目录

实验二
支持 5 级流水线的处理器设计与实现

一、实验要求

（一）目的

将实验一中的单周期处理器拓展为 5 级流水线处理器，要求支持实验一中的单周期处理器的所有指令，并且支持数据前推、lw 数据冒险的流水线暂停，正确执行所有指令并验证波形。

（二）指令概述

下表为 MIPS 典型整数指令，其中未加粗部分为实验要求完成的基本指令，加粗部分为小组额外完成的指令。

表 1 MIPS 典型整数指令

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			寄存器加
sub	000010	rs	rt	rd			寄存器减
and	000100	rs	rt	rd			寄存器与
or	000101	rs	rt	rd			寄存器或
xor	010000	rs	rt	rd			寄存器异或
sll	010100	000000	rt	rd	sa		左移
srl	010101	000000	rt	rd	sa		逻辑右移
sra	010110	000000	rt	rd	sa		算术右移
jr	001011	rs	000000	000000	000000	000000	寄存器跳转
I 型指令							
addi	000001	rs	rt	immediate			立即数加
andi	001110	rs	rt	immediate			立即数与

ori	000011	rs	rt	immediate	立即数或
xori	010001	rs	rt	immediate	立即数异或
subi	001001	rs	rt	immediate	立即数减
lw	001000	rs	rt	offset	取数
sw	000111	rs	rt	offset	存数
beq	001001	rs	rt	offset	相等转移
bne	001111	rs	rt	offset	不等转移
lui	001111	00000	rt	immediate	设置高位
J 型指令					
j	001010	address			跳转
jal	001100	address			调用

（三）运行指令设计

下表为在该实验中我们设计的待运行指令，运行结果依赖指令存储器中指令。

表 2 指令集设计

地址	指令	含义
0	**add $s3,$s1,$s2**	$s3=$s1+$2
	32'b000000_00001_00010_00011_00000_100000;	
4	**sub $s3,$s1,$s2**	$s3=$s1-$s2
	32'b000000_00001_00010_00011_00000_100010;	
8	**and $s3,$s1,$s2**	$s3=$s1&$s2
	32'b000000_00001_00010_00011_00000_100100;	
12	**or $s3,$s1,$s2**	$s3=$s1|$s2
	32'b000000_00001_00010_00011_00000_100101;	
16	**addi $s3,$s1,0x1234**	$3=$1+0x1234
	32'b001000_00001_00011_0001001000110100;	
20	**andi $s3,$s1,0x00ef**	$3=$1&239
	32'b001100_00001_00011_0000000011101111;	
24	**ori $s3,$s1,0x00ef**	
	32'b001101_00001_00011_0000000011101111;	
28	**lw $s5,1($s3)**	$s5=memory[$s3+1]
	32'b100011_00011_00101_0000000000000001;	
32	**sw $s1,1($s3)**	memory[$s3+1]=$s1
	32'b101011_00011_00001_0000000000000001;	
36	**beq $s4,$s5,8**	if($s4==$s5) goto PC+4+32

	32'b000100_00100_00101_0000000000010000;	
40	**bne $s5,$s5,8**	if($s5!=$s5) goto PC+4+32
	32'b000101_00101_00101_0000000000010000;	
44	**j 0x0010c8**	$PC=(PC+4)_{31..28}(0x0010C8)_{27..2}00$
	32'b000010_0 0000 0001 0000 1100 1000_01100;	
48	**xor $s3,$s1,$s2**	$s3=$s1⊕$s2
	32'b000000_00001_00010_00011_00000_100110;	
52	**xori $s3,$s1,0x00ef**	$s3=$s1⊕0x00ef
	32'b001110_00001_00011_0000000011101111;	
56	**lw $s2,30($s4)**	$s2=memory[$s4+30]
	32'b100011_00100_00010_0000000000011110;	
60	**addi $s3,$s2,0x1234**	$s3=$s1+0x1234
	32'b001000_00010_00011_0001001000110100;	
64	**andi $s3,$s2,0x00ef**	$s3=$s1&0x00ef
	32'b001100_00010_00011_0000000011101111;	
68	**jr $sa**	PC=$ra
	32'b000000_11111_00000_00000_00000_001000;	
72	**xor $s3,$s1,$s2**	$s3=$s1⊕$s2
	32'b000000_00001_00010_00011_00000_100110;	
76	**xori $s3,$s1,0x00ef**	$s3=$s1⊕0x00ef
	32'b001110_00001_00011_0000000011101111;	
80	**sll $s3,$s2,3**	$s3=$s2<<3
	32'b000000_00000_00010_00011_00011_000000;	
84	**srl $s4,$s3,3**	$s4=$s3>>3
	32'b000000_00000_00011_00100_00011_000010;	
88	**sra $s5,$s3,3**	$s5=$s3>>>3
	32'b000000_00000_00011_00101_00011_000011;	
92	**sll $s3,$s2,3**	$s3=$s2<<3
	32'b000000_00000_00010_00011_00011_000000;	
96	**add $4,$s3,$s2**	$4=$s3+$s2
	32'b000000_00011_00010_00100_00000_100000;	
100	**sub $s5,$s3,$s2**	$s5=$s3-$s2
	32'b000000_00011_00010_00101_00000_100010;	
104	**lui $s6,16<<6**	$s6=16<<6
	32'b001111_00000_00110_0000000000010000;	
108	**add $4,$s6,$s2**	$4=$s6+$s2
	32'b000000_00110_00010_00100_00000_100000;	
112	**subi $s5,$s6,0xef**	$s5=$s6-0xef
	32'b001001_00110_00101_0000000011101111;	
116	**jal 0x0021901**	$ra=PC+4 $PC=(PC+4)_{31..28}(0x0021901)_{27..2}00$
	32'b000011_00 0000 0010 0001 1001 0000 0001;	

120	**xor $s3,$s1,$s2**	$s3=$s1⊕$s2
	32'b000000_00001_00010_00011_00000_100110;	
124	**xori $s3,$s1,0x00ef**	$s3=$s1⊕0x00ef
	32'b001110_00001_00011_0000000011101111;	

