

实 验 报 告

课程名	计算机体系结构				
学生姓名		学号		指导老师	
实验地点	实验楼 A205		实验时间	2021 年 12 月 20 日	

一、实验项目名称

课程设计拓展——优化 CPU 系统

二、实验目的

1. 加深对计算机组成原理和体系结构理论知识的理解。
2. 培养对 CPU 设计的兴趣，在理解现有 CPU 架构的基础上，引发对体系结构的思考和创新。
3. 培养创新思维能力，并通过实践验证新想法。

三、必修或选修

必修

四、实验平台

1. 编程与仿真软件：Xilinx Vivado 2019.2
2. 实验箱：LS-CPU-EXB-002 教学系统实验箱一套

五、实验内容及步骤

1 实验内容与结构图

原 5 级流水分析已在之前的实验报告中详细分析过了，本次实验只说明修改内容。根据前期计划，结合实际情况，按照完成的递进顺序，完成的实验内容依次有如下几个部分：

- 1) 将 IF 与 MEM 两个访存阶段的周期降低至 1 周期；
- 2) 将 CPU 从 5 级流水修改为 6 级流水；
- 3) 在 6 级流水上实现静态分支预测；
- 4) 实现 EXE 阶段读寄存器的数据旁路；
- 5) 实现 MEM 阶段读寄存器的数据旁路。

实验中修改的内容及目的总结在表格 1 中。

表格 1 修改内容及目的

序号	修改内容	修改目的
1	一周期访存	降低取值(主要原因)和存取数据的阻塞周期
2	六级流水	简化 ID 阶段；加深流水线，使分支预测有意义
3	静态分支预测	降低控制相关的影响
4	EXE 数据旁路	降低 EXE 阶段数据相关的影响
5	MEM 数据旁路	降低 MEM 阶段数据相关的影响

2 实验原理图

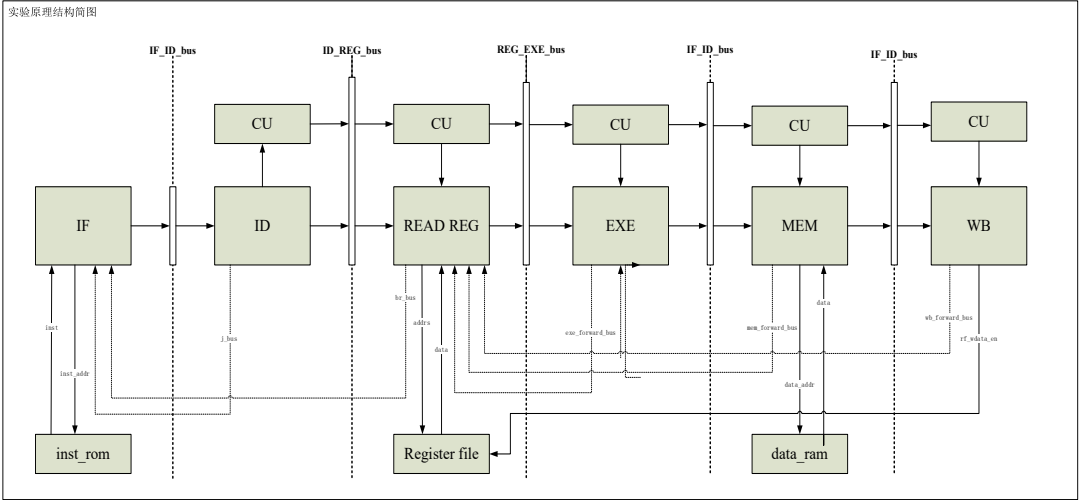


图 1 实验原理结构图

3 存储器：访存周期缩短至 1 周期

3.1 原有程序 BUG

原有程序不能正常运行 BUG 是与存储器访问相关的，在这一节进行说明。

3.1.1 问题描述

经过反复核实，Demo 代码中确实存在 bug 导致指令不能按照预期执行。经过排查，发生的第一个问题是 0x3C 处指令(表格 2)，\$3 寄存器不能成功写入(表格 3)。后面的指令也遇到类似的问题没有正常读写寄存器。实际的指令执行序列为：

0x34 => 0x48 跳转正确 0x84 =>0x60 跳转正确 0x6C => 0x70 跳转正确 0x7C =>0x90 跳转错误 0x00 => 0x30 异常返回错误 0x34 循环重复前面的。

表格 2 3CH 处的指令

指令地址	汇编指令	结果描述	机器指令的机器码	
3CH	addu \$3, \$2,\$1	[\$3]= 0000_0011H	00411821	0000_0000_0100_0001_0001_1000_0010_0001

表格 3 板上\$3 寄存器未成功写入现象

阶段	0x3C 写回完成前	0x3C 写回完成后
截图		
说明	REG03 应为 0	写入完成，REG03 应为 00000011

3.1.2 问题解决

通过仿真，对访存相关代码进行排查。为了便于调试，引出了一些其他数据线的在 testbench 中(图 2)。

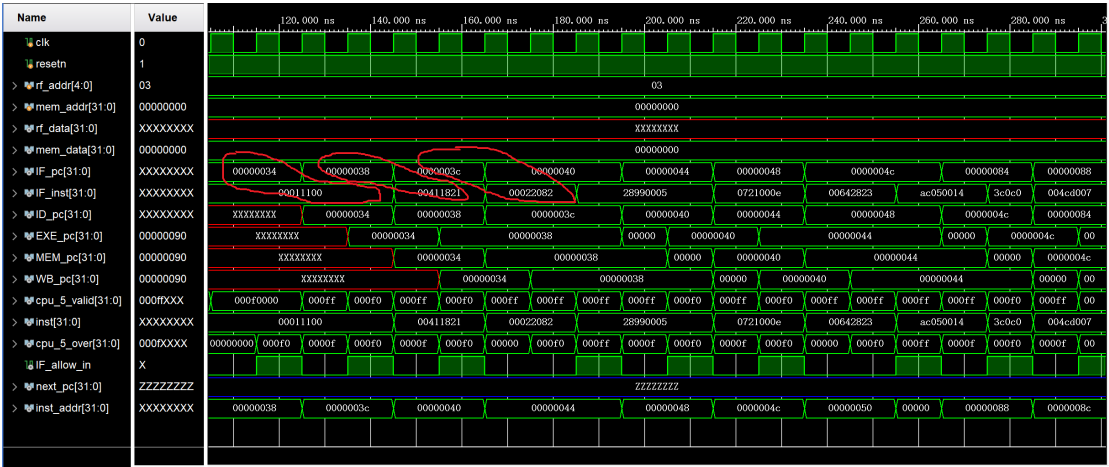


图 2 BUG 代码仿真结果

最终，将问题锁定到 IF 取值上。在**错误!未找到引用源。**中取值结果相比于取值地址落后一个周期，导致一系列错误。具体原因是 INST_ROM 是同步读的，需要一个始终脉冲输入读地址，然后再等一个时钟脉冲才能输出数据。再 DEMO 代码中虽然考虑了同步读的问题，但是没有考虑 PC 寄存器更新也需要一个时钟周期。

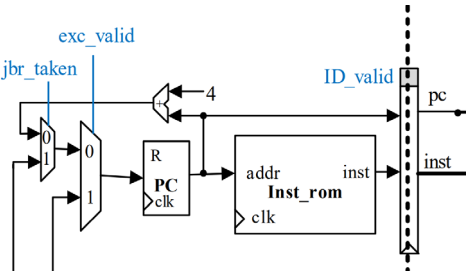


图 3 IF 设计结构

IF 结构如图 3 所示，PC 需要一个时钟周期才能更新为 next_pc 的值，所以第一个周期传入 Inst_rom 的是上一条指令的 PC 值，第二给周期传入的才是这个指令的 PC 值，第三个周期才能得到这一天指令的指令二进制码。

将 IF 结构改为 3 个时钟周期执行则可以解决这个问题，主要是修改 IF_over 的赋值，修改如下表所示。但是更好的解决方法是直接将 next_pc 发送给 inst_rom，只需要两个周期就能实现，但是初始化比较复杂，在下一小节说明。

表格 4 IF 代码修改前后对比

原本代码	修改后代码
<pre>always @(posedge clk) begin if (!resetn next_fetch) begin IF_over <= 1'b0;</pre>	<pre>reg stall1; always @(posedge clk) begin if (!resetn next_fetch) begin</pre>

```
end
else
begin
    IF_over <= IF_valid;
end
end
```

```
IF_over <= 1'b0;
stall1<=1'b0;

end
else if(!stall1)
begin
    stall1<=1'b1;
end
else
begin
    IF_over <= IF_valid;
end
end
```

修改后，代码正常运行，如下图所示。能够正常取值，寄存器能正常读写，程序能正常跳转。

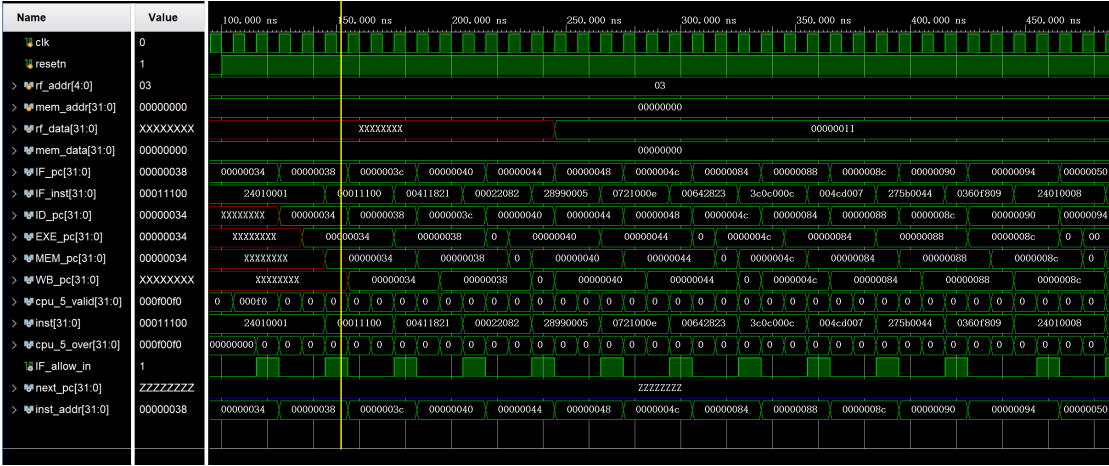


图 4 程序正常运行仿真结果

3.2 一周期访存

在这一小节稍作修改，使得取指级和访存级的 load 不需要多等一拍。为了使同步存储器的访存周期数减少为 1，需要注意一下两点设置。

第一、使用 Xilinx 的 IP 核的 ROM 和 RAM，但是不设置 primitives /core output 两个寄存器，而是由自己的 Verilog 代码进行控制，减少访存周期。这样一个时钟脉冲输入访存地址，下一个脉冲就能将输出从输出数据线保存到 CPU 的流水寄存器中（替代了原输出缓存寄存器）。

