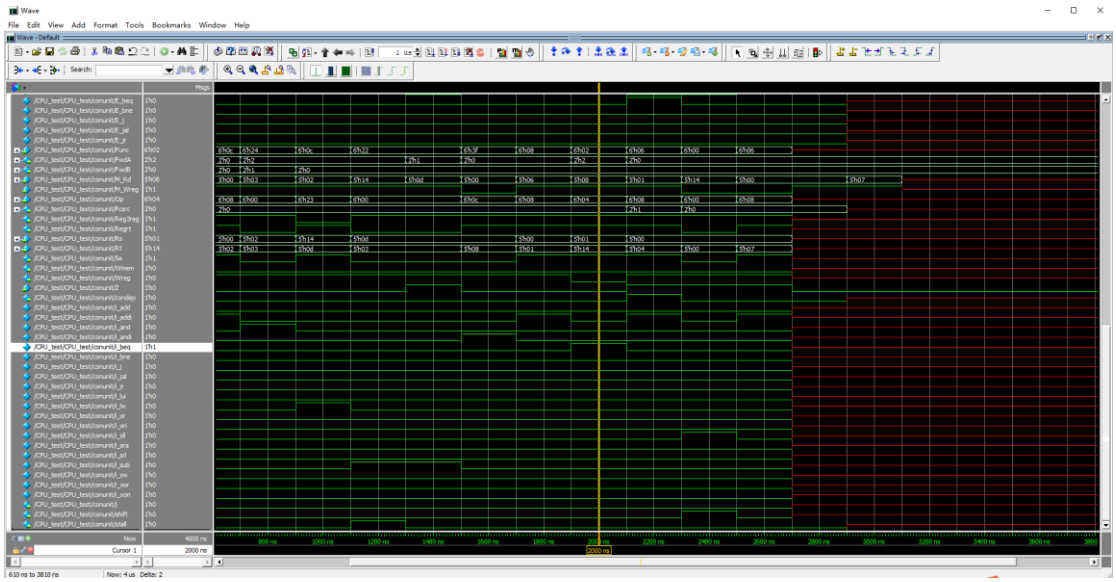

图 5-16 beq 控制信号

根据计算 beq，执行

assignRom[5'h0A]=32'b001000_00000_00111_0000_0000_0000_0110;//addi $7,$0,6 $7=6

由黄色框表示其中其中相差 4 个时钟周期，以上条指令为界限，后时钟周期包括 ID 识别 beq 指令阶段一个时钟周期，阻塞两个时钟周期，指令 addi 到达 WB 阶段相差一个时钟周期。

实现控制冒险的解决策略功能。

六、结论和体会

6.1 体会感悟

通过此次基于 MIPS 指令集的 32 位流水线 CPU 设计实验，让我们对流水线 CPU 内部组成以及各种冒险如何运作解决有了一个更深的理解。在此次实验中，从最开始一步一步写好各个模块并分别进行测试，再到最后的完整封装测试，经历了许多问题，包括流水线寄存器接口需要根据指令需求进行连接与编写，非常考验条理性。

特别实在新增指令的过程中，需要对 CONUNIT、ALU 等模块重写，还需要重新封装整个模块。

6.2 对本实验过程及方法、手段的改进建议

1.小模块写好就应该测试无误，防止封装到大模块后不知道哪里错误。

2.仿真测试过程中线路太多，要注意各个接口的命名。

3.有的模块可以通过合理使用一些逻辑运算符号来简化。