二、主要实验过程

（一）总体架构

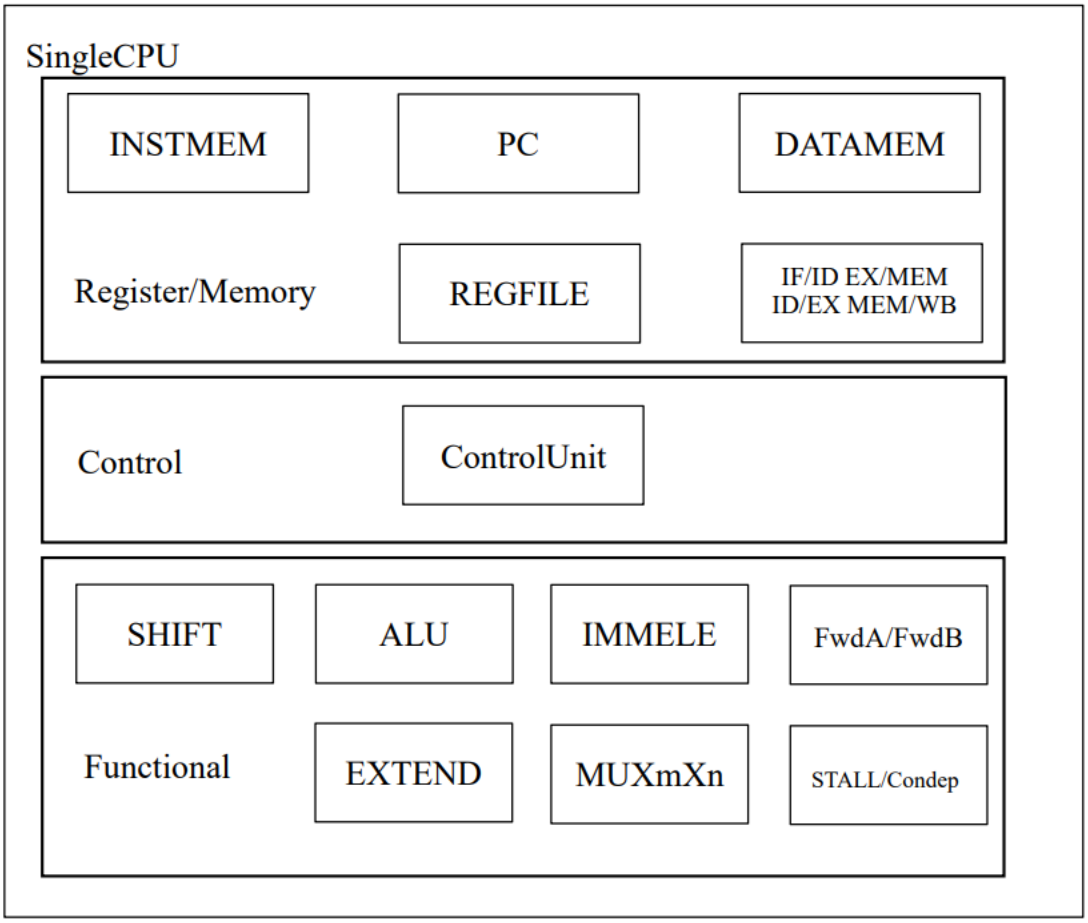

图 1 总体架构图

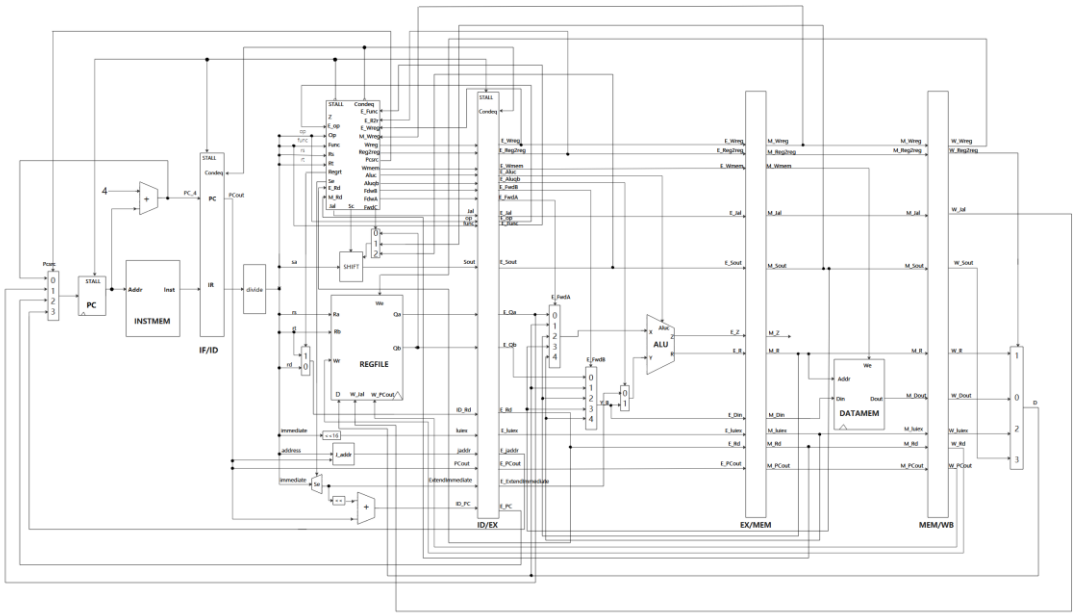

图 2 总体数据通路图

（二）数据通路分析

（1）R 型指令的数据通路

Step1:读取 PC，进入加法器进行 PC+4 操作，将结果作为下个周期的 PC 值

Step2:PC 输入地址到指令寄存器，指令寄存器根据地址[6:2]位输出指令

Step3:rs 输入到寄存器堆的 Ra，rt 输入到 Rb，rd 输入到 Wr

Step4:寄存器堆输出 Qa 到 ALU 的 X 口，Qb 到 ALU 的 Y 口

Step5:ALU 将结果经过选择器输出到寄存器堆的写入数据口，并根据 Wr 值存入对应寄存器

（2）R 型移位指令的数据通路

Step1:读取 PC，进入加法器进行 PC+4 操作，将结果作为下个周期的 PC 值

Step2:PC 输入地址到指令寄存器，指令寄存器根据地址[6:2]位输出指令

Step3:sa 输入 SHIFT 输入口，rt 输入到 Rb，rd 输入到 Wr

Step4:寄存器堆输出 Qb 到 SHIFT 的输入口

Step5:SHIFT 将结果经过选择器输出到寄存器堆的写入数据口，并根据 Wr 值存入对应寄存器

（3）I 型算术运算类指令的数据通路

Step1:读取 PC，进入加法器进行 PC+4 操作，将结果作为下个周期的 PC 值

Step2:PC 输入地址到指令寄存器，指令寄存器根据地址[6:2]位输出指令

Step3:rs 输入到寄存器堆的 Ra，rt 输入到 Wr，immediate 输入到扩展器

Step4:寄存器堆输出 Qa 到 ALU 的 X 口，扩展器结果输入到 ALU 的 Y 口

Step5:ALU 将结果经过选择器输出到寄存器堆的写入数据口，并根据 Wr 值存入对应寄存器

（4）I 型 lw 指令的数据通路

Step1:读取 PC，进入加法器进行 PC+4 操作，将结果作为下个周期的 PC 值

Step2:PC 输入地址到指令寄存器，指令寄存器根据地址[6:2]位输出指令

Step3:rs 输入到寄存器堆的 Ra，rt 输入到 Wr，immediate 输入到扩展器

Step4:寄存器堆输出 Qa 到 ALU 的 X 口，扩展器结果输入到 ALU 的 Y 口

Step5:ALU 将结果输入到数据寄存器的 Addr 端口

Step6:数据存储器根据地址[6:2]位访问并获取数据，Dout 端口将取得的数据输入到寄存器堆的写入数据口，根据 Wr 值存入对应寄存器

（5）I 型 sw 指令的数据通路

Step1:读取 PC，进入加法器进行 PC+4 操作，将结果作为下个周期的 PC 值

Step2:PC 输入地址到指令寄存器，指令寄存器根据地址[6:2]位输出指令

Step3:rs 输入到寄存器堆的 Ra，rt 输入到 Rb，immediate 输入到扩展器

Step4:寄存器堆输出 Qa 到 ALU 的 X 口，扩展器结果输入到 ALU 的 Y 口

Step5:ALU 将结果输入到数据寄存器的 Addr 端口

Step6:Qb 输入到数据寄存器的 Din 端口，数据存储器将寄存器 rt 的值写入该地址的存储单元

（6）I 型条件分支类指令的数据通路

Step1:读取 PC，进入加法器进行 PC+4 操作，将结果作为下个周期的备选 PC 值

Step2:PC 输入地址到指令寄存器，指令寄存器根据地址[6:2]位输出指令

Step3:rs 输入到寄存器堆的 Ra，rt 输入到 Rb，immediate 输入到扩展器

Step4:寄存器堆输出 Qa 到 ALU 的 X 接口，Qb 输入 ALU 的 Y 接口

Step5:扩展器结果左移 2 位输入到与 PC+4 的值相加的加法器中得到跳转地址

Step6:ALU 将输出 Z 传入控制单元进行判断

Step7:控制单元输出选择信号到 PC 选择器中

Step8:PC 选择器输出选择结果作为下一个 PC 值

（7）J 型指令的数据通路

Step1:读取 PC

Step2:PC 输入地址到指令寄存器，指令寄存器根据地址[6:2]位输出指令

Step3:PC+4 的最高 4 位与 address<<2 的值进行拼接得到跳转地址，并将结果输入到 PC 选择器中

Step4:选择器选择跳转地址作为下一个 PC 值

（8）jal 指令的数据通路

Step1:读取 PC，进入加法器进行 PC+4 操作

Step2:加法器将结果输入到寄存器堆的的 Jin 写入端口，准备写入$ra 寄

存器

Step3:PC 输入地址到指令寄存器，指令寄存器根据地址[6:2]位输出指令

Step4:PC+4 的最高四位与 address<<2 的值进行拼接得到跳转地址，并将结果输入到 PC 选择器中

Step5:选择器选择跳转地址作为下一个 PC 值

（9）jr 指令的数据通路

Step1:读取 PC

Step2:PC 输入地址到指令寄存器，指令寄存器根据地址[6:2]位输出指令

Step3:rs 输入到寄存器堆的 Ra 口

Step4:寄存器堆将 Qa 输出到 PC 选择器中作为跳转地址

Step5:PC 选择器选择跳转地址作为下一个 PC 值

（10）lui 指令的数据通路

Step1:读取 PC，进入加法器进行 PC+4 操作，将结果作为下个周期的 PC 值

Step2:PC 输入地址到指令寄存器，指令寄存器根据地址[6:2]位输出指令

Step3:rt 输入到寄存器堆的 Wr 接口

Step3:将 immediate 左移 16 位，其结果输入到 D 的多路选择器中

Step4:选择器将结果输出到寄存器堆的写入数据口，并存入 rt 寄存器中

（三）冒险解决

（1）寄存器堆写操作提前半个时钟周期

该设计可以解决间隔 2 条指令的数据冒险。

对于流水线 CPU 来说，需要写回寄存器堆的指令在第 5 时钟周期结束时写回，但寄存器的读/写操作时间实际上不到半个周期，故我们完全可以将寄存器堆的写操作提前半个周期，也即下降沿触发写操作，这样可以节省一个时钟周期的暂停时间。

（2）内部前推

该设计可以解决相邻（非 lw）或间隔 1 条指令的数据冒险。

计算型指令在 EX 级（第 3 个时钟周期）就计算出了结果，但在第 5 个时钟周期才写回；而 EX 级所需的两个寄存器的值在 ID 级就已经获得，等待了一个时钟周期才被使用。这样以来便浪费了一些时钟周期。

通过在 MEM、EX 级之间，以及 WB、EX 级间增加数据路径，使得 EX/MEM.R、MEM/WB.D 的值可以在写回寄存器堆之前就推到 EX 级作为 ALU 的输入进行计算，保证之后的指令运算结果是正确的。

可以在 ALU 的两个输入端前分别加上一个多路选择器，并用控制信号 FwdA、FwdB 控制多路选择器的选择。

（3）暂停流水线

该设计可以解决 lw、beq、bne 指令与相邻指令的数据冒险。

以 lw 指令为例，将第二条指令阻塞一个时钟周期，则其 EX 级与 lw 的 WB 级对齐，使得 MEM 输出的数据能在下一时钟周期推到第二条指令的 EX 级，完成运算。

（4）缩短分支的延迟

该设计可以解决条件分支指令的控制冒险，并提高效率。

虽然阻塞时钟周期能够实现控制冒险的解决，但对性能影响较大。我们可以使 CPU 尽早完成分支的决策，减少性能损失。具体操作为：在 EX 级判断条件分支指令的转移条件是否成立，若成立则消除紧跟条件分支指令进入流水线且分别处于 IF 级和 ID 级的后两条后续指令；设计 CONUNIT 的输出端口 Condep 连接寄存器组的 Clrn 端，以在合适时候清除错误指令。

（四）各模块设计

（1）PC 寄存器

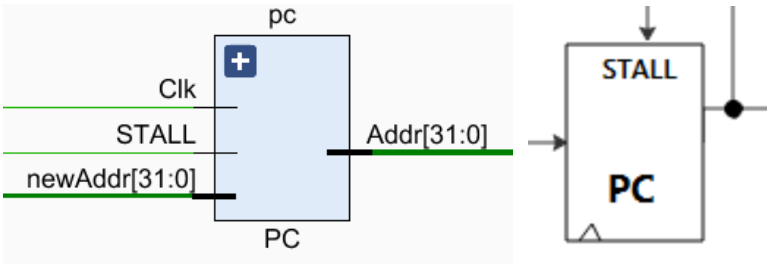

图 3 PC 寄存器

功能：根据时钟信号将输入的下一个 PC 值输出，并根据 STALL 信号选择是否阻塞。

（2）指令存储器

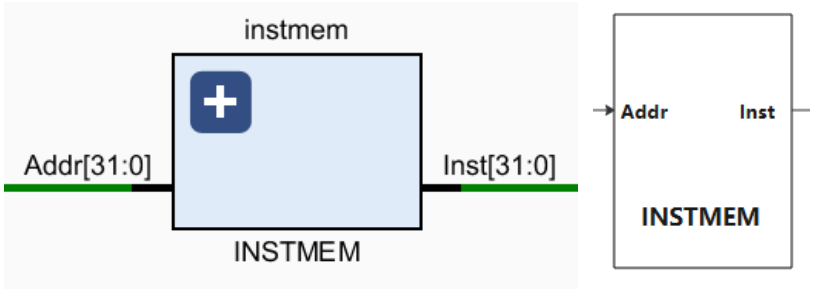

图 4 指令存储器

功能：存放实验所需使用的指令，根据 PC 输入的地址寻找对应指令并输出。预设指令的地址及作用详见一、实验要求（三）运行指令设计。

（3）指令分段模块

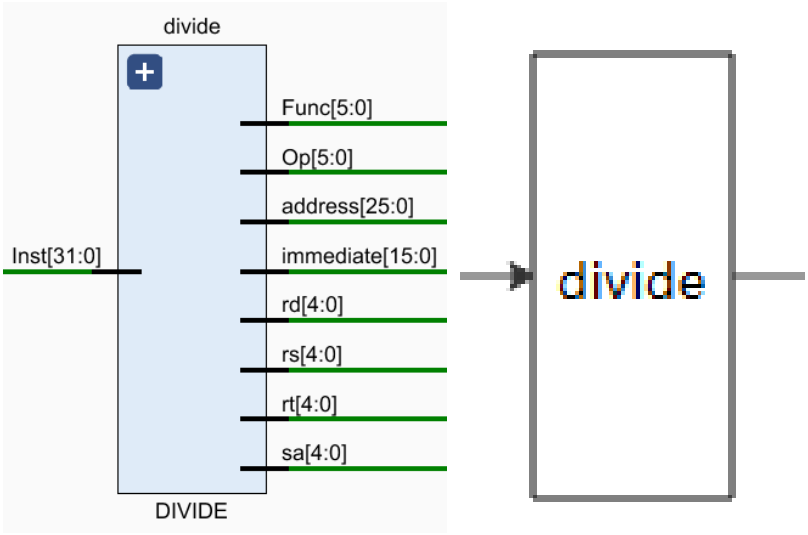

图 5 指令分段模块

功能：将指令存储器取出的指令不同字段分别输出，便于参与后续操作。

（4）寄存器堆

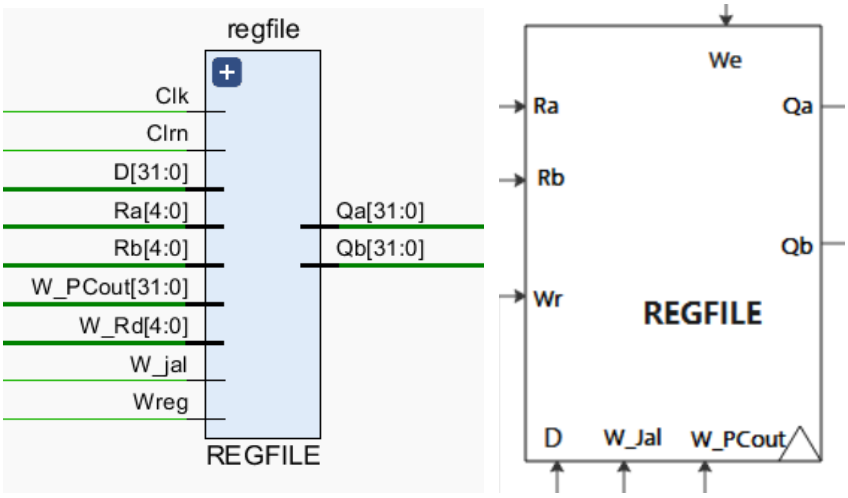

图 6 寄存器堆

功能：存放计算时需要使用的数据；根据 Ra，Rb 端口的输入来输出 Qa，Qb，根据 W_Rd 信号将 D 写入对应寄存器。初始化时我们将第 i 号寄存器的值设置为 i（i=0,1,...,30），第 31 号寄存器值设置为 80。

（5）算术逻辑单元

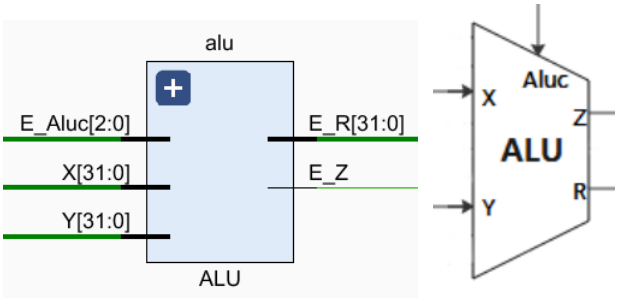

图 7 算术逻辑单元

功能：根据 E_Aluc 信号选择操作，对输入的 X 和 Y 进行加法、减法、按位与、按位或、按位异或的其中一种操作，并输出运算结果 R 和零标志信号 Z。

（6）数据存储器

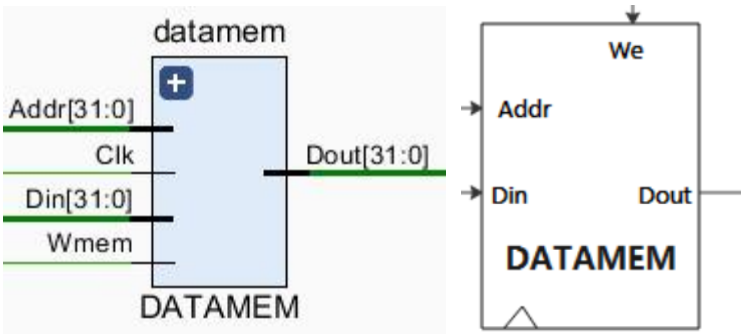

图 8 数据存储器

功能：存储数据，初始化时我们将第 i 个存储单元的值设置位为 i^2。根据 Addr 输入读出对应存储单元的数据，并通过 Dout 输出；或是根据 Addr 输入找到对应存储单元，并将 Din 的输入值存储到对应存储单元。