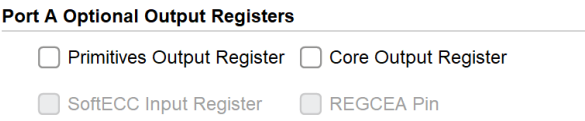


图 5 取消存储器 IP 核的输出缓存寄存器

第二、输入的访存地址也不能从流水寄存器或 PC 寄存器中获取，而是要从更新流水寄

4.1 拆分 ID 为两级

首先需要修改 ID 的输入输出，去除与读寄存器相关的内容。此外修改 ID 阶段的结束条件，去除数据相关的等待。

关键代码(decode.v)：ID 阶段的结束条件
1. assign ID_over = ID_valid & (~inst_jbr IF_over);

同时将需要读的寄存器地址等通过总线传递到下一级，告知下一级是否需要读寄存器。并在新的 REG 阶段则需要把原 ID 中读寄存器的功能迁移到本阶段，即可。

关键代码(decode.v)：ID 阶段将读寄存器延后
1. wire read_rs;
2. wire read_rt;
3. assign read_rs=~(inst_j_link inst_shf_sa);
4. assign read_rt=~(inst_j_link inst_imm_zero inst_imm_sign);
5. assign ID_EXE_bus = {read_rs,read_rt,br_control,br_target, rs,rt,inst_no_r s,inst_no_rt,/*其他保留，略*/}

4.2 条件跳转与静态分支预测

4.2.1 条件跳转

由于读寄存器单独为 ID 级之后的一个流水级，条件分支指令需要在 REG 级中进行条件判断，一些需要从寄存器中获得跳转地址的无条件跳转指令也需要在 REG 级进行处理。

首先需要将原 jbr_bus 分为两条跳转总线 j_bus 和 br_bus，分别表示 ID 阶段可以确定的大部分无条件跳转指令的跳转地址，和 REG 阶段才可以确定的条件跳转和少部分依赖寄存器的无条件跳转。

关键代码 (fetch.v)：下一条指令地址的更新，分别考虑 br_bus 与 j_bus
1. assign next_pc = exc_valid ? exc_pc : // 处理 WB 中的异常
2. br_taken ? br_target : // REG 中执行的是更早的一条指令
3. j_taken ? j_target :
4. seq_pc;

4.2.2 未考虑控制相关导致问题

单纯这样该,如果忘记考虑静态分支预测的话,会造成延迟槽后面的指令多余地被执行。一个意外执行地例子，如图 7 所示，相关代码如下表所示。

指令地址	汇编指令	结果描述	机器指令的机器码	
			16 进制	二进制
48H	bgez \$25,#14	跳转到 84H	0721000E	0000_0111_0010_0001_0000_0000_0000_1110
4CH	subu \$5, \$3,\$4	[\$5] = 0000_000DH	00642823	0000_0000_0110_0100_0010_1000_0010_0011

50H	sw \$5, #20(\$0)	Mem[0000_0014H] = 0000_000DH	AC050014	1010_1100_0000_0101_0000_0000_0001_0100
-----	------------------	---------------------------------	----------	---

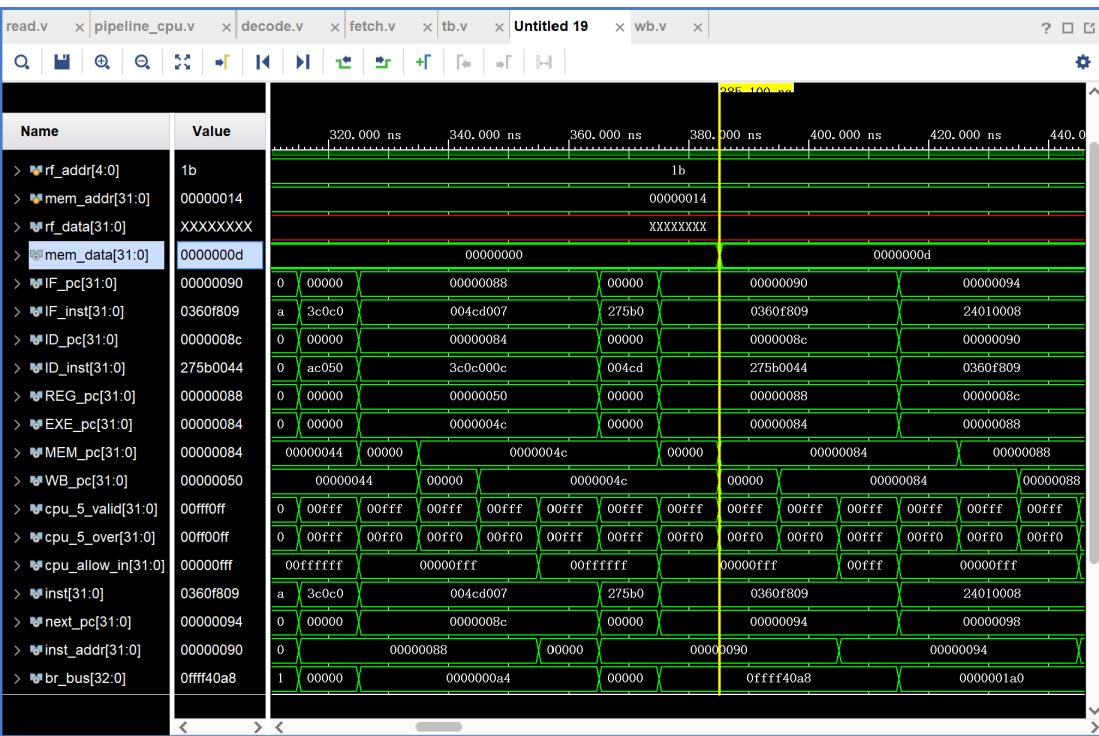


图 7 指令 48H 跳转但未取消 50H 指令的执行，导致内存 0x14 意外地被写入

4.2.3 静态分支预测

由于条件跳转指令被迫延后，一条延迟槽已经不能抵消控制相关的代价了，需要引入分支预测机制。跳转地址有些可以在 ID 获得，如 beq；有些不能在 ID 段获得跳转地址，这也包括一些无条件跳转指令，如 jar。因此采取静态预测不跳转的方式处理起来比较统一。

关键的一点就是要在分支预测失败时，及时取消预测执行的指令。从实现上讲，就是在 REG 阶段获知预测失败后，禁止 IF_ID_bus_r 流水线寄存器的更新。这样，就能避免预测指令的意外执行。

关键代码 (read.v)：获知条件分支预测失败
1. assign predict_fail=br_taken;

关键代码 (pipeline_cpu.v)：禁止 IF_ID_bus_r 流水线寄存器的更新
1. always @(posedge clk)
2. begin
3. if(IF_over && ID_allow_in && ~predict_fail)
4. begin
5. IF_ID_bus_r <= IF_ID_bus;
6. end
7. end

5 数据旁路

在降低存储器访存周期后，数据相关成为造成阻塞、影响性能的最大影响因素。可以添加两条数据旁路，减少数据相关造成的阻塞。第一条、从 WB、MEM、EXE 到 EXE 的数据旁路，可以减少大部分指令数据相关；第二条、从 WB、MEM 到 MEM 的数据旁路，主要针对 store 类指令。

需要说明的一点是，WB 也需要加旁路。示例实现的 5 级流水中，WB 阶段并不是在阶段开始时就进行寄存器的写入，而是在 WB 阶段结束时才进行写入。这与理论课上学的一些结构是不同的。

5.1 获得各阶段预计写回结果

首先需要分别从 WB、MEM、EXE 阶段引出数据线，包括：写入寄存器地址(`xxx_wdest`)，写入寄存器数据(`xxx_wvalue`)，当前数据是否是最终写入寄存器的数据(`xxx_wvalid`)。这样在 EXE 旁路中，就能对数据相关是否实验旁路进行判断处理。

各模块引出的数据中 `xxx_wvalid` 的考虑较为复杂，需要单独详细说明。

关键代码 (`exe.v`, `mem.v`, `wb.v`) : `wvalid` 的考虑

```
1.    assign EXE_wvalid = ~(|mem_control) // store、load 指令 exe 中不是写回结果
2.                                     & EXE_over & rf_wen
3.                                     & ~mfc0; //CP0 寄存器在 WB 中，exe 中非写回结果
4.    assign MEM_wvalid = rf_wen & MEM_over
5.                                     & ~mfc0; //CP0 寄存器在 WB 中，exe 中非写回结果
6.    assign WB_wvalid  = rf_wen;
```

5.2 EXE 数据旁路

EXE 数据旁路的处理在 REG 阶段进行处理，首先判断数据相关，其次考虑数据旁路是否能消除相关，最后考虑阻塞流水线。

关键代码 (`read.v`) : 数据相关判断

```
7.    assign rs_wait = ~inst_no_rs
8.                                     & (rs!=5'd0)
9.                                     & ( (rs==EXE_wdest && ~EXE_wvalid)
10.                                     | (rs==MEM_wdest && ~MEM_wvalid)
11.                                     | (rs==WB_wdest && ~WB_wvalid));
12.    assign rt_wait = ~inst_no_rt
13.                                     & (rt!=5'd0)
14.                                     & ( (rt==EXE_wdest && ~EXE_wvalid)
15.                                     | (rt==MEM_wdest && ~MEM_wvalid)
16.                                     | (rt==WB_wdest && ~MEM_wvalid))
17.                                     & ~store_read_rf; //可以使用 MEM 旁路则不阻塞
```