（7）控制单元

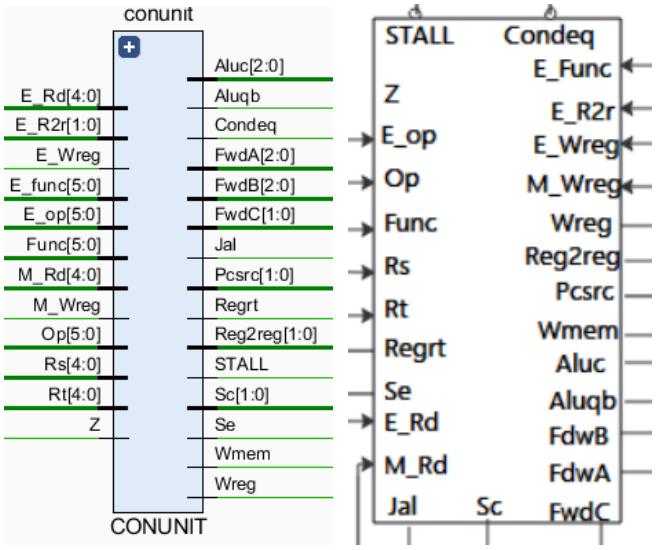

图 9 控制单元

功能：根据含指令 op、func 字段、零标志信号 Z、E_Rd 等反馈信号的输入信号来产生相应的控制信号，对整个 CPU 进行调控，使不同数据通路连通，得到正确的运行结果。

其具体输出信号与输入信号的关系如下表：

表 3 控制部件的输入输出信号真值表

输入端口				输出端口									
Op [5:0]	Func [5:0]	备注	Z	Regrt	Se	Wreg	Aluqb	Aluc [2:0]	Wmem	Pcsrc [1:0]	Reg2reg [1:0]	jal	Sc [1:0]
000000	100000	add	X	0	0	1	1	000	0	00	01	0	00
000000	100010	sub	X	0	0	1	1	001	0	00	01	0	00
000000	100100	and	X	0	0	1	1	010	0	00	01	0	00
000000	100101	or	X	0	0	1	1	011	0	00	01	0	00
001000	—	addi	X	1	1	1	0	000	0	00	01	0	00
001001	—	subi	X	1	0	1	0	001	0	00	01	0	00
001100	—	andi	X	1	0	1	0	010	0	00	01	0	00
001101	—	ori	X	1	0	1	0	011	0	00	01	0	00
100011	—	lw	X	1	1	1	0	000	0	00	00	0	00
101011	—	sw	X	1	1	0	0	000	1	00	01	0	00
000100	—	beq	0	1	1	0	1	001	0	00	01	0	00
000100	—	beq	1	1	1	0	1	001	0	10	01	0	00
000000	100110	xor	X	0	0	1	1	100	0	00	01	0	00
001110	—	xori	X	1	0	1	0	100	0	00	01	0	00
000000	000000	sll	X	0	0	1	1	000	0	00	11	0	00
000000	000010	srl	X	0	0	1	1	000	0	00	11	0	01
000000	000011	sra	X	0	0	1	1	000	0	00	11	0	10

000101	—	bne	0	1	1	0	1	001	0	10	01	0	00
000101	—	bne	1	1	1	0	1	001	0	00	01	0	00
001111	—	lui	X	1	1	1	0	000	0	00	10	0	00
000010	—	j	X	1	0	0	1	000	0	11	01	0	00
000011	—	jal	X	1	0	0	1	000	0	11	01	1	00
000000	001000	jr	X	0	0	1	1	000	0	01	01	0	00

（8）扩展器

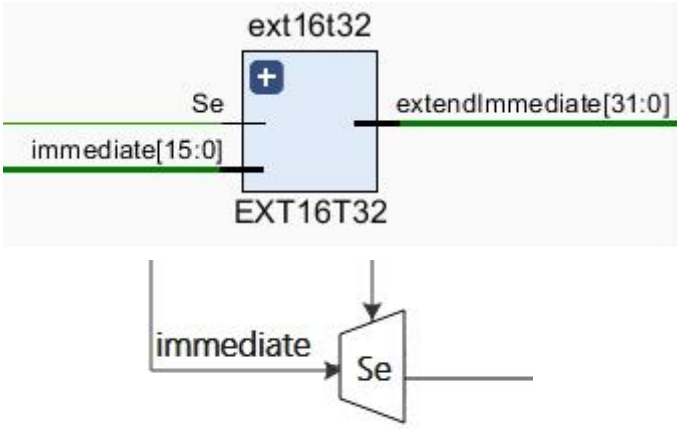

图 10 扩展器

功能：将 16 位立即数扩展成 32 位，并根据输入信号区分是零扩展还是符号扩展。

（9）多路选择器

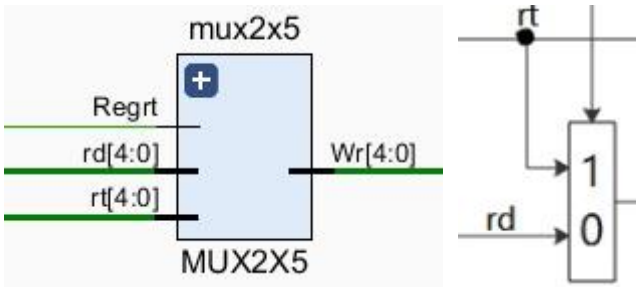

图 11 五位 2 选 1 多路选择器

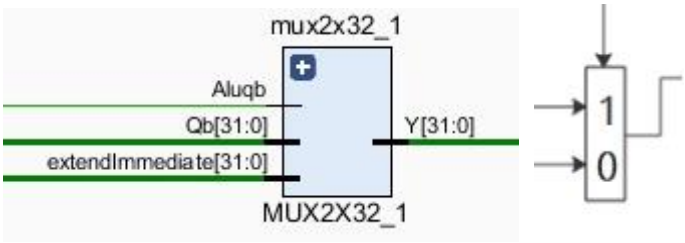

图 11 三十二位 2 选 1 多路选择器

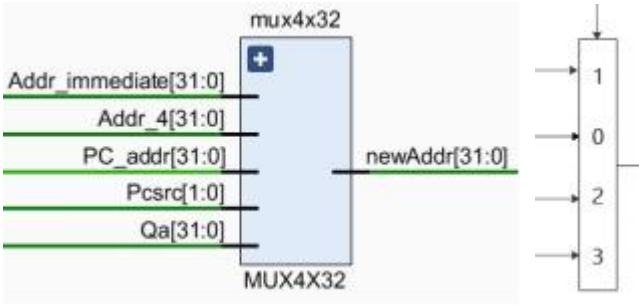

图 12 三十二位 4 选 1 多路选择器(地址)

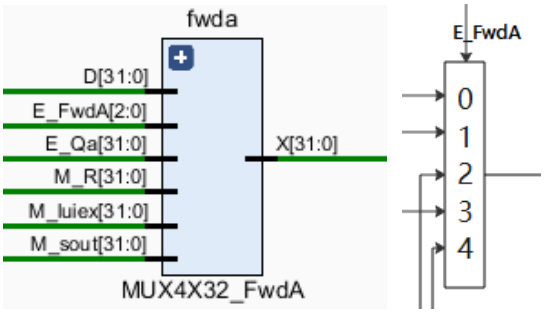

图 13 三十二位 4 选 1 多路选择器(FwdA)

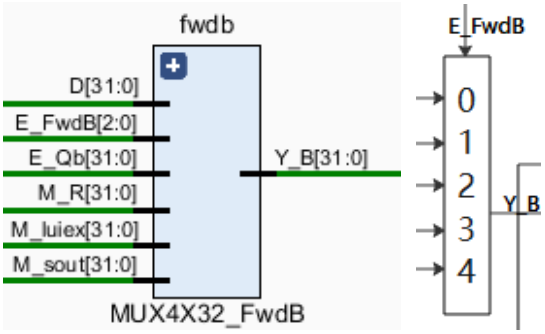

图 14 三十二位 4 选 1 多路选择器(FwdB)

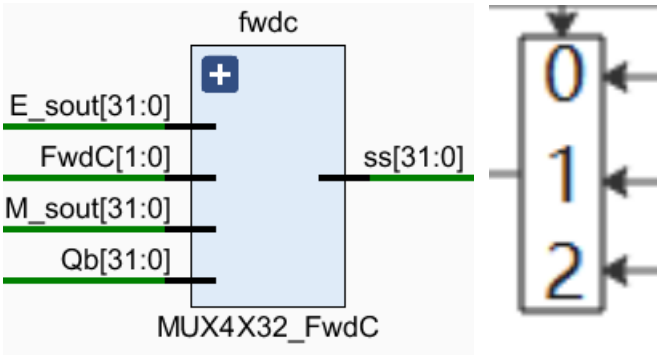

图 15 三十二位 3 选 1 多路选择器(FwdC)

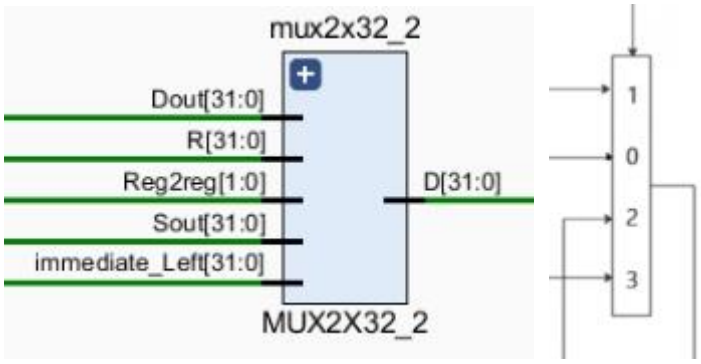

图 16 三十二位 4 选 1 多路选择器(写回)

功能：根据输入的控制信号判断从多路输入中选择哪个信号作为输出信号。

（10）移位器

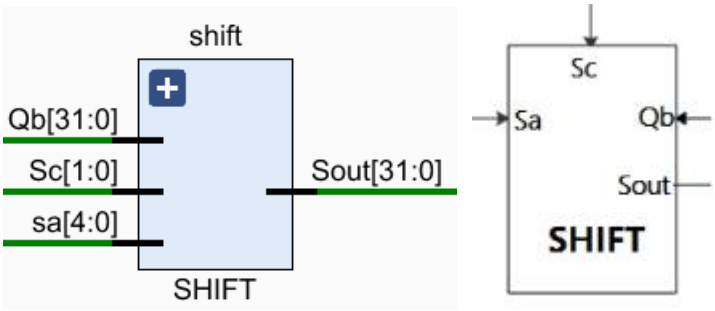

图 17 移位器

功能：根据输入的控制信号 Sc 判断移位方式，根据 sa 判断移位位数，并将移位结果通过 Sout 输出。

（11）支持 lui 的功能部件

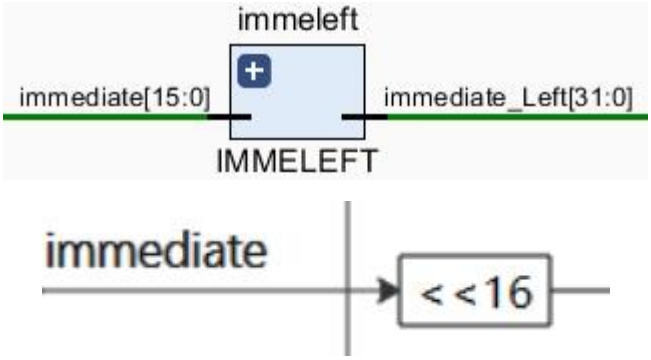

图 18 支持 lui 的功能部件

功能：为支持 lui 设置高位操作，将输入的立即数左移 16 位，并将结果输出。

（12）IF/ID 寄存器组

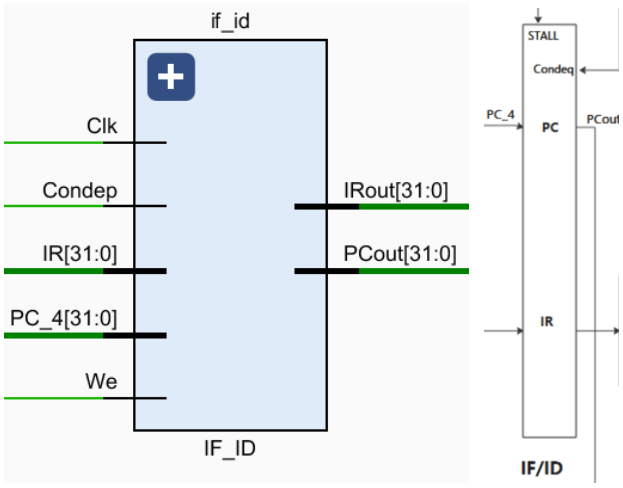

图 19 IF/ID 寄存器组

功能：将指令和 PC 寄存，在时钟周期结束时输出指令和 PC，并根据 STALL、Condep 信号选择是否阻塞或清零。

（13）ID/EX 寄存器组

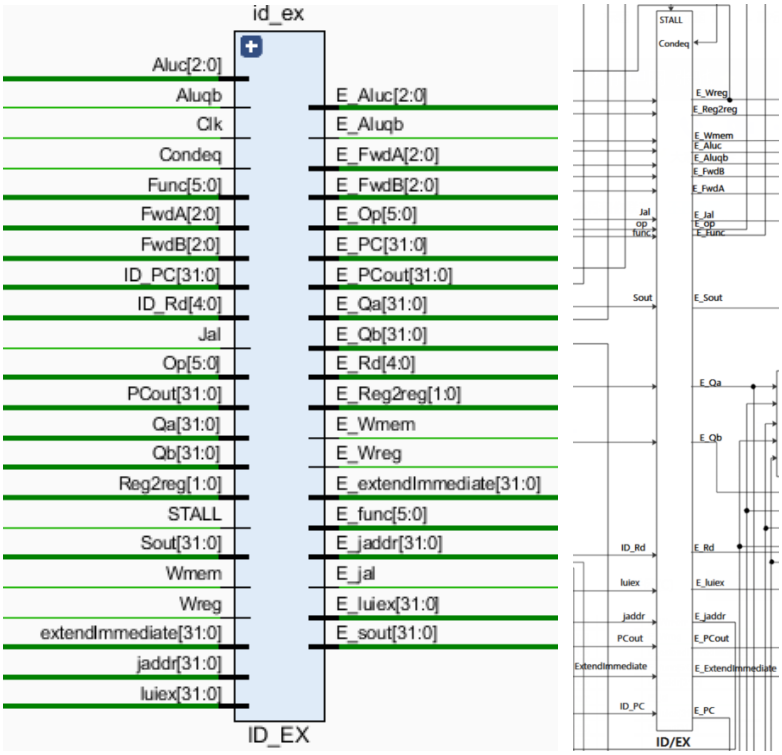

图 20 ID/EX 寄存器组

功能：将运算结果、控制信号、指令部分字段和 PC 寄存，在时钟周期结束时输出，并根据 STALL、Condep 信号选择是否阻塞或清零。

（14）EX/MEM 寄存器组

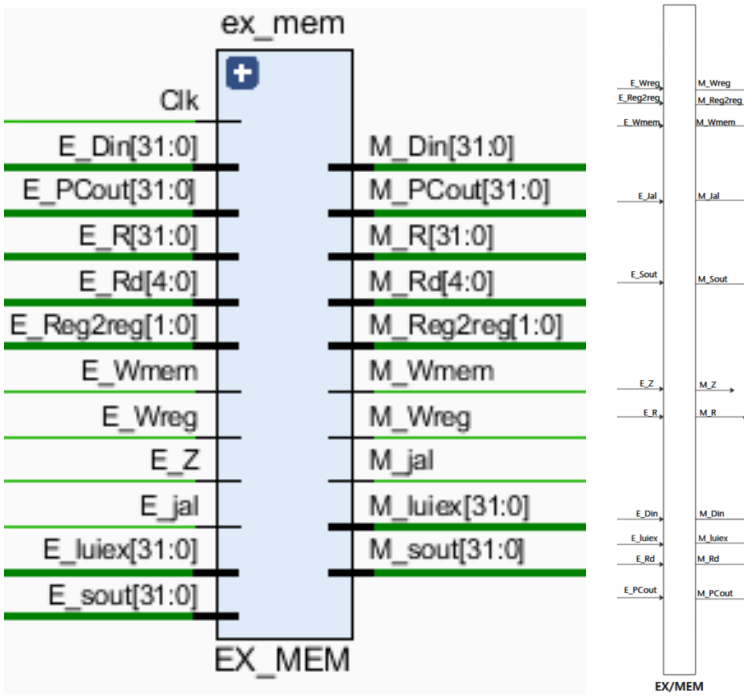

图 21 ID/EX 寄存器组

功能：将运算结果、控制信号、指令部分字段和 PC 寄存，在时钟周期结束时输出。

（15）MEM/WB 寄存器组

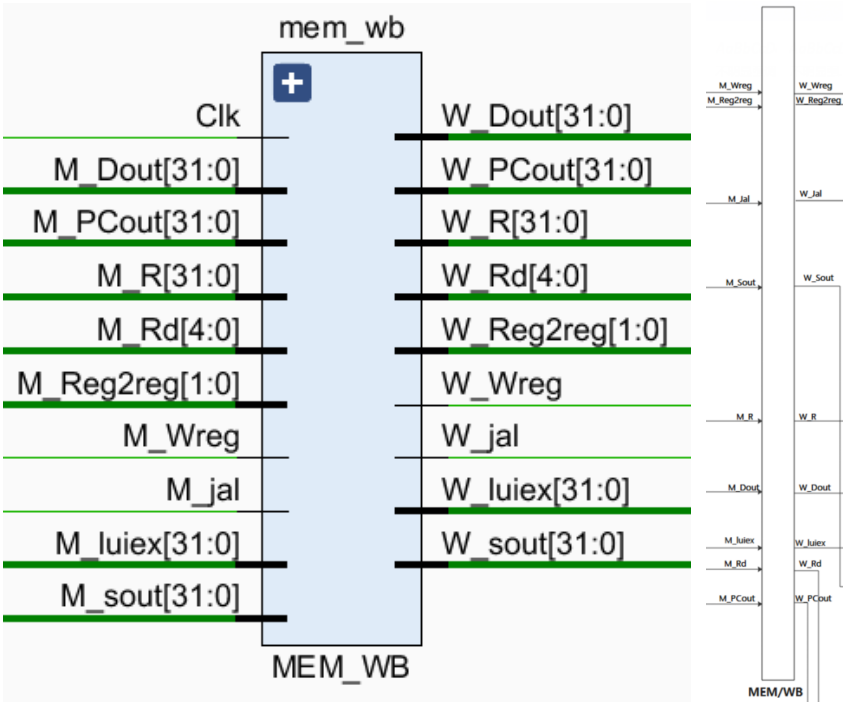

图 22 ID/EX 寄存器组

功能：将运算结果、控制信号、指令部分字段和 PC 寄存，在时钟周期结束时输出。

（16）加法器

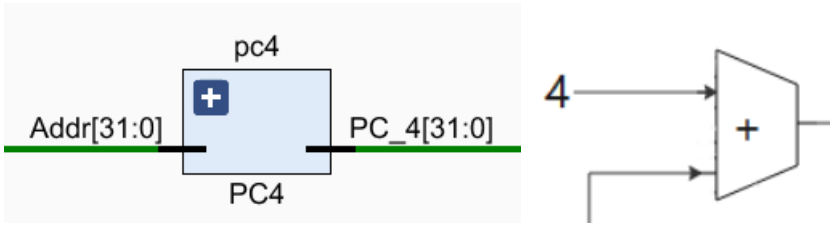

图 23 实现 PC+4 功能的加法器

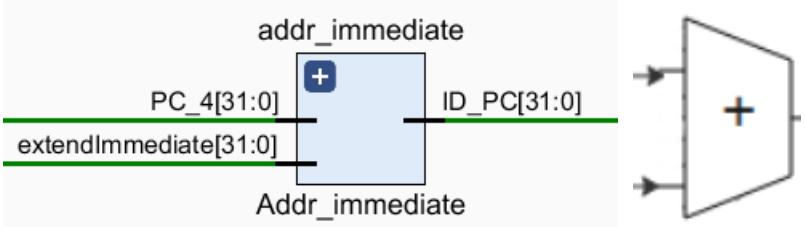

图 24 实现 beq、bne 指令跳转地址的加法器

功能：分别生成跳转地址，对于图 24，PC+4 下一个地址；对于图 25，PC+4 加上立即数作为条件成立（不成立）时的跳转地址。

（17）J 型指令的跳转地址

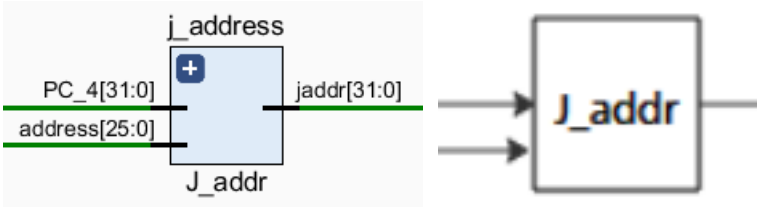

图 25 实现 J 型指令拼接的部件

功能：生成跳转地址，PC+4 作为最高 4 位，立即数左移 2 位作为剩下 28 位，完成跳转地址的拼接。

（五）封装

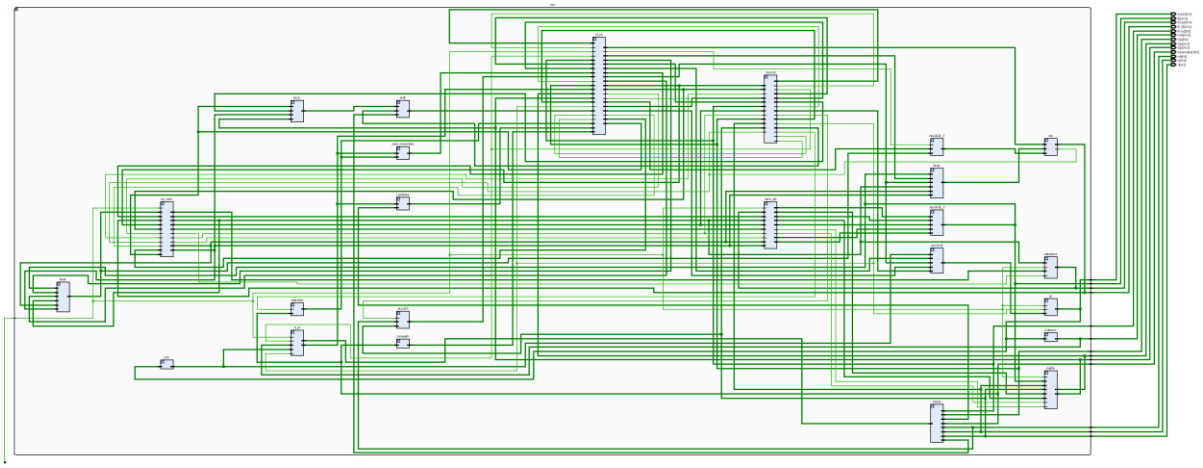

图 26 封装完成的单周期 CPU（全览）

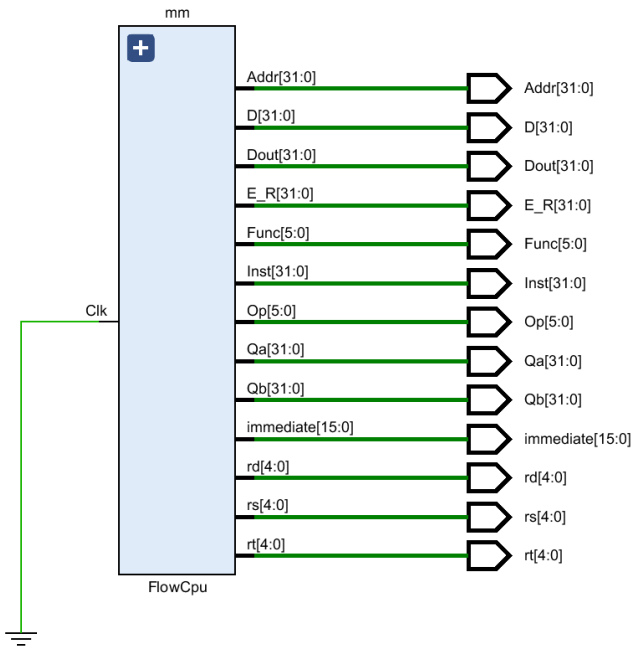

图 27 封装完成的单周期 CPU

功能：将各模块连接起来，形成一个完整的流水线 CPU。预留输入输出接口，便于输入测试信号进行测试，以及方便输出波形。

三、主要实现代码

（一）主要功能代码

（1）PC 寄存器

```
`timescale 1ns/1ps
module PC(Clk,newAddr,Addr,STALL);
input Clk,STALL;                    //输入时钟信号、暂停信号
input [31:0] newAddr;               //输入新地址
output reg [31:0] Addr;             //输出当前地址
initial    begin
   Addr<=0;
end
always@(posedge Clk)
begin
      if(STALL)                     //如果STALL=1 输出新地址
         Addr <= newAddr;
```

```
    else
        Addr <= Addr;              //如果STALL=0 输出维持不变
end
endmodule
```