关键代码 (read.v) : 使用数据旁路	
1.	assign forward_rs_value = (rs == EXE_wdest) ? EXE_wvalue:
2.	(rs==MEM_wdest) ? MEM_wvalue:
3.	(rs==WB_wdest) ? WB_wvalue:
4.	rs_value;
5.	assign forward_rt_value = (rt == EXE_wdest) ? EXE_wvalue:
6.	(rt ==MEM_wdest) ? MEM_wvalue:
7.	(rt ==WB_wdest) ? WB_wvalue:
8.	rt_value;

5.3 MEM 数据旁路

MEM 数据旁路类似，但是需要在 EXE 旁路中考虑到 MEM 旁路的存在，避免意外阻塞本可以使用旁路的指令。MEM 旁路在 EXE 阶段处理考虑 store_data 的数据相关。

关键代码 (read.v) : 在 EXE 旁路中考虑到 MEM 旁路的存在	
1.	assign store_read_rf = ~inst_no_rt & (rt!=5'd0)& mem_control[2];
2.	assign rs_wait = /*略*/ & ~store_read_rf; //可以使用 MEM 旁路则不阻塞
3.	assign store_rf_addr = rt & {5{store_read_rf}};

关键代码 (exe.v) : 使用 MEM 旁路	
1.	assign wait_store_data= (store_rf_addr)
2.	&((store_rf_addr==MEM_wdest & ~MEM_wvalid)
3.	(store_rf_addr==WB_wdest & WB_wvalid));
4.	assign forward_store_data = (store_rf_addr==MEM_wdest) ? MEM_wvalue:
5.	(store_rf_addr==WB_wdest) ? WB_wvalue:
6.	store_data;

6 实验仿真结果

程序运行正确性已经经过对比，不再详述。从仿真结果来看，除乘法指令外，由于 1 周期访存的改进，数据旁路的添加，流水线阻塞情况大大减少，几乎是一个周期一条指令。

> IF_pc[31:0]	00000088	00000034	00000038	0000003c	00000040	00000044	00000048	0000004c	00000050	00000084	00000088
> IF_inst[31:0]	004cd007	24010001	09011100	00411821	00022082	28990005	0721000e	00642823	ae050014	3c0c000c	004cd007
> ID_pc[31:0]	00000084	00000000	00000034	00000038	0000003c	00000040	00000044	00000048	0000004c		00000084
> ID_inst[31:0]	3c0c000c	00000000	24010001	09011100	00411821	00022082	28990005	0721000e	00642823		3c0c000c
> REG_pc[31:0]	0000004c	00000000	00000034	00000038	0000003c	00000040	00000044	00000048		0000004c	
> EXE_pc[31:0]	0000004c	00000000	00000034	00000038	0000003c	00000040	00000044	00000048	00000044	00000048	0000004c
> MEM_pc[31:0]	00000048	00000000	00000034	00000038	0000003c	00000040	00000044	00000048	00000044	00000048	0000004c
> WB_pc[31:0]	00000044	00000000	00000034	00000038	0000003c	00000040	00000044	00000048	00000044	00000048	0000004c
> cpu_5_valid[31:0]	00ffff	00f00000	00ff0000	00fff000	00ffff00	00ffff00			00ffffff		
> cpu_5_over[31:0]	00ffff	00f00000	00ff0000	00fff000	00ffff00	00ffff00			00ffffff		

图 8 最终仿真结果节选，阻塞大大减少

7 实验箱验证结果

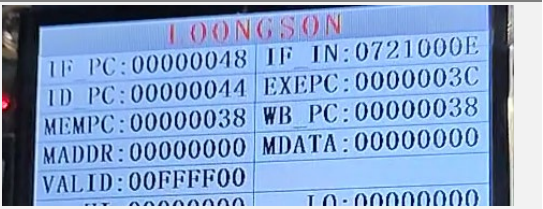
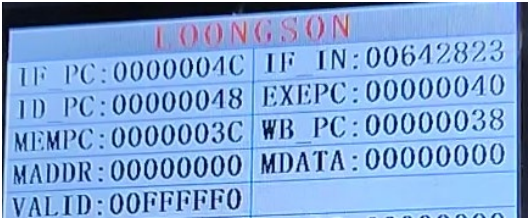
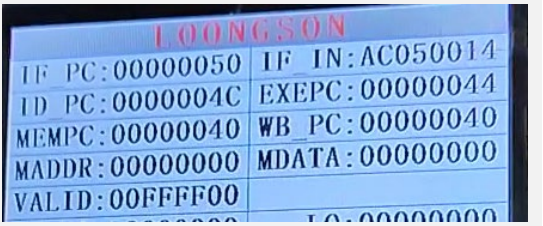
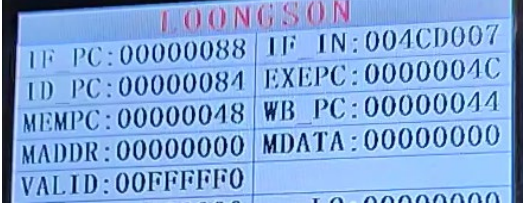
几类指令的执行过程的相关内容在上一次实验已详细说明，不再一一说明分析。这里针对优化部分详细分析指令执行过程。数据旁路已在仿真结果中体现，且不好在实验箱截图中看出数据旁路的影响，这里以静态分支预测为例，分析指令的执行过程。

这里分析 0x48 指令的执行。

表格 6 48H 相关代码节选

指令地址	汇编指令	结果描述	机器指令的机器码	
			16 进制	二进制
44H	slti \$25,\$4,#5	[\$25] = 0000_0001H	44H	slti \$25,\$4,#5
48H	bgez \$25,#14	跳转到 84H	48H	bgez \$25,#14

表格 7 48H 指令执行过程分析

阶段	说明	截图
IF	跳转指令取指	
ID	跳转指令译码，执行分支延迟槽指令	
REG	跳转指令判断，进行静态分支预测 (预测不跳转)	
跳转	预测失败，成功跳转 (由于按键抖动，图中是连续执行两个周期后的结果)	

六、关键问题讨论

修改思想、实现细节、调试过程在上一节已经详细说明了，这里主要说明未实现的其他优化方向的想法。

存储器访问周期、数据旁路是改进最大的两个修改。在这之外，程序的瓶颈在于乘法器的低效。原计划将乘法器独立出来，类型下面这样。

IF	ID	REG	ALU	MEM	WB
			MUL		

但是，目前实现的指令集中，并没有浮点计算、除法等复杂的模块，实现类似的乱序执行工程量太大，也没有必要。此外，在原设计中 HI 和 LO 两个寄存器在 WB 模块中，并且在 MULT 指令执行之后，往往紧跟 mflo 和 mfhi 指令。因此，即便把乘法器像乱序执行一样独立出来，mflo 和 mfhi 指令由于要读 HI 和 LO 寄存器，也会使原流水线继续阻塞。

另一种优化乘法器的方法是将乘法器流水化，但是这样对于大多数情况，未必比现在的迭代乘法器好。迭代乘法器对于较小的整数，其阻塞周期是较少的，但是流水化乘法器的流水周期却是固定的，如果流水级过深反而不如简单迭代乘法器。

简单可行的优化是让乘法器一周期多迭代几位。

七、总结

通过对简单 5 级流水的修改，更清晰地理解了示例代码的程序框架，更好地理解了性能优化方法，实现了一些对性能的改进，如数据旁路。碍于时间，未能对乱序执行、寄存器重命名等高级方法进行实践。

附录 A：涉及修改的完整源代码

A.1 pipeline_cpu.v

完整代码

```
1.    `timescale 1ns / 1ps
2.
3.    module pipeline_cpu( // 多周期 cpu
4.        input clk,      // 时钟
5.        input resetn,    // 复位信号，低电平有效
6.
7.        //display data
8.        input  [4:0] rf_addr,
9.        input  [31:0] mem_addr,
10.       output [31:0] rf_data,
11.       output [31:0] mem_data,
12.       output [31:0] IF_pc,
13.       output [31:0] IF_inst,
14.       output [31:0] ID_pc,
15.       output [31:0] REG_pc,
16.       output [31:0] EXE_pc,
17.       output [31:0] MEM_pc,
18.       output [31:0] WB_pc,
19.
20.       //5 级流水新增
21.       output [31:0] cpu_5_valid,
22.       output [31:0] HI_data,
23.       output [31:0] LO_data,
24.
25.       //调试线
26.       output [31:0] dbg_inst,
27.       output [31:0] dbg_ID_inst,
28.       output [31:0] cpu_5_over,
29.       output [31:0] cpu_allow_in,
30.       output [31:0] dbg_inst_addr,
31.       output [31:0] dbg_next_pc,
32.       output [32:0] dbg_j_bus,
33.       output [32:0] dbg_br_bus,
34.       output [4:0] dbg_rs,
35.       output [4:0] dbg_rt,
36.       output [31:0] dbg_rs_value,
37.       output [31:0] dbg_rt_value,
38.       output [4:0] dbg_EXE_wdest,
```

```

39.         output [4:0] dbg_MEM_wdest,
40.         output [4:0] dbg_WB_wdest,
41.         output dbg_EXE_wvalid,
42.         output dbg_MEM_wvalid,
43.         output dbg_WB_wvalid,
44.         output [31:0] dbg_EXE_wvalue,
45.         output [31:0] dbg_MEM_wvalue,
46.         output [31:0] dbg_WB_wvalue
47.     );
48.     //-----{5级流水控制信号}begin-----
        //
49.     //5 模块的 valid 信号
50.     reg IF_valid;
51.     reg ID_valid;
52.     reg REG_valid;
53.     reg EXE_valid;
54.     reg MEM_valid;
55.     reg WB_valid;
56.     //5 模块执行完成信号,来自各模块的输出
57.     wire IF_over;
58.     wire ID_over;
59.     wire REG_over;
60.     wire EXE_over;
61.     wire MEM_over;
62.     wire WB_over;
63.     //5 模块允许下一级指令进入
64.     wire IF_allow_in;
65.     wire ID_allow_in;
66.     wire REG_allow_in;
67.     wire EXE_allow_in;
68.     wire MEM_allow_in;
69.     wire WB_allow_in;
70.     //6 级流水分支预测
71.     wire predict_fail;
72.
73.     // syscall 和 eret 到达写回级时会发出 cancel 信号,
74.     wire cancel;    // 取消已经取出的正在其他流水级执行的指令
75.
76.     //各级允许进入信号:本级无效,或本级执行完成且下级允许进入
77.     assign IF_allow_in  = (IF_over & ID_allow_in) | cancel;
78.     assign ID_allow_in  = ~ID_valid  | (ID_over  & REG_allow_in);
79.     assign REG_allow_in = ~REG_valid  | (REG_over  & EXE_allow_in);
80.     assign EXE_allow_in = ~EXE_valid | (EXE_over & MEM_allow_in);
81.     assign MEM_allow_in = ~MEM_valid | (MEM_over & WB_allow_in );