（2）指令存储器

```
`timescale 1ns/1ps
module INSTMEM(Addr, Inst);
input [31:0] Addr;
output [31:0] Inst;
wire [31:0] ram [0:63];
assign ram[5'h00] = 32'b000000_00001_00010_00011_00000_100000; //add $3=$1+$2
assign ram[5'h01] = 32'b000000_00001_00010_00011_00000_100010; //sub $3=$1-$2
assign ram[5'h02] = 32'b000000_00001_00010_00011_00000_100100; //and $3=$1&$2
assign ram[5'h03] = 32'b000000_00001_00010_00011_00000_100101; //or $3=$1|$2
assign ram[5'h04] = 32'b001000_00001_00011_0001001000110100; //addi $3=$1+4660
assign ram[5'h05] = 32'b001100_00001_00011_0000000011101111; //andi $3=$1&239
assign ram[5'h06] = 32'b001101_00001_00011_0000000011101111; //ori $3=$1|239
data adventure lw rs/ori rd
assign ram[5'h07] = 32'b100011_00011_00101_0000000000000001; //lw $5=datamem($3+1)
data adventure lw rs/ori rd
assign ram[5'h08] = 32'b101011_00011_00001_0000000000000001; //sw datamem($3+1)=$1
assign ram[5'h09] = 32'b000100_00100_00101_0000000000010000; //beq if($4=$5) goto
pc+4+16*4
assign ram[5'h0a] = 32'b000101_00101_00101_0000000000010000; //bne if($5!=$5) goto
pc+4+16*4
assign ram[5'h0b] = 32'b000010_00000001000011001000_01100;//j goto ram[12]-xor
j control adventure
assign ram[5'h0c] = 32'b000000_00001_00010_00011_00000_100110; //xor $3=$1@$2
assign ram[5'h0d] = 32'b001110_00001_00011_0000000011101111; //xori $3=$1@239
assign ram[5'h0e] = 32'b100011_00100_00010_0000000000011110; //lw $2=datamem($4+30)
lw data adventure
assign ram[5'h0f] = 32'b001000_00010_00011_0001001000110100; //addi $3=$2+4660
lw data adventure
assign ram[5'h10] = 32'b001100_00010_00011_0000000011101111; //andi $3=$2&239
assign ram[5'h11] = 32'b000000_11111_00000_00000_00000_001000; //jr $ra goto $31
ram[20]-sll jr control adventure
assign ram[5'h12] = 32'b000000_00001_00010_00011_00000_100110; //xor $3=$1@$2
assign ram[5'h13] = 32'b001110_00001_00011_0000000011101111; //xori $3=$1@239
assign ram[5'h14] = 32'b000000_00000_00010_00011_00011_000000; //sll $3=$2<<3
assign ram[5'h15] = 32'b000000_00000_00011_00100_00011_000010; //srl $4=$3>>3 logical
```

```
data adventure
assign ram[5'h16] = 32'b000000_00000_00011_00101_00011_000011; //sra $5=$3>>>3 math data adventure
assign ram[5'h17] = 32'b000000_00000_00010_00011_00011_000000; //sll $3=$2<<3
assign ram[5'h18] = 32'b000000_00011_00010_00100_00000_100000; //add $4=$3+$2 data adventure
assign ram[5'h19] = 32'b000000_00011_00010_00101_00000_100010; //sub $5=$3-$2 data adventure
assign ram[5'h1a] = 32'b001111_00000_00110_0000000000010000; //lui $6=2^20
assign ram[5'h1b] = 32'b000000_00110_00010_00100_00000_100000; //add $4=$6+$2 data adventure
assign ram[5'h1c] = 32'b001001_00110_00101_0000000011101111; //subi $5=$6-239 data adventure
assign ram[5'h1d] = 32'b000011_00000001000011001000_00001;//jal goto ram[1]
assign ram[5'h1e] = 32'b000000_00001_00010_00011_00000_100110; //xor $3=$1@$2
assign ram[5'h1f] = 32'b001110_00001_00011_0000000011101111; //xori $3=$1@239*/
assign Inst = ram[Addr[6:2]];
endmodule
```

（3）指令分段模块

```
`timescale 1ns/1ps
module DIVIDE(Inst,Op,rs,rt,rd,immediate,Func,address,sa);
output [5:0] Op;
output [5:0] Func;
output [4:0] rs;
output [4:0] rt;
output [4:0] rd;
output [15:0] immediate;
output [25:0] address;
output [4:0] sa;
input [31:0] Inst;
assign Op = Inst[31:26];
assign rs = Inst[25:21];
assign rt = Inst[20:16];
assign rd = Inst[15:11];
assign immediate = Inst[15:0];
assign Func = Inst[5:0];
assign address = Inst[25:0];
assign sa = Inst[10:6];                    //以上代码实现将指令各字段分开输出
endmodule
```

（4）寄存器堆

```
`timescale 1ns/1ps
module REGFILE(Wreg,Ra,Rb,W_Rd,Clk,D,Qa,Qb,W_jal,W_PCout,Clrn);
input  Clk,Wreg,W_jal,Clrn;     //输入时钟信号、目的寄存器写入信号、jal指令写入信号、清零信号
input [4:0] Ra;
input [4:0] Rb;
input [4:0] W_Rd;          //输入目的寄存器编号
input [31:0] D,W_PCout;       //输入目的寄存器写入值、jal指令的PC+4 写入值
output [31:0] Qa,Qb;
reg [31:0] register[0:31];
integer i;
initial begin                                          //初始化
                for(i = 0; i < 32; i = i + 1)
                  begin
                     if(i!=31) begin
                           register[i] <= i;       //前 31 个寄存器存入编号对应值
                      end
                     else begin
                       register[i] <= 80;       //第 32 个寄存器存入十进制 80 的值
                      end                  //目的是使jr指令跳转到ram[20]即sll指令
                  end
end
assign Qa=register[Ra];
assign Qb=register[Rb];
always@(negedge Clk)                  //实现下降沿写入
   begin
      if(W_jal)
         register[31]<=W_PCout;     //当执行jal指令时，PC+4 写入$ra
   end
always@(negedge Clk)                //实现下降沿写入
            begin
                 if (Wreg&&W_Rd != 0)
                 register[W_Rd]=D;         //写入目的寄存器操作
            end
always@(Clk or Clrn)
  begin
  if(Clrn==0)
  begin
     for(i = 0; i < 32; i = i + 1)
         begin
           register[i] <= 0;          //Clrn=0 时实现清零功能
         end
```

```
      end
  end
endmodule
```

（5）算术逻辑单元

```
`timescale 1ns/1ps
module ALU(X,Y,E_Aluc,E_R,E_Z);
input [31:0] X,Y;
input [2:0] E_Aluc;
output reg [31:0] E_R;
output reg E_Z;
always@(*)
   begin
    case(E_Aluc)
       3'b000:E_R=X+Y;                          //加
       3'b001:E_R=X-Y;                          //减
       3'b010:E_R=X&Y;                          //按位与
       3'b011:E_R=X|Y;                          //按位或
       3'b100:E_R=(X&(~Y))|((~X)&Y);            //异或
       default:E_R=0;
    endcase
if(E_R)
       E_Z=0;
    else
       E_Z=1;
end
endmodule
```

（6）数据存储器

```
`timescale 1ns/1ps
module DATAMEM(Wmem,Addr,Din,Dout,Clk);
input [31:0] Addr,Din;
input Clk,Wmem;
output [31:0] Dout;
reg [31:0] Ram[0:31];
assign Dout=Ram[Addr[6:2]];
always@(posedge Clk)
   begin
```

```
    if(Wmem)
        Ram[Addr[6:2]]<=Din;                //写入操作
end
integer i;
initial begin                               //初始化
for(i=0;i<32;i=i+1)
    Ram[i]=i*i;                             //第i个寄存器存储其编号的平方值
end
endmodule
```

（7）控制单元

```
`timescale 1ns / 1ps
module
CONUNIT(Op,Func,Z,Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Pcsrc,Reg2reg,STALL,Condeq,E_R2r,
E_Wreg,M_Wreg,FwdA,FwdB,M_Rd,E_Rd,E_op,Rs,Rt,E_func,Sc,FwdC,Jal);
input [5:0] Op,Func,E_op,E_func;
input Z,E_Wreg,M_Wreg;
input [1:0] E_R2r;
input [4:0] Rs,Rt,M_Rd,E_Rd;
output reg [1:0] Sc,FwdC; //输出sll、sra、srl的选择信号及其被移位值的选择信号
output reg [2:0] FwdA,FwdB;
output reg Regrt,Se,Wreg,Aluqb,Wmem,Condeq;
output reg [1:0] Pcsrc,Reg2reg;
output reg [2:0] Aluc;
output reg STALL,Jal;          //Jal为jal指令写入的控制信号
reg stallo,E_beq,E_bne,condeqo,a,E_j,E_jr,E_jal;
initial
  begin
{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Condeq,Pcsrc,STALL,Sc}=16'b0000_000_0_00_1_
00_1_00;
    FwdA=3'b000;
    FwdB=3'b000;
    FwdC=2'b00;
end
always@(Op or Z or Func)        //按照控制信号输出表输出控制信号
  begin
    case(Op)
      6'b000000:
        begin
          case(Func)
```

```
        6'b100000:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b0011_000_001_00;//add
        6'b100010:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b0011_001_001_00;//sub
        6'b100100:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b0011_010_001_00;//and
        6'b100101:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b0011_011_001_00;//or
        6'b100110:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b0011_100_001_00;//xor
        6'b000000:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b0011_000_011_00;//sll
        6'b000010:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b0011_000_011_01;//srl
        6'b000011:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b0011_000_011_10;//sra
        6'b001000:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b0011_000_001_00;//jr
            endcase
                end
        6'b001000:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1110_000_001_00;//addi
        6'b001001:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1010_001_001_00;//subi
        6'b001100:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1010_010_001_00;//andi
          6'b001101:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1010_011_001_00;//ori
          6'b001110:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1010_100_001_00;//xori
          6'b100011:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1110_000_000_00;//lw
          6'b101011:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1100_000_101_00;//sw
          6'b001111:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1110_000_010_00;//lui
          6'b000100:
            begin
                case(Z)
1'b0:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1101_001_001_00;//beq
1'b1:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1101_001_001_00;//beq
                endcase
            end
          6'b000101:
            begin
                case(Z)
1'b0:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1101_001_001_00;//bne
1'b1:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1101_001_001_00;//bne
                endcase
            end
          6'b000010:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1001_000_001_00;//j
          6'b000011:{Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Reg2reg,Sc}=12'b1001_000_001_00;//jal
        endcase
    end
always@(Rs or Rt or M_Rd or E_Rd or E_Wreg or M_Wreg)
begin
          FwdA=3'b000;
if((Rs==E_Rd)&(E_Rd!=0)&(E_Wreg==1))
        begin
          if(E_R2r==2'b11)       //判断sll sra srl指令之后的数据冒险，实现内部前推
              begin
```

```
            FwdA=3'b011;
          end
        else
          begin
            if(E_op==6'b001111) //判断 lui指令之后的数据冒险，实现内部前推
              FwdA=3'b100;
            else
              FwdA=3'b010;        //实现M_R的内部前推
          end
    end
else
    begin
      if((Rs==M_Rd)&(M_Rd!=0)&(M_Wreg==1))
        begin
      FwdA=3'b001;              //实现D的内部前推
        end
    end
end
always@(Rs or Rt or M_Rd or E_Rd or E_Wreg or M_Wreg)
begin
      FwdB=3'b000;
if(((Rt==E_Rd)&((Op==6'b000000)|(Op==6'b101011)|(Op==6'b000100)|(Op==6'b000101)))&(E
_Rd!=0)&(E_Wreg==1))
    begin
      if(E_R2r==2'b11)      //判断sll sra srl指令之后的数据冒险，实现内部前推
          begin
            FwdB=3'b011;
          end
      else
          begin
            if(E_op==6'b001111) //判断lui指令之后的数据冒险，实现内部前推
              FwdB=3'b100;
            else
              FwdB=3'b010;      //实现M_R的内部前推
          end
    end
else
    begin
if(((Rt==M_Rd)&((Op==6'b000000)|(Op==6'b101011)|(Op==6'b000100)|(Op==6'b000101)))&(
M_Rd!=0)&(M_Wreg==1))
          begin
      FwdB=3'b001;          //实现D的内部前推
          end
    end
```

```
end
always@(Rs or Rt or M_Rd or E_Rd or E_Wreg or M_Wreg)
begin                          //实现sll、sra、srl三者之间的数据冒险
        FwdC=2'b00;          //接入Qa值
if((Rt==E_Rd)&(E_Rd!=0)&(E_Wreg==1))
      begin
        FwdC=2'b10;          //实现E_sout的内部前推
      end
else
      begin
        if((Rt==M_Rd)&(M_Rd!=0)&(M_Wreg==1))
          begin
         FwdC=2'b01;       //实现M_sout的内部前推
          end
      end
end
always@(*)       //判断lw指令的数据冒险，实现暂停功能
begin
  stallo<=((Rs==E_Rd)|(Rt==E_Rd))&(E_R2r==0)&(E_Rd!=0)&(E_Wreg==1);
  STALL<=~stallo;     //STALL信号已在控制部件内部取反
end
always@(*)       //解决beq、bne、j、jr、jal指令的控制冒险
begin
  E_beq<=(~E_op[5])&(~E_op[4])&(~E_op[3])&(E_op[2])&(~E_op[1])&(~E_op[0]);
  E_bne<=(~E_op[5])&(~E_op[4])&(~E_op[3])&(E_op[2])&(~E_op[1])&(E_op[0]);
  E_j<=(~E_op[5])&(~E_op[4])&(~E_op[3])&(~E_op[2])&(E_op[1])&(~E_op[0]);
  E_jr<=(~E_func[5])&(~E_func[4])&(E_func[3])&(~E_func[2])&(~E_func[1])&(~E_func[0]);
  E_jal<=(~E_op[5])&(~E_op[4])&(~E_op[3])&(~E_op[2])&(E_op[1])&(E_op[0]);
  condeqo<=(E_beq&Z)|(E_bne&~Z)|(E_j)|(E_jr)|(E_jal);
  Condeq<=~condeqo;
  if(E_j|E_jal)
     begin
       Pcsrc<=2'b11;     //jal、j指令的跳转地址
     end
  else
     begin
     if(E_jr)
       begin
         Pcsrc<=2'b01;     //jr指令的跳转地址
       end
     else
       begin
         a<=(E_beq&Z)|(E_bne&~Z);
         Pcsrc<={a,1'b0};
```

```
        end
      end
end
always@(*)          //实现jal指令写入的控制信号
begin
if(Op==6'b000011)
   Jal=1;
else
   Jal=0;
end
endmodule
```

（8）扩展器

```
`timescale 1ns / 1ps
module EXT16T32(Se,immediate,extendImmediate);
input Se;
input [15:0] immediate;
output [31:0] extendImmediate;     //输出扩展后的立即数
wire [31:0] E0,E1;
wire [15:0] e={16{immediate[15]}};
parameter z=16'b0;
assign E0={z,immediate};
assign E1={e,immediate};
function [31:0]select;
input [31:0] E0,E1;
input Se;
case(Se)
1'b0:select=E0;         //零扩展
1'b1:select=E1;         //符号扩展
endcase
endfunction
assign extendImmediate=select(E0,E1,Se);
endmodule
```

（9）多路选择器

```
//（输出新地址的 32 位四选一多路选择器）
`timescale 1ns/1ps
module MUX4X32(Addr_4,E_PC,Pcsrc,newAddr,E_jaddr,E_Qa);
input [31:0] Addr_4,E_PC,E_jaddr,E_Qa;
```

```
input [1:0] Pcsrc;
output [31:0] newAddr;
function [31:0]select;
input [31:0] Addr_4,E_PC,E_jaddr,E_Qa;
input [1:0] Pcsrc;
case(Pcsrc)
2'b00:select=Addr_4;                //输出PC+4 的值
2'b01:select=E_Qa;                  //输出jr指令的跳转地址
2'b10:select=E_PC;                  //beq、bne指令的跳转地址
2'b11:select=E_jaddr;               //jal、j指令的跳转地址
endcase
endfunction
assign newAddr=select(Addr_4,E_PC,E_jaddr,E_Qa,Pcsrc);
endmodule

//（Wr前的五位二选一多路选择器）
`timescale 1ns/1ps
module MUX2X5(rt,rd,Regrt,ID_Rd);     //选择目的寄存器
input [4:0] rt,rd;
output [4:0] ID_Rd;
input Regrt;
function [4:0]select;
input [4:0] rd,rt;
input Regrt;
case(Regrt)
1'b0:select=rd;
1'b1:select=rt;
endcase
endfunction
assign ID_Rd=select(rd,rt,Regrt);
endmodule

//（SHIFT输入接口的 32 位三选一多路选择器
`timescale 1ns / 1ps
module MUX4X32_FwdC(FwdC,Qb,E_sout,M_sout,ss);     //选择被移位的值
input [31:0] Qb,E_sout,M_sout;
input [1:0] FwdC;
output [31:0] ss;
function [31:0]select;
input [31:0] Qb,E_sout,M_sout;
input [1:0] FwdC;
case(FwdC)
2'b00:select=Qb;                    //$rt的值
2'b01:select=M_sout;                //MEM级需要写入$rt的值
```

```
2'b10:select=E_sout;              //EX级需要写入$rt的值
endcase
endfunction
assign ss=select(Qb,E_sout,M_sout,FwdC);
endmodule

//（ALU部件X接口前的 32 位五选一多路选择器）
`timescale 1ns/1ps
module MUX4X32_FwdA(E_Qa,D,M_R,E_FwdA,X,M_sout,M_luiex);//选择X值
input [31:0] E_Qa,D,M_R,M_sout,M_luiex;
input [2:0] E_FwdA;
output [31:0] X;
function [31:0]select;
input [31:0] E_Qa,D,M_R,M_sout,M_luiex;
input [2:0] E_FwdA;
case(E_FwdA)
3'b000:select=E_Qa;            //Qa的值
3'b001:select=D;               //WB级需要写入目的寄存器的值
3'b010:select=M_R;             //MEM级需要写入目的寄存器的值
3'b011:select=M_sout;          // MEM级移位指令需要写入目的寄存器的值
3'b100:select=M_luiex;         // MEM级lui指令需要写入目的寄存器的值
endcase
endfunction
assign X=select(E_Qa,D,M_R,M_sout,M_luiex,E_FwdA);
endmodule

//（Qb后的 32 位五选一多路选择器）
`timescale 1ns/1ps
module MUX4X32_FwdB(E_Qb,D,M_R,E_FwdB,Y_B,M_sout,M_luiex);//选择Y
input [31:0] E_Qb,D,M_R,M_sout,M_luiex;
input [2:0] E_FwdB;
output [31:0] Y_B;
function [31:0]select;
input [31:0] E_Qb,D,M_R,M_sout,M_luiex;
input [2:0] E_FwdB;
case(E_FwdB)
3'b000:select=E_Qb;            //Qa的值
3'b001:select=D;               //WB级需要写入目的寄存器的值
3'b010:select=M_R;             //MEM级需要写入目的寄存器的值
3'b011:select=M_sout;          // MEM级移位指令需要写入目的寄存器的值
3'b100:select=M_luiex;         // MEM级lui指令需要写入目的寄存器的值
endcase
endfunction
assign Y_B=select(E_Qb,D,M_R,M_sout,M_luiex,E_FwdB);
```

```
endmodule

//（Y接口前的 32 位二选一多路选择器）
`timescale 1ns/1ps
module MUX2X32_1(Y_B,E_extendImmediate,E_Aluqb,Y);
input [31:0] E_extendImmediate,Y_B;
output [31:0] Y;
input E_Aluqb;
function [31:0]select;
input [31:0] E_extendImmediate,Y_B;
input [1:0] E_Aluqb;
case(E_Aluqb)
1'b0:select=E_extendImmediate;        //输出扩展后的立即数
1'b1:select=Y_B;                      //输出Qb后 32 位五选一选择器的输出值
endcase
endfunction
assign Y=select(E_extendImmediate,Y_B,E_Aluqb);
endmodule

//（选择写入目的寄存器的 32 位四选一多路选择器）
`timescale 1ns/1ps
module MUX2X32_2(W_R,W_Dout,W_Reg2reg,D,W_sout,W_luiex);
input [31:0] W_Dout,W_R,W_sout,W_luiex;
output [31:0] D;
input [1:0] W_Reg2reg;
function [31:0]select;
input [31:0] W_R,W_Dout,W_sout,W_luiex;
input [1:0] W_Reg2reg;
case(W_Reg2reg)
2'b01:select=W_R;                     //输出R值
2'b00:select=W_Dout;                  //输出Dout值
2'b10:select=W_luiex;                 //输出lui指令设置高位后的值
2'b11:select=W_sout;                  //输出移位指令移位后的值
endcase
endfunction
assign D=select(W_R,W_Dout,W_sout,W_luiex,W_Reg2reg);
endmodule
```

（10）移位器

```
`timescale 1ns / 1ps
module SHIFT(Qb,sa,Sc,Sout);          //实现sll、sra、srl指令功能
```

```
input [31:0] Qb;
input [4:0] sa;
input [1:0] Sc;
output [31:0] Sout;
wire [31:0] a,c;
reg [31:0] b;
always @(*) begin
	case(sa)
		5'b00000:b<=Qb;
		5'b00001:b<={{Qb[31]},Qb[31:1]};
		5'b00010:b<={{2{Qb[31]}},Qb[31:2]};
		5'b00011:b<={{3{Qb[31]}},Qb[31:3]};
		5'b00100:b<={{4{Qb[31]}},Qb[31:4]};
		5'b00101:b<={{5{Qb[31]}},Qb[31:5]};
		5'b00110:b<={{6{Qb[31]}},Qb[31:6]};
		5'b00111:b<={{7{Qb[31]}},Qb[31:7]};
		5'b01000:b<={{8{Qb[31]}},Qb[31:8]};
		5'b01001:b<={{9{Qb[31]}},Qb[31:9]};
		5'b01010:b<={{10{Qb[31]}},Qb[31:10]};
		5'b01011:b<={{11{Qb[31]}},Qb[31:11]};
		5'b01100:b<={{12{Qb[31]}},Qb[31:12]};
		5'b01101:b<={{13{Qb[31]}},Qb[31:13]};
		5'b01110:b<={{14{Qb[31]}},Qb[31:14]};
		5'b01111:b<={{15{Qb[31]}},Qb[31:15]};
		5'b10000:b<={{16{Qb[31]}},Qb[31:16]};
		5'b10001:b<={{17{Qb[31]}},Qb[31:17]};
		5'b10010:b<={{18{Qb[31]}},Qb[31:18]};
		5'b10011:b<={{19{Qb[31]}},Qb[31:19]};
		5'b10100:b<={{20{Qb[31]}},Qb[31:20]};
		5'b10101:b<={{21{Qb[31]}},Qb[31:21]};
		5'b10110:b<={{22{Qb[31]}},Qb[31:22]};
		5'b10111:b<={{23{Qb[31]}},Qb[31:23]};
		5'b11000:b<={{24{Qb[31]}},Qb[31:24]};
		5'b11001:b<={{25{Qb[31]}},Qb[31:25]};
		5'b11010:b<={{26{Qb[31]}},Qb[31:26]};
		5'b11011:b<={{27{Qb[31]}},Qb[31:27]};
		5'b11100:b<={{28{Qb[31]}},Qb[31:28]};
		5'b11101:b<={{29{Qb[31]}},Qb[31:29]};
		5'b11110:b<={{30{Qb[31]}},Qb[31:30]};
		5'b11111:b<={{31{Qb[31]}},Qb[31]};
		default:b<=Qb;
	endcase
end
assign a=Qb>>sa;
```

```
assign c=Qb<<sa;
function [31:0]select;
input [31:0] a,b,c;
input [1:0] Sc;
case(Sc)
      2'b00:select=c;                 //左移
      2'b01:select=a;                 //逻辑右移
      2'b10:select=b;                 //算术右移
endcase
endfunction
assign Sout=select(a,b,c,Sc);
endmodule
```

（11）支持 lui 的功能部件

```
`timescale 1ns/1ps
module IMMELEFT(immediate,immediate_Left);          //实现设置高位功能
input [15:0] immediate;
output [31:0] immediate_Left;
assign immediate_Left=immediate<<16;        //左移 16 位
endmodule
```

（12）IF/ID 寄存器组

```
`timescale 1ns/1ps
module IF_ID(We,PC_4,IR,Condep,PCout,IRout,Clk);
input We;
input Clk;
input [31:0] PC_4;      //输入PC+4 的值
input [31:0] IR;
input Condep;
output reg [31:0] PCout;
output reg [31:0] IRout;
always@(posedge Clk)
begin
if(Condep==0)                   //实现清零功能
   begin
      PCout <= 0;
      IRout <= 0;
   end
else
```

```
  begin
    if(We)
      begin
      PCout <= PC_4;
      IRout <= IR;
      end
    else
      begin                    //实现暂停功能
      PCout <= PCout;
      IRout <= IRout;
      end
  end
end
endmodule
```