```

```

82.         assign WB_allow_in  = ~WB_valid  | WB_over;
83.
84.         //IF_valid, 在复位后, 一直有效
85.         always @(posedge clk)
86.             begin
87.                 if (!resetn)
88.                     begin
89.                         IF_valid <= 1'b0;
90.                     end
91.                 else
92.                     begin
93.                         IF_valid <= 1'b1;
94.                     end
95.             end
96.
97.         //ID_valid
98.         always @(posedge clk)
99.             begin
100.                if (!resetn || cancel)
101.                    begin
102.                        ID_valid <= 1'b0;
103.                    end
104.                else if (ID_allow_in)
105.                    begin
106.                        ID_valid <= IF_over;
107.                    end
108.            end
109.
110.        //REG_valid
111.        always @(posedge clk)
112.            begin
113.                if (!resetn || cancel)
114.                    begin
115.                        REG_valid <= 1'b0;
116.                    end
117.                else if (REG_allow_in)
118.                    begin
119.                        REG_valid <= ID_over;
120.                    end
121.            end
122.
123.        //EXE_valid
124.        always @(posedge clk)
125.            begin

```

```

126.         if (!resetn || cancel)
127.         begin
128.             EXE_valid <= 1'b0;
129.         end
130.         else if (EXE_allow_in)
131.         begin
132.             EXE_valid <= REG_over;
133.         end
134.     end
135.
136.     //MEM_valid
137.     always @(posedge clk)
138.     begin
139.         if (!resetn || cancel)
140.         begin
141.             MEM_valid <= 1'b0;
142.         end
143.         else if (MEM_allow_in)
144.         begin
145.             MEM_valid <= EXE_over;
146.         end
147.     end
148.
149.     //WB_valid
150.     always @(posedge clk)
151.     begin
152.         if (!resetn || cancel)
153.         begin
154.             WB_valid <= 1'b0;
155.         end
156.         else if (WB_allow_in)
157.         begin
158.             WB_valid <= MEM_over;
159.         end
160.     end
161.
162.     //调试信号
163.     assign cpu_allow_in= {8'b0          ,{4{IF_allow_in}},{4{ID_allow_in}}
164.                          ,{4{REG_allow_in}},
165.                          {4{EXE_allow_in}},{4{MEM_allow_in}},{4{WB_allow_
166.                          in}}};
167.
168.     //展示5级的 valid 信号
169.     assign cpu_5_valid = {8'd0          ,{4{IF_valid }},{4{ID_valid}},{4{RE
170.                          G_valid}}},

```

```

167.                {4{EXE_valid}}, {4{MEM_valid}}, {4{WB_valid}}};
168.    //展示 5 级的 over 信号
169.    assign cpu_5_over = {8'd0                , {4{IF_over }}, {4{ID_over}}, {4{REG_o
        ver}}},
170.                {4{EXE_over}}, {4{MEM_over}}, {4{WB_over}}};
171.
172.    //-----{5 级流水控制信号}end-----
        //
173.
174.    //-----{5 级间的总线}begin-----
        //
175.    wire [ 63:0] IF_ID_bus;    // IF->ID 级总线
176.    wire [188:0] ID_REG_bus;   // ID->REG 级总线
177.    wire [171:0] REG_EXE_bus;  // REG->EXE 级总线
178.    wire [153:0] EXE_MEM_bus;  // EXE->MEM 级总线
179.    wire [117:0] MEM_WB_bus;   // MEM->WB 级总线
180.
181.    //锁存以上总线信号
182.    reg [ 63:0] IF_ID_bus_r;
183.    reg [188:0] ID_REG_bus_r;
184.    reg [171:0] REG_EXE_bus_r;
185.    reg [153:0] EXE_MEM_bus_r;
186.    reg [117:0] MEM_WB_bus_r;
187.    initial begin
188.        IF_ID_bus_r={64'b0};
189.        ID_REG_bus_r={177'b0};
190.        REG_EXE_bus_r={172'b0};
191.        EXE_MEM_bus_r={154'b0};
192.        MEM_WB_bus_r={118'b0};
193.    end
194.
195.    //IF 到 ID 的锁存信号
196.    always @(posedge clk)
197.    begin
198.        //      if(IF_over && ID_allow_in)
199.        //MOD 6 级流水预测失败
200.        if(IF_over && ID_allow_in && ~predict_fail)
201.        begin
202.            IF_ID_bus_r <= IF_ID_bus;
203.        end
204.    end
205.    //ID 到 REG 的锁存信号
206.    always @(posedge clk)
207.    begin

```



```

208.         if(ID_over && REG_allow_in)
209.         begin
210.             ID_REG_bus_r <= ID_REG_bus;
211.         end
212.     end
213.     //REG 到 EXE 的锁存信号
214.     always @(posedge clk)
215.     begin
216.         if(REG_over && EXE_allow_in)
217.         begin
218.             REG_EXE_bus_r <= REG_EXE_bus;
219.         end
220.     end
221.     //EXE 到 MEM 的锁存信号
222.     always @(posedge clk)
223.     begin
224.         if(EXE_over && MEM_allow_in)
225.         begin
226.             EXE_MEM_bus_r <= EXE_MEM_bus;
227.         end
228.     end
229.     //MEM 到 WB 的锁存信号
230.     always @(posedge clk)
231.     begin
232.         if(MEM_over && WB_allow_in)
233.         begin
234.             MEM_WB_bus_r <= MEM_WB_bus;
235.         end
236.     end
237.     //-----{5 级间的总线}end-----
        //
238.
239.     //-----{其他交互信号}begin-----
        //
240.     //跳转总线
241.     //    wire [ 32:0] jbr_bus;
242.     //MOD 跳转总线
243.     wire [ 32:0] j_bus;
244.     wire [ 32:0] br_bus;
245.
246.     //IF 与 inst_rom 交互
247.     wire [31:0] inst_addr;
248.     wire [31:0] inst;
249.

```

```

250.    // ID 与 EXE、MEM、WB 交互 (MOD 其实也是旁路线)
251.    wire [ 4:0] EXE_wdest;
252.    wire [ 4:0] MEM_wdest;
253.    wire [ 4:0] WB_wdest;
254.
255.    //MOD 6 级流水旁路数据线
256.    wire [31:0] EXE_wvalue;
257.    wire [31:0] MEM_wvalue;
258.    wire [31:0] WB_wvalue;
259.    wire EXE_wvalid;
260.    wire MEM_wvalid;
261.    wire WB_wvalid;
262.
263.    //MEM 与 data_ram 交互
264.    wire [ 3:0] dm_wen;
265.    wire [31:0] dm_addr;
266.    wire [31:0] dm_wdata;
267.    wire [31:0] dm_rdata;
268.
269.    //ID 与 regfile 交互
270.    wire [ 4:0] rs;
271.    wire [ 4:0] rt;
272.    wire [31:0] rs_value;
273.    wire [31:0] rt_value;
274.
275.    //WB 与 regfile 交互
276.    wire      rf_wen;
277.    wire [ 4:0] rf_wdest;
278.    wire [31:0] rf_wdata;
279.
280.    //WB 与 IF 间的交互信号
281.    wire [32:0] exc_bus;
282.    //-----{其他交互信号}end-----
        //
283.
284.    //-----{各模块实例化}begin-----
        //
285.    wire next_fetch; //即将运行取指模块，需要先锁存 PC 值
286.    //IF 允许进入时，即锁存 PC 值，取下一条指令
287.    assign next_fetch = IF_allow_in;
288.    fetch IF_module(          // 取指级
289.        .clk      (clk      ), // I, 1
290.        .resetn    (resetn    ), // I, 1
291.        .IF_valid  (IF_valid  ), // I, 1

```

```

292.         .next_fetch(next_fetch), // I, 1
293.         .inst      (inst      ), // I, 32
294. //         .jbr_bus  (jbr_bus  ), // I, 33
295.         .inst_addr (inst_addr ), // 0, 32
296.         .IF_over   (IF_over   ), // 0, 1
297.         .IF_ID_bus (IF_ID_bus ), // 0, 64
298.
299.         //5 级流水新增接口
300.         .exc_bus   (exc_bus   ), // I, 32
301.
302.         //MOD 6 级流水修改
303.         .br_bus    (br_bus    ),
304.         .j_bus     (j_bus     ),
305.
306.         //展示 PC 和取出的指令
307.         .IF_pc     (IF_pc     ), // 0, 32
308.         .IF_inst   (IF_inst   ), // 0, 32
309.         .dbg_next_pc(dbg_next_pc)
310.     );
311.
312.     decode ID_module( // 译码级
313.         .ID_valid   (ID_valid   ), // I, 1
314.         .IF_ID_bus_r(IF_ID_bus_r), // I, 64
315. //         .rs_value  (rs_value  ), // I, 32
316. //         .rt_value  (rt_value  ), // I, 32
317. //         .rs        (rs        ), // 0, 5
318. //         .rt        (rt        ), // 0, 5
319. //         .jbr_bus   (jbr_bus   ), // 0, 33
320.         .ID_over    (ID_over    ), // 0, 1
321.         .ID_EXE_bus (ID_REG_bus), // 0, 167
322.
323.         //5 级流水新增
324.         .IF_over     (IF_over     ),// I, 1
325. //         .EXE_wdest  (EXE_wdest  ),// I, 5
326. //         .MEM_wdest  (MEM_wdest  ),// I, 5
327. //         .WB_wdest   (WB_wdest   ),// I, 5
328.
329.         //MOD 6 级流水修改
330.         .j_bus       (j_bus       ),
331.
332.         //展示 PC
333.         .ID_pc        (ID_pc        ) // 0, 32
334.     );
335.