（13）ID/EX 寄存器组

```
`timescale 1ns/1ps
module
ID_EX(Wreg,Reg2reg,Wmem,Aluc,Aluqb,FwdA,FwdB,Op,ID_PC,Qa,Qb,extendImmediate,ID_
Rd,E_Wreg,E_Reg2reg,E_Wmem,E_Aluc,E_Aluqb,E_FwdA,E_FwdB,E_Op,E_PC,E_Qa,E_Qb,
E_extendImmediate,E_Rd,Clk,jaddr,E_jaddr,STALL,Condeq,Func,E_func,Sout,E_sout,luiex,E_lu
iex,Jal,E_jal,PCout,E_PCout);
input Wreg,Wmem,Aluqb,STALL,Condeq,Jal;
input [1:0] Reg2reg;
input [2:0] Aluc,FwdA,FwdB;
input [31:0] ID_PC,jaddr,Sout,PCout;
input [5:0] Op,Func;
input [31:0] Qa,Qb,extendImmediate,luiex;
input [4:0] ID_Rd;
input Clk;
output reg E_Wreg,E_Wmem,E_Aluqb,E_jal;
output reg [1:0] E_Reg2reg;
output reg [2:0] E_Aluc,E_FwdA,E_FwdB;
output reg [5:0] E_Op,E_func;
output reg [31:0] E_PC,E_jaddr;
output reg [31:0] E_Qa,E_Qb,E_extendImmediate,E_sout,E_luiex,E_PCout;
output reg [4:0] E_Rd;
always@(posedge Clk)
begin
if(Condeq==0)                //实现清零功能
begin
```

```
E_Wreg <= 0;
E_Reg2reg <= 0;
E_Wmem <= 0;
E_Aluc <= 0;
E_Aluqb <= 0;
E_FwdA <= 0;
E_FwdB <= 0;
E_PC <= 0;
E_Op <= 0;
E_Qa <= 0;
E_Qb <= 0;
E_extendImmediate <= 0;
E_Rd <= 0;
E_jaddr <= 0;
E_func <= 0;
E_sout <= 0;
E_luiex <= 0;
E_jal <= 0;
E_PCout <= 0;
end
else
begin
      if(STALL)
        begin
          E_Wreg <= Wreg;
          E_Reg2reg <= Reg2reg;
          E_Wmem <= Wmem;
          E_Aluc <= Aluc;
          E_Aluqb <= Aluqb;
          E_FwdA <= FwdA;
          E_FwdB <= FwdB;
          E_PC <= ID_PC;
          E_Op <= Op;
          E_Qa <= Qa;
          E_Qb <= Qb;
          E_extendImmediate <= extendImmediate;
          E_Rd <= ID_Rd;
          E_jaddr <= jaddr;
          E_func <= Func;
          E_sout <= Sout;
          E_luiex <= luiex;
          E_jal <= Jal;
          E_PCout <= PCout;
        end
```

```
        else
            begin
                E_Wreg <= 0;
                E_Reg2reg <= 0;
                E_Wmem <= 0;
                E_Aluc <= 0;
                E_Aluqb <= 0;
                E_FwdA <= 0;
                E_FwdB <= 0;
                E_PC <= 0;
                E_Op <= 0;
                E_Qa <= 0;
                E_Qb <= 0;
                E_extendImmediate <= 0;
                E_Rd <= 0;
                E_jaddr <= 0;
                E_func <= 0;
                E_sout <= 0;
                E_luiex <= 0;
                E_jal <= 0;
                E_PCout <= 0;
            end
end
end
endmodule
```

（14）EX/MEM 寄存器组

```
`timescale 1ns/1ps
module
EX_MEM(E_Wreg,E_Reg2reg,E_Wmem,E_Z,E_R,E_Din,E_Rd,M_Wreg,M_Reg2reg,M_Wme
m,M_Z,M_R,M_Din,M_Rd,Clk,E_sout,M_sout,E_luiex,M_luiex,E_jal,M_jal,E_PCout,M_PCout
);
input E_Wreg,E_Wmem,E_Z,Clk,E_jal;
input [1:0] E_Reg2reg;
input [31:0] E_R,E_Din,E_sout,E_luiex,E_PCout;
input [4:0] E_Rd;
output reg M_Wreg,M_Wmem,M_Z,M_jal;
output reg [1:0] M_Reg2reg;
output reg [31:0] M_R,M_Din,M_sout,M_luiex,M_PCout;
output reg [4:0] M_Rd;
always@(posedge Clk)
begin
```

```
M_Wreg <= E_Wreg;
M_Reg2reg <= E_Reg2reg;
M_Wmem <= E_Wmem;
M_Z <= E_Z;
M_R <= E_R;
M_Din <= E_Din;
M_Rd <= E_Rd;
M_sout <= E_sout;
M_luiex <= E_luiex;
M_jal <= E_jal;
M_PCout <= E_PCout;
end
endmodule
```

（15）MEM/WB 寄存器组

```
`timescale 1ns/1ps
module
MEM_WB(Clk,M_Wreg,M_Reg2reg,M_R,M_Dout,M_Rd,W_Wreg,W_Reg2reg,W_R,W_Dout,
W_Rd,M_sout,W_sout,M_luiex,W_luiex,M_jal,W_jal,M_PCout,W_PCout);
input Clk,M_Wreg,M_jal;
input [1:0] M_Reg2reg;
input [31:0] M_R,M_Dout,M_sout,M_luiex,M_PCout;
input [4:0] M_Rd;
output reg W_Wreg,W_jal;
output reg [1:0] W_Reg2reg;
output reg [31:0] W_R,W_Dout,W_sout,W_luiex,W_PCout;
output reg [4:0] W_Rd;
always @(posedge Clk)
begin
W_Wreg <= M_Wreg;
W_Reg2reg <= M_Reg2reg;
W_R <= M_R;
W_Dout <= M_Dout;
W_Rd <= M_Rd;
W_sout <= M_sout;
W_luiex <= M_luiex;
W_jal <= M_jal;
W_PCout <= M_PCout;
end
endmodule
```

（16）加法器

（实现PC+4 功能的加法器）

```
`timescale 1ns/1ps
module PC4(Addr,PC_4);
input [31:0] Addr;
output [31:0] PC_4;
assign PC_4 = Addr + 4;
endmodule
```

（实现beq、bne指令跳转地址的加法器）

```
`timescale 1ns/1ps
module Addr_immediate(extendImmediate,PC_4,ID_PC);
input [31:0] PC_4,extendImmediate;
output [31:0] ID_PC;
wire [31:0] BB;
assign BB=extendImmediate<<2;
assign ID_PC = PC_4+BB;            //PC+4 与立即数左移两位的值相加
endmodule
```

（17）J 型指令的跳转地址

```
`timescale 1ns/1ps
module J_addr(address,PC_4,jaddr);
input [25:0] address;
input [31:0] PC_4;
output [31:0] jaddr;
assign jaddr={PC_4[31:28],address,2'b00};
endmodule        //PC+4 的最高四位与address左移两位的值进行拼接
```

（二）封装代码

```
`timescale 1ns/1ps
module FlowCpu(Clk,Op,rs,rt,rd,immediate,Inst,Qa,Qb,D,Dout,Func,Addr,E_R,Clrn);
input Clk,Clrn;
output [5:0] Op;
output [4:0] rs;
output [4:0] rt;
output [4:0] rd;
output [15:0] immediate;
```

```
output [31:0] Inst;
output [31:0] Qa;
output [31:0] Qb;
output [31:0] D;
output [31:0] Dout;
output [5:0] Func;
output [31:0] Addr;
output [31:0] E_R;
wire Condeq,STALL,Jal,E_jal,M_jal,W_jal;
wire [4:0] sa;
wire [1:0] Sc,W_Reg2reg,M_Reg2reg,E_Reg2reg,Reg2reg;
wire [2:0] Aluc,E_Aluc,FwdA,FwdB,E_FwdA,E_FwdB;
wire [31:0] Y,newAddr;
wire [31:0]
PC_4,E_PC,extendImmediate,Addr_immediate,PC_addr,PCout,IRout,E_extendImmediate,Sout,E
_sout,M_sout,W_sout,ss,E_PCout,M_PCout,W_PCout;
wire [4:0] ID_Rd,E_Rd,W_Rd,M_Rd;
wire [25:0] address;
wire [31:0]
ID_PC,E_Qa,E_Qb,M_R,W_R,X,Y_B,M_Din,W_Dout,jaddr,E_jaddr,luiex,E_luiex,M_luiex,W_l
uiex;
wire [1:0] Pcsrc,FwdC;
wire E_Z,M_Z,Regrt,Se,Wreg,Aluqb,Wmem,W_Wreg,E_Wmem,M_Wmem,E_Aluqb;
wire E_Wreg,M_Wreg;
wire [5:0] E_op,E_func;
MUX4X32 mux4x32(PC_4,E_PC, Pcsrc,newAddr,E_jaddr,E_Qa);
PC pc(Clk,newAddr,Addr,STALL);
INSTMEM instmem(Addr, Inst);
PC4 pc4(Addr,PC_4);
IF_ID if_id(STALL,PC_4,Inst,Condeq,PCout,IRout,Clk);
DIVIDE divide(IRout,Op,rs,rt,rd,immediate,Func,address,sa);
MUX2X5 mux2x5(rt,rd,Regrt,ID_Rd);
EXT16T32 ext16t32(Se,immediate,extendImmediate);
IMMELEFT immeleft(immediate,luiex);
CONUNIT
conunit(Op,Func,E_Z,Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Pcsrc,Reg2reg,STALL,Condeq,E_Reg2
reg,E_Wreg,M_Wreg,FwdA,FwdB,M_Rd,E_Rd,E_op,rs,rt,E_func,Sc,FwdC,Jal);
REGFILE regfile(W_Wreg,rs,rt,W_Rd,Clk,D,Qa,Qb,W_jal,W_PCout,Clrn);
MUX4X32_FwdC fwdc(FwdC,Qb,E_sout,M_sout,ss);
SHIFT shift(ss,sa,Sc,Sout);
Addr_immediate addr_immediate(extendImmediate,PCout,ID_PC);
J_addr j_address(address,PCout,jaddr);
ID_EX
id_ex(Wreg,Reg2reg,Wmem,Aluc,Aluqb,FwdA,FwdB,Op,ID_PC,Qa,Qb,extendImmediate,ID_Rd
```

```
,E_Wreg,E_Reg2reg,E_Wmem,E_Aluc,E_Aluqb,E_FwdA,E_FwdB,E_op,E_PC,E_Qa,E_Qb,E_e
xtendImmediate,E_Rd,Clk,jaddr,E_jaddr,STALL,Condeq,Func,E_func,Sout,E_sout,luiex,E_luiex,
Jal,E_jal,PCout,E_PCout);
MUX4X32_FwdA fwda(E_Qa,D,M_R,E_FwdA,X,M_sout,M_luiex);
MUX4X32_FwdB fwdb(E_Qb,D,M_R,E_FwdB,Y_B,M_sout,M_luiex);
MUX2X32_1 mux2x32_1(Y_B,E_extendImmediate,E_Aluqb,Y);
ALU alu(X,Y,E_Aluc,E_R,E_Z);
EX_MEM
ex_mem(E_Wreg,E_Reg2reg,E_Wmem,E_Z,E_R,Y_B,E_Rd,M_Wreg,M_Reg2reg,M_Wmem,M
_Z,M_R,M_Din,M_Rd,Clk,E_sout,M_sout,E_luiex,M_luiex,E_jal,M_jal,E_PCout,M_PCout);
DATAMEM datamem(M_Wmem,M_R,M_Din,Dout,Clk);
MEM_WB
mem_wb(Clk,M_Wreg,M_Reg2reg,M_R,Dout,M_Rd,W_Wreg,W_Reg2reg,W_R,W_Dout,W_R
d,M_sout,W_sout,M_luiex,W_luiex,M_jal,W_jal,M_PCout,W_PCout);
MUX2X32_2 mux2x32_2(W_R,W_Dout,W_Reg2reg,D,W_sout,W_luiex);
endmodule
```

（三）仿真代码

```
`timescale 1ns/1ps
module FlowCpu_tb(Op,rs,rt,rd,immediate,Inst,Qa,Qb,D,Addr,Func,Dout,E_R);
reg Clk,Clrn;
output [5:0] Op;
output [4:0] rs;
output [4:0] rt;
output [4:0] rd;
output [15:0] immediate;
output [31:0] Inst;
output [31:0] Qa;
output [31:0] Qb;
output [31:0] D;
output [31:0] Addr;
output [5:0] Func;
output [31:0] Dout;
output [31:0] E_R;
wire [5:0] Op;
wire [4:0] rs;
wire [4:0] rt;
wire [4:0] rd;
wire [15:0] immediate;
wire [31:0] Inst;
wire [31:0] Qa;
```

```
wire [31:0] Qb;
wire [31:0] D;
wire [31:0] Dout;
wire [5:0] Func;
wire [31:0] Addr;
wire [31:0] E_R;
FlowCpu mm(
     .Clk(Clk),
      .Op(Op),
      .rs(rs),
      .rt(rt),
      .rd(rd),
      .immediate(immediate),
            .Inst(Inst),
      .Qa(Qa),
      .Qb(Qb),
      .D(D),
      .Dout(Dout),
      .Func(Func),
      .Addr(Addr),
      .E_R(E_R),
      .Clrn(Clrn)
);
initial begin
Clk=0;
Clrn=1;
#50;
     Clk=!Clk;
     Clrn=1;
forever #50 begin
   Clk=!Clk;
   Clrn=1;        //前 50ns是一条指令，之后每 100ns是一条指令、一个时钟周期
end
end
endmodule
```

四、仿真结果

（一）仿真结果

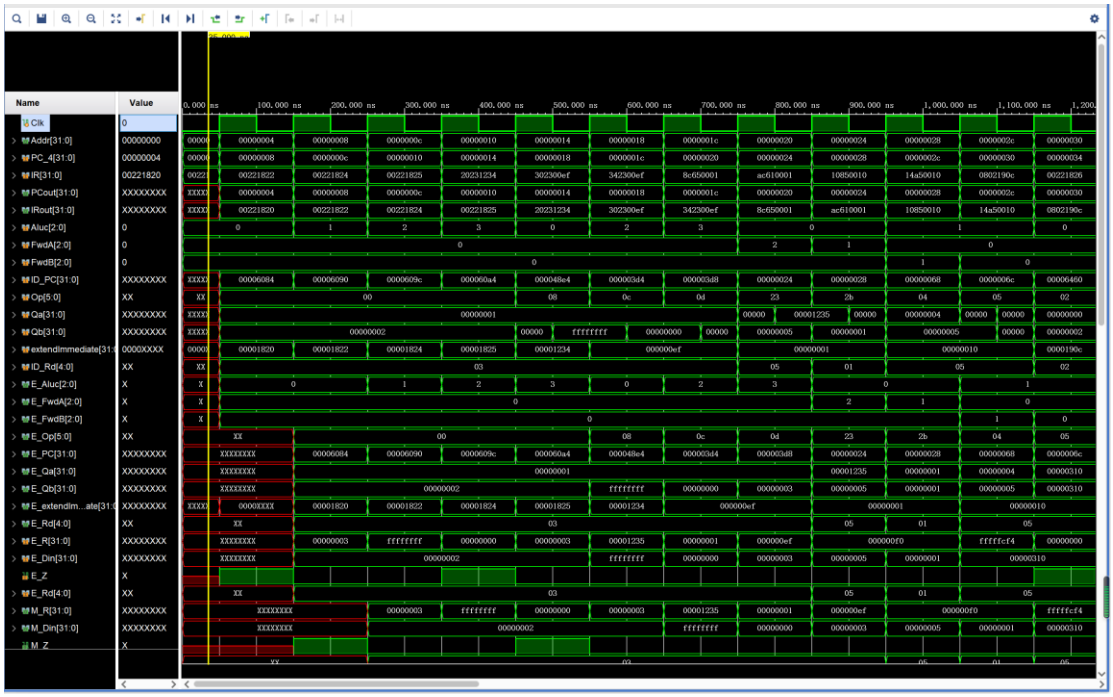

图 28 实际仿真波形（第一段）

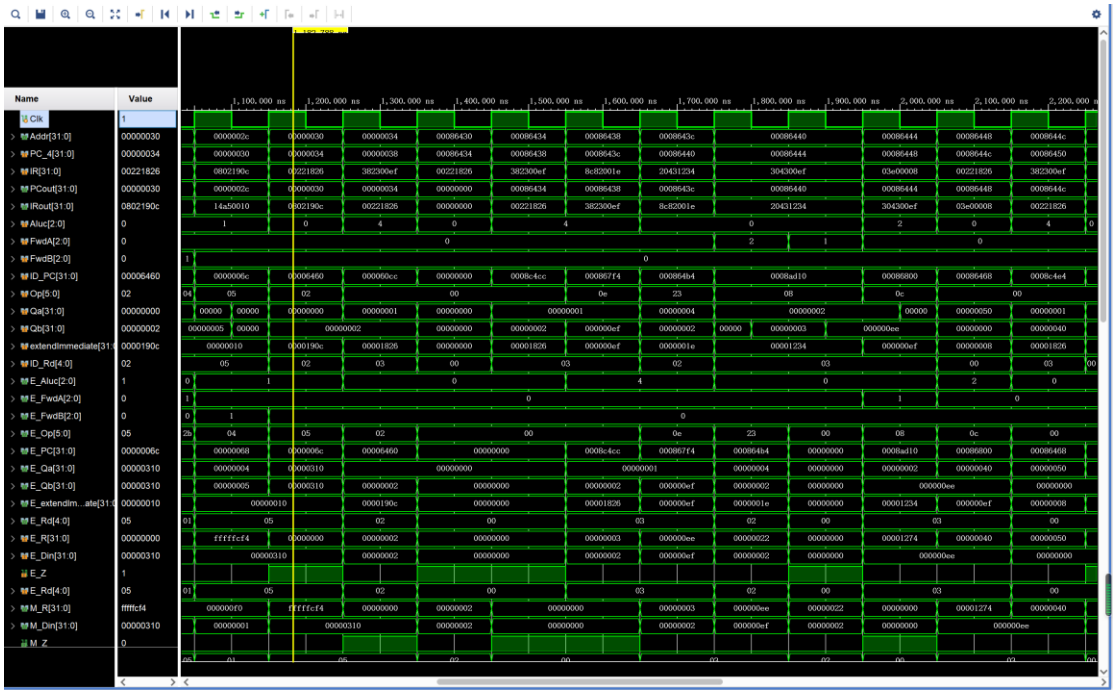

图 29 实际仿真波形（第二段）

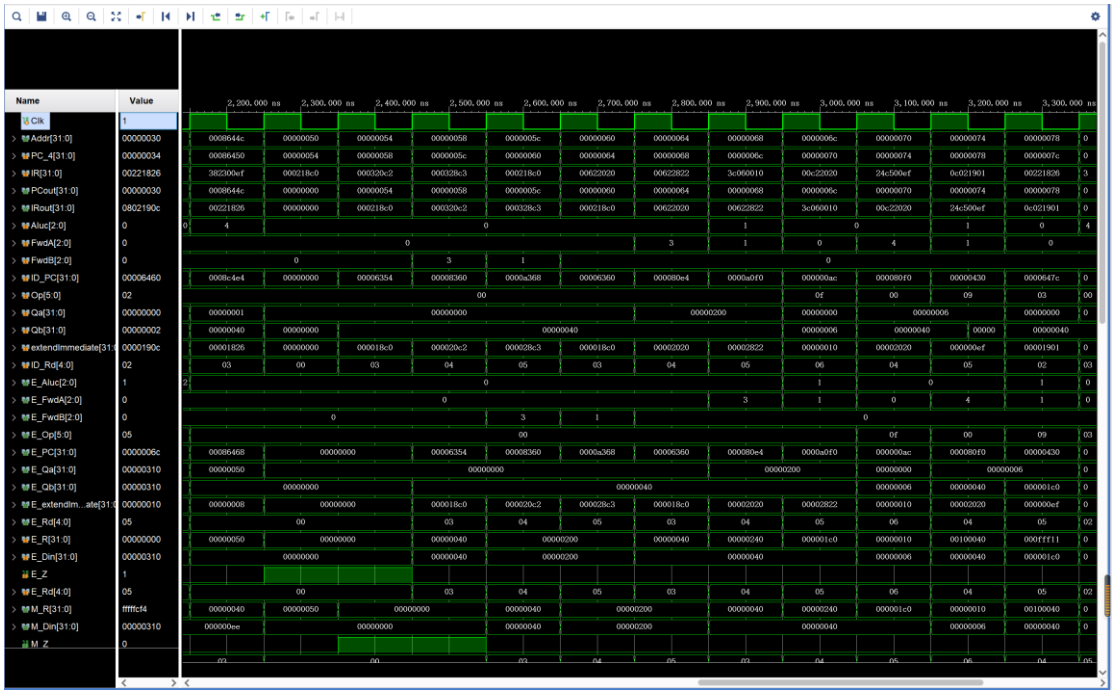

图 30 实际仿真波形（第三段）

图中 R 表示每条指令在 ALU 中计算的结果，D 表示每个时钟周期结束时写回寄存器堆的值。

（二）分析

选取完成的几个内容分别分析：

（1）数据前推的实现

以设计的指令集中第 7 条指令 ori：

32'b001101_00001_00011_0000000011101111

和第 8 条指令 lw：

32'b100011_00011_00101_0000000000000001

为例，分析数据冒险的解决。

ori 指令：将 1 号寄存器的值 1 与立即数 0xef 进行按位或操作，运行结果存入 3 号寄存器中；lw 指令：将 3 号寄存器的值与立即数 1 相加得到数据存储器访存地址，按该地址取数存入目的寄存器即 5 号寄存器。两条指令间存在数据冒险。

通过内部前推可以解决此数据冒险：在 940ns 时，ori 指令位于 MEM 级，lw 指令位于 EX 级，MEM 级 M_R 推到的 EX 级 ALU 的 X 端口，使得 lw 指令所用到 3 号寄存器的值是此前 ori 指令的运算结果。

（2）lw 数据冒险的解决

以设计的指令集中第 15 条指令 lw：

32'b100011_00100_00010_0000000000011110;

和第 16 条指令 addi：

32'b001000_00010_00011_0001001000110100;

为例，分析 lw 数据冒险的解决。

lw 指令：将 4 号寄存器的值与立即数 30 相加得到数据存储器访存地址，按该地址取数存入目的寄存器即 2 号寄存器；addi 指令：将 2 号寄存器的值与立即数 4660 相加得到运算结果，并将结果存入 3 号寄存器。两条指令间存在 lw 指令的数据冒险。

通过暂停、内部前推可以解决此数据冒险：在 1840ns 时，lw 指令位于 EX 级，addi 指令位于 ID 级;在 1940ns 时，lw 指令位于 MEM 级，addi 指令经暂停位于 ID 级，EX 级无指令（原指令清零）；在 2040ns 时，lw 指令位于 WB 级，addi 指令位于 EX 级，WB 级 D 输出 Dout 值并推到 EX 级 ALU 的 X 端口。

（3）跳转指令控制冒险的解决

以设计的指令集中第 18 条指令 jr：

32'b000000_11111_00000_00000_00000_001000;

为例，控制冒险的解决。

jr 指令：访问$ra 寄存器，取出值作为跳转地址，跳转到第 21 条 sll 指令。

通过清零信号 Condeq 解决 jr 指令的控制冒险。在 2240ns 时 jr 指令位于 EX 级，xor 指令位于 ID 级，xori 指令位于 IF 级；在 2340ns 时，jr 指令位于 MEM 级，EX 级和 ID 级清零，无指令；跳转指令 sll 位于 IF 级，实现跳转，解决控制冒险。

（4）其他指令的冒险解决

除课本所示冒险解决外，我们还解决了 lui 指令的数据冒险；跳转指令 jal、j、jr 指令的控制冒险；移位指令 sll、srl、sra 三者间的数据冒险以及与其他指令间的数据冒险。

（三）结论

因篇幅原因，**分析部分**仅验证了部分指令。但按照如上检测方法，将测试结果对比预测结果，可以验证所有测试结果均符合预测，波形输出正常，流水线 CPU 设计成功。

五、实验结论与体会

（一）实验结论

我们成功将实验一中的单周期处理器拓展成为了 5 级流水线处理器。5 级流水线 CPU 各个必要的模块均功能正常，将各模块封装形成的完整流水线 CPU，也能正常运行。

一共 21 条指令全部能够正确，波形输出正确。成功实现了数据前推，解决了 lw 的数据冒险、跳转分支指令的控制冒险，成功实现流水线暂停、寄存器组清零。

（二）体会

在完成实验的过程中，我们对 5 级流水线 CPU 的运行原理和计算机相关知识更加清楚。

通过认真研究每一条指令的结构和功能，理解每一个模块的设计原理，了解每个接口的作用，我们最终成功完成了 21 条指令的数据通路设计，成功实现了数据前推，解决了 lw 的数据冒险、跳转分支指令的控制冒险，成功实现流水线暂停、寄存器组清零等功能。

该实验课是我们大学以来遇到的较大型的实验课，十分考验我们的动手能力、将理论知识运用实际的能力；它也极大程度训练了我们工程方面的意识：如何分析、评估、解决一个眼前的大问题？如何为要完成的任务做出合理的规划？团队如何达到有效的合作？如何解释理论和实际的差距？等等。在这门课中收获到的绝对不止知识，还有许多必要的能力。

最终成功搭出了 5 级流水线 CPU，这不仅是对我们努力的肯定，也显示出了

对我们课程知识的一次运用。相信通过这门课能学习到更多与计算机相关的有用知识。