```

```

336. //MOD 6 级流水修改
337. read REG_module( // 读寄存器级
338.     .REG_valid(REG_valid),
339.     .ID_REG_bus(ID_REG_bus_r),
340.     .IF_over (IF_over ), // I, 1
341.     .rs_value (rs_value ), // I, 32
342.     .rt_value (rt_value ), // I, 32
343.     .rs (rs ), // 0, 5
344.     .rt (rt ), // 0, 5
345.     .EXE_wdest (EXE_wdest ), // I, 5
346.     .MEM_wdest (MEM_wdest ), // I, 5
347.     .WB_wdest (WB_wdest ), // I, 5
348.     .EXE_wvalue (EXE_wvalue),
349.     .MEM_wvalue (MEM_wvalue),
350.     .WB_wvalue (WB_wvalue),
351.     .EXE_wvalid (EXE_wvalid),
352.     .MEM_wvalid (MEM_wvalid),
353.     .WB_wvalid (WB_wvalid ),
354.
355.     .REG_over(REG_over),
356.     .REG_EXE_bus(REG_EXE_bus),
357.     .br_bus (br_bus ),
358.     .predict_fail(predict_fail),
359.
360.     .REG_pc(REG_pc)
361. );
362.
363. exe EXE_module( // 执行级
364.     .EXE_valid (EXE_valid ), // I, 1
365.     .ID_EXE_bus_r(REG_EXE_bus_r), // I, 167
366.     .EXE_over (EXE_over ), // 0, 1
367.     .EXE_MEM_bus (EXE_MEM_bus ), // 0, 154
368. //MOD store rt 的旁路
369.     .MEM_wdest (MEM_wdest ), // I, 5
370.     .WB_wdest (WB_wdest ), // I, 5
371.     .MEM_wvalue (MEM_wvalue),
372.     .WB_wvalue (WB_wvalue),
373.     .MEM_wvalid (MEM_wvalid),
374.     .WB_wvalid (WB_wvalid ),
375. //5 级流水新增
376.     .clk (clk ), // I, 1
377.     .EXE_wdest (EXE_wdest ), // 0, 5
378.
379. //数据旁路

```

```

380.         .EXE_wvalue(EXE_wvalue),
381.         .EXE_wvalid(EXE_wvalid),
382.
383.         //展示 PC
384.         .EXE_pc      (EXE_pc      )    // 0, 32
385.     );
386.
387.     mem MEM_module(                // 访存级
388.         .clk          (clk          ), // I, 1
389.         .MEM_valid    (MEM_valid    ), // I, 1
390.         .EXE_MEM_bus_r(EXE_MEM_bus_r), // I, 154
391.         .dm_rdata     (dm_rdata     ), // I, 32
392.         .dm_addr      (dm_addr      ), // 0, 32
393.         .dm_wen       (dm_wen       ), // 0, 4
394.         .dm_wdata     (dm_wdata     ), // 0, 32
395.         .MEM_over     (MEM_over     ), // 0, 1
396.         .MEM_WB_bus   (MEM_WB_bus   ), // 0, 118
397.
398.         //5 级流水新增接口
399.         .MEM_allow_in (MEM_allow_in ), // I, 1
400.         .MEM_wdest    (MEM_wdest    ), // 0, 5
401.
402.         //6 级旁路
403.         .MEM_wvalue    (MEM_wvalue   ),
404.         .MEM_wvalid    (MEM_wvalid   ),
405.
406.         //展示 PC
407.         .MEM_pc        (MEM_pc        ) // 0, 32
408.     );
409.
410.     wb WB_module(                // 写回级
411.         .WB_valid     (WB_valid      ), // I, 1
412.         .MEM_WB_bus_r(MEM_WB_bus_r), // I, 118
413.         .rf_wen       (rf_wen       ), // 0, 1
414.         .rf_wdest     (rf_wdest     ), // 0, 5
415.         .rf_wdata     (rf_wdata     ), // 0, 32
416.         .WB_over      (WB_over      ), // 0, 1
417.
418.         //5 级流水新增接口
419.         .clk          (clk          ), // I, 1
420.         .resetn       (resetn       ), // I, 1
421.         .exc_bus      (exc_bus      ), // 0, 32
422.         .WB_wdest     (WB_wdest     ), // 0, 5
423.         .cancel       (cancel       ), // 0, 1

```

```

424.         .WB_wvalid  (WB_wvalid  ),
425.         .WB_wvalue  (WB_wvalue  ),
426.
427.         //展示 PC 和 HI/LO 值
428.         .WB_pc      (WB_pc      ), // 0, 32
429.         .HI_data    (HI_data    ), // 0, 32
430.         .LO_data    (LO_data    ) // 0, 32
431.     );
432.
433.     inst_rom inst_rom_module(        // 指令存储器
434.         .clk        (clk          ), // I, 1 ,时钟
435.         .addra      (inst_addr[9:2]), // I, 8 ,指令地址
436.         .douta      (inst         )  // 0, 32,指令
437.     );
438.
439.     regfile rf_module(               // 寄存器堆模块
440.         .clk        (clk          ), // I, 1
441.         .wen        (rf_wen       ), // I, 1
442.         .raddr1     (rs           ), // I, 5
443.         .raddr2     (rt           ), // I, 5
444.         .waddr      (rf_wdest     ), // I, 5
445.         .wdata      (rf_wdata     ), // I, 32
446.         .rdata1     (rs_value     ), // 0, 32
447.         .rdata2     (rt_value     ), // 0, 32
448.
449.         //display rf
450.         .test_addr  (rf_addr),    // I, 5
451.         .test_data  (rf_data)    // 0, 32
452.     );
453.
454.     data_ram data_ram_module(        // 数据存储模块
455.         .clk        (clk          ), // I, 1, 时钟
456.         .wea        (dm_wen       ), // I, 1, 写使能
457.         .addra      (dm_addr[9:2]), // I, 8, 读地址
458.         .dina       (dm_wdata     ), // I, 32, 写数据
459.         .douta      (dm_rdata     ), // 0, 32, 读数据
460.
461.         //display mem
462.         .clkb       (clk          ), // I, 1, 时钟
463.         .web        (4'd0         ), // 不使用端口 2 的写功能
464.         .addrb      (mem_addr[9:2]), // I, 8, 读地址
465.         .doutb      (mem_data     ), // I, 32, 写数据
466.         .dinb       (32'd0        ) // 不使用端口 2 的写功能
467.     );

```

```

468. //-----{各模块实例化}end-----
      //
469. //展示
470. assign dbg_inst=inst;
471. assign dbg_IF_allow_in=IF_allow_in;
472. assign dbg_inst_addr=inst_addr;
473. assign dbg_ID_inst=IF_ID_bus_r[31:0];
474. assign dbg_j_bus=j_bus;
475. assign dbg_br_bus=br_bus;
476. assign dbg_rs=rs;
477. assign dbg_rt=rt;
478. assign dbg_rs_value=rs_value;
479. assign dbg_rt_value=rt_value;
480. assign dbg_EXE_wdest=EXE_wdest;
481. assign dbg_MEM_wdest=MEM_wdest;
482. assign dbg_WB_wdest=WB_wdest;
483. assign dbg_EXE_wvalid=EXE_wvalid;
484. assign dbg_MEM_wvalid=MEM_wvalid;
485. assign dbg_WB_wvalid=WB_wvalid;
486. assign dbg_EXE_wvalue=EXE_wvalue;
487. assign dbg_MEM_wvalue=MEM_wvalue;
488. assign dbg_WB_wvalue=WB_wvalue;
489. endmodule

```

A.2 fetch

完整代码

```

1. `timescale 1ns / 1ps
2.
3. `define STARTADDR 32'H00000034 // 程序起始地址为 34H
4. module fetch( // 取指级
5.     input clk, // 时钟
6.     input resetn, // 复位信号，低电平有效
7.     input IF_valid, // 取指级有效信号
8.     input next_fetch, // 取下一条指令，用来锁存 PC 值
9.     input [31:0] inst, // inst_rom 取出的指令
10. // input [32:0] jbr_bus, // 跳转总线
11.     output [31:0] inst_addr, // 发往 inst_rom 的取指地址
12.     output reg IF_over, // IF 模块执行完成
13.     output [63:0] IF_ID_bus, // IF->ID 总线
14.
15. //5 级流水新增接口
16.     input [32:0] exc_bus, // Exception pc 总线
17.
18. //MOD 6 级流水修改

```

```

19.     input    [32:0] j_bus,    // 无条件跳转总线
20.     input    [32:0] br_bus,   // 分支跳转总线
21.
22.     //展示 PC 和取出的指令
23.     output    [31:0] IF_pc,
24.     output    [31:0] IF_inst,
25.     output    [31:0] dbg_next_pc
26. );
27.
28.     //-----{程序计数器 PC}begin
29.     wire [31:0] next_pc;
30.     wire [31:0] seq_pc;
31.     reg  [31:0] pc;
32.
33.     //跳转 pc
34.     //     wire          jbr_taken;
35.     //     wire [31:0] jbr_target;
36.     //     assign {jbr_taken, jbr_target} = jbr_bus; // 跳转总线传是否跳转和目标
           地址
37.     wire j_taken;
38.     wire [31:0] j_target;
39.     wire br_taken;
40.     wire [31:0] br_target;
41.     assign {j_taken,j_target}=j_bus;
42.     assign {br_taken,br_target}=br_bus;
43.
44.     //Exception PC
45.     wire      exc_valid;
46.     wire [31:0] exc_pc;
47.     assign {exc_valid,exc_pc} = exc_bus;
48.
49.     //pc+4
50.     assign seq_pc[31:2]    = pc[31:2] + 1'b1; // 下一指令地址: PC=PC+4
51.     assign seq_pc[1:0]    = pc[1:0];
52.
53.     // 新指令: 若有 Exception,则 PC 为 Exceptio 入口地址
54.     //           若指令跳转, 则 PC 为跳转地址; 否则为 pc+4
55.     reg init=0;
56.
57.     //     assign next_pc = exc_valid ? exc_pc :
58.     //                               jbr_taken ? jbr_target :
59.     //                               seq_pc;
60.     //MOD
61.     assign next_pc = exc_valid ? exc_pc      :

```



```

62.                br_taken ? br_target :
63.                j_taken  ? j_target  :
64.                seq_pc;
65.
66.    always @(posedge clk)    // PC 程序计数器
67.    begin
68.        if (!resetn)
69.        begin
70.            pc <= `STARTADDR; // 复位，取程序起始地址
71.            init <=1;
72.        end
73.        else if (next_fetch)
74.        begin
75.            pc <= next_pc;    // 不复位，取新指令
76.        end
77.    end
78.
79.
80.    //-----{程序计数器 PC}end
81.
82.    //-----{发往 inst_rom 的取指地址}begin
83.        assign inst_addr = next_fetch ? next_pc : pc;
84.    //-----{发往 inst_rom 的取指地址}end
85.
86.    //-----{IF 执行完成}begin
87.        //由于指令 rom 为同步读写的，
88.        //取数据时，有一拍延时
89.        //即发地址的下一拍时钟才能得到对应的指令
90.        //故取指模块需要两拍时间
91.        //故每次 PC 刷新，IF_over 都要置 0
92.        //然后将 IF_valid 锁存一拍即是 IF_over 信号
93.    always @(posedge clk)
94.    begin
95.        if (!resetn)
96.        //        if (!resetn || next_fetch)
97.        begin
98.            IF_over <= 1'b0;
99.        end
100.        else
101.        begin
102.            IF_over <= IF_valid;
103.        end
104.    end
105.    //如果指令 rom 为异步读的，则 IF_valid 即是 IF_over 信号，

```

```

106.      //即取指一拍完成
107.      //-----{IF 执行完成}end
108.
109.      //-----{IF->ID 总线}begin
110.          assign IF_ID_bus = {pc, inst}; // 取指级有效时，锁存 PC 和指令
111.      //-----{IF->ID 总线}end
112.
113.      //-----{展示 IF 模块的 PC 值和指令}begin
114.          assign IF_pc    = pc;
115.          assign IF_inst = inst;
116.      //-----{展示 IF 模块的 PC 值和指令}end
117.          assign dbg_next_pc=next_pc;
118.      endmodule

```

A. 3 decode. v

完整代码

```

1.      `timescale 1ns / 1ps
2.
3.      module decode(                                // 译码级
4.          input          ID_valid,                  // 译码级有效信号
5.          input    [ 63:0] IF_ID_bus_r,             // IF->ID 总线
6.          output         ID_over,                   // ID 模块执行完成
7.          output    [188:0] ID_EXE_bus,             // ID->EXE 总线
8.
9.          //5 级流水新增
10.         input          IF_over,                   //对于分支指令，需要该信号
11.
12.         //MOD 6 级流水修改
13.         output    [ 32:0] j_bus,
14.
15.         //展示 PC
16.         output    [ 31:0] ID_pc
17.     );
18.     //-----{IF->ID 总线}begin
19.         wire [31:0] pc;
20.         wire [31:0] inst;
21.         assign {pc, inst} = IF_ID_bus_r; // IF->ID 总线传 PC 和指令
22.     //-----{IF->ID 总线}end
23.
24.     //-----{指令译码}begin
25.         wire [5:0] op;
26.         wire [4:0] rd;
27.         wire [4:0] sa;
28.         wire [5:0] funct;

```

```

29.    wire [15:0] imm;
30.    wire [15:0] offset;
31.    wire [25:0] target;
32.    wire [2:0] cp0r_sel;
33.    wire [ 4:0] rs;
34.    wire [ 4:0] rt;
35.
36.    assign op      = inst[31:26]; // 操作码
37.    assign rs      = inst[25:21]; // 源操作数 1
38.    assign rt      = inst[20:16]; // 源操作数 2
39.    assign rd      = inst[15:11]; // 目标操作数
40.    assign sa      = inst[10:6];  // 特殊域, 可能存放偏移量
41.    assign funct    = inst[5:0];  // 功能码
42.    assign imm      = inst[15:0]; // 立即数
43.    assign offset   = inst[15:0]; // 地址偏移量
44.    assign target   = inst[25:0]; // 目标地址
45.    assign cp0r_sel = inst[2:0];  // cp0 寄存器的 select 域
46.
47.    // 实现指令列表
48.    wire inst_ADDU, inst_SUBU, inst_SLT, inst_AND;
49.    wire inst_NOR, inst_OR, inst_XOR, inst_SLL;
50.    wire inst_SRL, inst_ADDIU, inst_BEQ, inst_BNE;
51.    wire inst_LW, inst_SW, inst_LUI, inst_J;
52.    wire inst_SLTU, inst_JALR, inst_JR, inst_SLLV;
53.    wire inst_SRA, inst_SRAV, inst_SRLV, inst_SLTIU;
54.    wire inst_SLTI, inst_BGEZ, inst_BGTZ, inst_BLEZ;
55.    wire inst_BLTZ, inst_LB, inst_LBU, inst_SB;
56.    wire inst_ANDI, inst_ORI, inst_XORI, inst_JAL;
57.    wire inst_MULT, inst_MFLO, inst_MFHI, inst_MTLO;
58.    wire inst_MTHI, inst_MFC0, inst_MTC0;
59.    wire inst_ERET, inst_SYSCALL;
60.    wire op_zero; // 操作码全 0
61.    wire sa_zero; // sa 域全 0
62.    assign op_zero = ~(|op);
63.    assign sa_zero = ~(|sa);
64.    assign inst_ADDU = op_zero & sa_zero & (funct == 6'b100001); // 无符号加法
65.    assign inst_SUBU = op_zero & sa_zero & (funct == 6'b100011); // 无符号减法
66.    assign inst_SLT = op_zero & sa_zero & (funct == 6'b101010); // 小于则置位
67.    assign inst_SLTU = op_zero & sa_zero & (funct == 6'b101011); // 无符号小于则置
68.    assign inst_JALR = op_zero & (rt==5'd0) & (rd==5'd31)

```

69.		& sa_zero & (funct == 6'b001001);	//跳转寄存器并链接
70.	assign inst_JR	= op_zero & (rt==5'd0) & (rd==5'd0)	
71.		& sa_zero & (funct == 6'b001000);	//跳转寄存器
72.	assign inst_AND	= op_zero & sa_zero & (funct == 6'b100100);	//与运算
73.	assign inst_NOR	= op_zero & sa_zero & (funct == 6'b100111);	//或非运算
74.	assign inst_OR	= op_zero & sa_zero & (funct == 6'b100101);	//或运算
75.	assign inst_XOR	= op_zero & sa_zero & (funct == 6'b100110);	//异或运算
76.	assign inst_SLL	= op_zero & (rs==5'd0) & (funct == 6'b000000);	//逻辑左移
77.	assign inst_SLLV	= op_zero & sa_zero & (funct == 6'b000100);	//变量逻辑左移
78.	assign inst_SRA	= op_zero & (rs==5'd0) & (funct == 6'b000011);	//算术右移
79.	assign inst_SRAV	= op_zero & sa_zero & (funct == 6'b000111);	//变量算术右移
80.	assign inst_SRL	= op_zero & (rs==5'd0) & (funct == 6'b000010);	//逻辑右移
81.	assign inst_SRLV	= op_zero & sa_zero & (funct == 6'b000110);	//变量逻辑右移
82.	assign inst_MULT	= op_zero & (rd==5'd0)	
83.		& sa_zero & (funct == 6'b011000);	//乘法
84.	assign inst_MFLO	= op_zero & (rs==5'd0) & (rt==5'd0)	
85.		& sa_zero & (funct == 6'b010010);	//从LO读取
86.	assign inst_MFHI	= op_zero & (rs==5'd0) & (rt==5'd0)	
87.		& sa_zero & (funct == 6'b010000);	//从HI读取
88.	assign inst_MTLO	= op_zero & (rt==5'd0) & (rd==5'd0)	
89.		& sa_zero & (funct == 6'b010011);	//向LO写数据
90.	assign inst_MTHI	= op_zero & (rt==5'd0) & (rd==5'd0)	
91.		& sa_zero & (funct == 6'b010001);	//向HI写数据
92.	assign inst_ADDIU	= (op == 6'b001001);	//立即数无符号加法
93.	assign inst_SLTI	= (op == 6'b001010);	//小于立即数则置位
94.	assign inst_SLTIU	= (op == 6'b001011);	//小于立即数则置位(无符号)

```

95.      assign inst_BEQ    = (op == 6'b000100);           //判断相等跳转
96.      assign inst_BGEZ  = (op == 6'b000001) & (rt==5'd1); //大于等于 0 跳转
97.      assign inst_BGTZ  = (op == 6'b000111) & (rt==5'd0); //大于 0 跳转
98.      assign inst_BLEZ  = (op == 6'b000110) & (rt==5'd0); //小于等于 0 跳转
99.      assign inst_BLTZ  = (op == 6'b000001) & (rt==5'd0); //小于 0 跳转
100.     assign inst_BNE   = (op == 6'b000101);           //判断不等跳转
101.     assign inst_LW    = (op == 6'b100011);           //从内存装载字
102.     assign inst_SW    = (op == 6'b101011);           //向内存存储字
103.     assign inst_LB     = (op == 6'b100000);           //load 字节（符号扩
展）
104.     assign inst_LBU   = (op == 6'b100100);           //load 字节（无符号扩
展）
105.     assign inst_SB    = (op == 6'b101000);           //向内存存储字节
106.     assign inst_ANDI   = (op == 6'b001100);           //立即数与
107.     assign inst_LUI    = (op == 6'b001111) & (rs==5'd0); //立即数装载高半字
节
108.     assign inst_ORI   = (op == 6'b001101);           //立即数或
109.     assign inst_XORI   = (op == 6'b001110);           //立即数异或
110.     assign inst_J     = (op == 6'b000010);           //跳转
111.     assign inst_JAL   = (op == 6'b000011);           //跳转和链接
112.     assign inst_MFC0  = (op == 6'b010000) & (rs==5'd0)
113.     & sa_zero & (funct[5:3] == 3'b000); // 从 cp0 寄存器
装载
114.     assign inst_MTC0  = (op == 6'b010000) & (rs==5'd4)
115.     & sa_zero & (funct[5:3] == 3'b000); // 向 cp0 寄存器
存储
116.     assign inst_SYSCALL = (op == 6'b000000) & (funct == 6'b001100); // 系
统调用
117.     assign inst_ERET  = (op == 6'b010000) & (rs==5'd16) & (rt==5'd0)
118.     & (rd==5'd0) & sa_zero & (funct == 6'b011000); //异
常返回
119.
120.     //跳转分支指令
121.     wire inst_jr;      //寄存器跳转指令
122.     wire inst_j_link;  //链接跳转指令
123.     wire inst_jbr;     //所有分支跳转指令
124.     assign inst_jr     = inst_JALR | inst_JR;
125.     assign inst_j_link = inst_JAL | inst_JALR;
126.
127.     //load store
128.     wire inst_load;
129.     wire inst_store;
130.     assign inst_load = inst_LW | inst_LB | inst_LBU; // load 指令
131.     assign inst_store = inst_SW | inst_SB;           // store 指令

```

```

132.
133.    //alu 操作分类
134.    wire inst_add, inst_sub, inst_slt,inst_sltu;
135.    wire inst_and, inst_nor, inst_or, inst_xor;
136.    wire inst_sll, inst_srl, inst_sra,inst_lui;
137.    assign inst_add = inst_ADDU | inst_ADDIU | inst_load
138.                | inst_store | inst_j_link;           // 做加法
139.    assign inst_sub = inst_SUBU;                       // 减法
140.    assign inst_slt = inst_SLT | inst_SLTI;            // 有符号小于置
    位
141.    assign inst_sltu= inst_SLTIU | inst_SLTU;          // 无符号小于置
    位
142.    assign inst_and = inst_AND | inst_ANDI;           // 逻辑与
143.    assign inst_nor = inst_NOR;                       // 逻辑或非
144.    assign inst_or  = inst_OR  | inst_ORI;            // 逻辑或
145.    assign inst_xor = inst_XOR | inst_XORI;           // 逻辑异或
146.    assign inst_sll = inst_SLL | inst_SLLV;           // 逻辑左移
147.    assign inst_srl = inst_SRL | inst_SRLV;           // 逻辑右移
148.    assign inst_sra = inst_SRA | inst_SRAV;           // 算术右移
149.    assign inst_lui = inst_LUI;                       // 立即数装载高
    位
150.
151.    //使用 sa 域作为偏移量的移位指令
152.    wire inst_shf_sa;
153.    assign inst_shf_sa = inst_SLL | inst_SRL | inst_SRA;
154.
155.    //依据立即数扩展方式分类
156.    wire inst_imm_zero; //立即数 0 扩展
157.    wire inst_imm_sign; //立即数符号扩展
158.    assign inst_imm_zero = inst_ANDI | inst_LUI | inst_ORI | inst_XORI;
159.    assign inst_imm_sign = inst_ADDIU | inst_SLTI | inst_SLTIU
160.                | inst_load | inst_store;
161.
162.    //依据目的寄存器号分类
163.    wire inst_wdest_rt; // 寄存器堆写入地址为 rt 的指令
164.    wire inst_wdest_31; // 寄存器堆写入地址为 31 的指令
165.    wire inst_wdest_rd; // 寄存器堆写入地址为 rd 的指令
166.    assign inst_wdest_rt = inst_imm_zero | inst_ADDIU | inst_SLTI
167.                | inst_SLTIU | inst_load | inst_MFC0;
168.    assign inst_wdest_31 = inst_JAL;
169.    assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_SLTU
170.                | inst_JALR | inst_AND | inst_NOR | inst_OR

```

```

171.                | inst_XOR | inst_SLL | inst_SLLV | inst_SRA
172.                | inst_SRAV | inst_SRL | inst_SRLV
173.                | inst_MFHI | inst_MFLO;
174.
175.    //依据源寄存器号分类
176.    wire inst_no_rs; //指令 rs 域非 0, 且不是从寄存器堆读 rs 的数据
177.    wire inst_no_rt; //指令 rt 域非 0, 且不是从寄存器堆读 rt 的数据
178.    assign inst_no_rs = inst_MTC0 | inst_SYSCALL | inst_ERET;
179.    assign inst_no_rt = inst_ADDIU | inst_SLTI | inst_SLTIU
180.                | inst_BGEZ | inst_load | inst_imm_zero
181.                | inst_J | inst_JAL | inst_MFC0
182.                | inst_SYSCALL;
183.    //-----{指令译码}end
184.
185.    //-----{分支指令执行}begin
186.    //bd_pc,分支跳转指令参与计算的为延迟槽指令的 PC 值, 即当前分支指令的 PC+4
187.    wire [31:0] bd_pc; //延迟槽指令 PC 值
188.    assign bd_pc = pc + 3'b100;
189.
190.    //无条件跳转
191.    wire j_taken;
192.    wire [31:0] j_target;
193.    //MOD
194.    assign j_taken = (inst_J | inst_JAL) & ID_over ;
195.    //寄存器跳转地址为 rs_value,其他跳转为{bd_pc[31:28],target,2'b00}
196.    assign j_target = {bd_pc[31:28],target,2'b00};
197.    assign j_bus = {j_taken, j_target};
198.
199.    //-----{ID 执行完成}end
200.
201.    //-----{ID->EXE 总线}begin
202.    //EXE 需要用到的信息
203.    wire multiply; //乘法 MULT
204.    wire mthi; //MTHI
205.    wire mtlo; //MTLO
206.    assign multiply = inst_MULT;
207.    assign mthi = inst_MTHI;
208.    assign mtlo = inst_MTLO;
209.    //ALU 两个源操作数和控制信号
210.    wire [11:0] alu_control;
211.    wire [31:0] alu_operand1;
212.    wire [31:0] alu_operand2;
213.

```

```

214.    //所谓链接跳转是将跳转返回的 PC 值存放到 31 号寄存器里
215.    //在流水 CPU 里，考虑延迟槽，故链接跳转需要计算 PC+8，存放到 31 号寄存器里
216.
217.    assign alu_control = {inst_add,          // ALU 操作码，独热编码
218.                          inst_sub,
219.                          inst_slt,
220.                          inst_sltu,
221.                          inst_and,
222.                          inst_nor,
223.                          inst_or,
224.                          inst_xor,
225.                          inst_sll,
226.                          inst_srl,
227.                          inst_sra,
228.                          inst_lui};
229.
230.    //MOD REG 需要用到的
231.    assign alu_operand1 = inst_j_link ? pc :
232.                          inst_shf_sa ? {27'd0,sa} : {31'd0};
233.    assign alu_operand2 = inst_j_link ? 32'd8 :
234.                          inst_imm_zero ? {16'd0, imm} :
235.                          inst_imm_sign ? {{16{imm[15]}}, imm} : {31'd0};
236.
237.    wire read_rs;
238.    wire read_rt;
239.    wire [7:0] br_control;
240.    wire [31:0] br_target;
241.
242.    assign read_rs=~(inst_j_link|inst_shf_sa);
243.    assign read_rt=~(inst_j_link|inst_imm_zero|inst_imm_sign);
244.    assign br_control={inst_jr,
245.                      inst_BEQ,
246.                      inst_BNE,
247.                      inst_BGEZ,
248.                      inst_BGTZ,
249.                      inst_BLEZ,
250.                      inst_BLTZ};
251.    assign br_target[31:2] = bd_pc[31:2] + {{14{offset[15]}}, offset};
252.    assign br_target[1:0] = bd_pc[1:0];
253.    //rs,rt
254.
255.    //访存需要用到的 load/store 信息
256.    wire lb_sign; //load 一字节为有符号 load

```



```

257.    wire ls_word; //load/store 为字节还是字,0:byte;1:word
258.    wire [3:0] mem_control; //MEM 需要使用的控制信号
259.    assign lb_sign = inst_LB;
260.    assign ls_word = inst_LW | inst_SW;
261.    assign mem_control = {inst_load,
262.                          inst_store,
263.                          ls_word,
264.                          lb_sign };
265.
266.    //写回需要用到的信息
267.    wire mfhi;
268.    wire mflo;
269.    wire mtc0;
270.    wire mfc0;
271.    wire [7 :0] cp0r_addr;
272.    wire        syscall; //syscall 和 eret 在写回级有特殊的操作
273.    wire        eret;
274.    wire        rf_wen; //写回的寄存器写使能
275.    wire [4:0] rf_wdest; //写回的目的寄存器
276.    assign syscall = inst_SYSCALL;
277.    assign eret    = inst_ERET;
278.    assign mfhi    = inst_MFHI;
279.    assign mflo    = inst_MFLO;
280.    assign mtc0    = inst_MTC0;
281.    assign mfc0    = inst_MFC0;
282.    assign cp0r_addr= {rd,cp0r_sel};
283.    assign rf_wen   = inst_wdest_rt | inst_wdest_31 | inst_wdest_rd;
284.    assign rf_wdest = inst_wdest_rt ? rt : //在不写寄存器堆时设置为 0
285.                          inst_wdest_31 ? 5'd31 : //以便能准确判断数据相关
286.                          inst_wdest_rd ? rd : 5'd0;
287.    //    assign store_data = rt_value;
288.    assign ID_EXE_bus = {read_rs,read_rt,br_control,br_target, //MOD:REG 需
        用的信息
289.                          rs,rt,inst_no_rs,inst_no_rt,
290.                          multiply,mthi,mtlo, //EXE 需用的
        信息,新增
291.                          alu_control,alu_operand1,alu_operand2, //EXE 需用的
        信息
292.                          mem_control,/*store_data,*/ //MEM 需用的
        信号
293.                          mfhi,mflo, //WB 需用的
        信号,新增
294.                          mtc0,mfc0,cp0r_addr,syscall,eret, //WB 需用的
        信号,新增

```

```

295.             rf_wen, rf_wdest,           //WB 需用的
           信号
296.             pc};                       //PC 值
297.
298. //-----{ID->EXE 总线}end
299.
300. //-----{展示 ID 模块的 PC 值}begin
301.     assign ID_pc = pc;
302. //-----{展示 ID 模块的 PC 值}end
303. endmodule

```

A.4 read.v

完整代码

```

1. `timescale 1ns / 1ps
2.
3. module read(
4.     input REG_valid,
5.     input [188:0] ID_REG_bus,
6.     input IF_over, //对于分支指令, 需要该信号
7.     input [31:0] rs_value,
8.     input [31:0] rt_value,
9.
10.    input [ 4:0] EXE_wdest, // EXE 级要写回寄存器堆的目标地址号
11.    input [ 4:0] MEM_wdest, // MEM 级要写回寄存器堆的目标地址号
12.    input [ 4:0] WB_wdest, // WB 级要写回寄存器堆的目标地址号
13.
14.    input [31:0] EXE_wvalue,
15.    input [31:0] MEM_wvalue,
16.    input [31:0] WB_wvalue,
17.
18.    input EXE_wvalid,
19.    input MEM_wvalid,
20.    input WB_wvalid,
21.
22.    output REG_over,
23.    output [171:0] REG_EXE_bus,
24.    output [4:0] rs,
25.    output [4:0] rt,
26.    output [32:0] br_bus,
27.    output predict_fail,
28.    //展示 PC
29.    output [ 31:0] REG_pc
30. );
31.

```

```

32.    //-----{ID->REG 总线}begin
33.        wire read_rs;
34.        wire read_rt;
35.        wire inst_no_rs;
36.        wire inst_no_rt;
37.        wire [7:0] br_control;
38.        wire [31:0] br_target;
39.        wire multiply;
40.        wire mthi;
41.        wire mtlo;
42.        wire [11:0] alu_control;
43.        wire [31:0] alu_operand1;
44.        wire [31:0] alu_operand2;
45.        wire [3:0] mem_control;
46.        wire mfhi;
47.        wire mflo;
48.        wire mtc0;
49.        wire mfc0;
50.        wire [7 :0] cp0r_addr;
51.        wire syscall;
52.        wire eret;
53.        wire rf_wen;
54.        wire [4:0] rf_wdest;
55.        wire [31:0]pc;
56.        assign {read_rs,read_rt,br_control,br_target,
57.                rs,rt,inst_no_rs,inst_no_rt,
58.                multiply,mthi,mtlo,
59.                alu_control,alu_operand1,alu_operand2,
60.                mem_control,
61.                mfhi,mflo,
62.                mtc0,mfc0,cp0r_addr,syscall,eret,
63.                rf_wen, rf_wdest,
64.                pc}=ID_REG_bus;
65.
66.        assign REG_pc=pc;
67.    //-----{ID->REG 总线}end
68.
69.    //-----{数据相关}begin
70.        //store 旁路单独处理，可以在 exe 再读
71.        wire [4:0] store_rf_addr;
72.        wire store_read_rf;
73.
74.        assign store_read_rf = ~inst_no_rt & (rt!=5'd0) & mem_control[2]; //store inst

```

```

75.      assign store_rf_addr = rt &{5{store_read_rf}};
76.
77.      //由于是流水的，存在数据相关
78.      wire rs_wait;
79.      wire rt_wait;
80.      assign rs_wait = ~inst_no_rs & (rs!=5'd0)
81.                  & ( (rs==EXE_wdest && ~EXE_wvalid) | (rs==MEM_wdest &&
~MEM_wvalid) | (rs==WB_wdest && ~WB_wvalid));
82.      assign rt_wait = ~inst_no_rt & (rt!=5'd0)
83.                  & ( (rt==EXE_wdest && ~EXE_wvalid) | (rt==MEM_wdest &&
~MEM_wvalid) | (rt==WB_wdest && ~MEM_wvalid))
84.                  & ~store_read_rf;
85.
86.      wire inst_jbr;
87.      assign inst_jbr= |br_control;
88.      //对于分支跳转指令，只有在 IF 执行完成后，才可以算 ID 完成;
89.      //否则，ID 级先完成了，而 IF 还在取指令，则 next_pc 不能锁存到 PC 里去，
90.      //那么等 IF 完成，next_pc 能锁存到 PC 里去时，jbr_bus 上的数据已变成无效，
91.      //导致分支跳转失败
92.      //(~inst_jbr | IF_over)即是(~inst_jbr | (inst_jbr & IF_over))
93.
94.      assign REG_over = REG_valid & ~rs_wait & ~rt_wait & (~inst_jbr | IF_ov
er);
95.
96.      wire [31:0] forward_rs_value;
97.      wire [31:0] forward_rt_value;
98.
99.      assign forward_rs_value = (rs == EXE_wdest) ? EXE_wvalue:
100.                               (rs==MEM_wdest) ? MEM_wvalue:
101.                               (rs==WB_wdest) ? WB_wvalue:
102.                               rs_value;
103.      assign forward_rt_value = (rt == EXE_wdest) ? EXE_wvalue:
104.                               (rt ==MEM_wdest) ? MEM_wvalue:
105.                               (rt ==WB_wdest) ? WB_wvalue:
106.                               rt_value;
107.
108.
109.      wire [31:0] alu_op1;
110.      wire [31:0] alu_op2;
111.      assign alu_op1 = read_rs ? forward_rs_value : alu_operand1;
112.      assign alu_op2 = read_rt ? forward_rt_value : alu_operand2;
113.
114.
115.      //-----{数据相关}end

```

```

116.
117.    //-----{分支}begin
118.        wire inst_jr;
119.        wire inst_BEQ;
120.        wire inst_BNE;
121.        wire inst_BGEZ;
122.        wire inst_BGTZ;
123.        wire inst_BLEZ;
124.        wire inst_BLTZ;
125.        assign {inst_jr,
126.                inst_BEQ,
127.                inst_BNE,
128.                inst_BGEZ,
129.                inst_BGTZ,
130.                inst_BLEZ,
131.                inst_BLTZ}=br_control;
132.
133.        wire br_taken;
134.        wire rs_equql_rt;
135.        wire rs_ez;
136.        wire rs_ltz;
137.        assign rs_equql_rt = (forward_rs_value == forward_rt_value); // GPR[r
                                s]==GPR[rt]
138.        assign rs_ez      = ~(|forward_rs_value);           // rs 寄存器值为
                                0
139.        assign rs_ltz     = forward_rs_value[31];           // rs 寄存器值小于
                                0
140.        assign br_taken = inst_BEQ & rs_equql_rt           // 相等跳转
141.                        | inst_BNE & ~rs_equql_rt          // 不等跳转
142.                        | inst_BGEZ & ~rs_ltz              // 大于等于 0 跳转
143.                        | inst_BGTZ & ~rs_ltz & ~rs_ez     // 大于 0 跳转
144.                        | inst_BLEZ & (rs_ltz | rs_ez)      // 小于等于 0 跳转
145.                        | inst_BLTZ & rs_ltz               // 小于 0 跳转
146.                        | inst_jr;
147.
148.        wire [31:0] r_br_target;
149.        assign r_br_target=inst_jr ? forward_rs_value : br_target;
150.        assign br_bus={br_taken,r_br_target};
151.        assign predict_fail=br_taken;
152.    //-----{分支}end
153.
154.    //-----{REG->EXE 总线}begin
155.        wire [31:0]store_data;
156.        assign store_data=forward_rt_value;

```

```

157.         assign REG_EXE_bus = {multiply,mthi,mtlo,           //EXE 需用
           的信息,新增
158.                                     alu_control,alu_op1,alu_op2,       //EXE 需用
           的信息
159.                                     mem_control,store_data,store_rf_addr, //MEM 需用
           的信号
160.                                     mfhi,mflo,           //WB 需用的
           信号,新增
161.                                     mtc0,mfc0,cp0r_addr,syscall,eret,     //WB 需用的
           信号,新增
162.                                     rf_wen, rf_wdest,           //WB 需用的
           信号
163.                                     pc};           //PC 值
164.
165. //-----{REG->EXE 总线}end
166.
167.     endmodule

```

A.5 exe.v

完整代码	
1.	`timescale 1ns / 1ps
2.	
3.	module exe(// 执行级
4.	input EXE_valid, // 执行级有效信号
5.	input [171:0] ID_EXE_bus_r, // ID->EXE 总线
6.	output EXE_over, // EXE 模块执行完成
7.	output [153:0] EXE_MEM_bus, // EXE->MEM 总线
8.	
9.	//MOD store inst 旁路
10.	input [4:0] MEM_wdest, // MEM 级要写回寄存器堆的目标地址号
11.	input [4:0] WB_wdest, // WB 级要写回寄存器堆的目标地址号
12.	input [31:0] MEM_wvalue,
13.	input [31:0] WB_wvalue,
14.	input MEM_wvalid,
15.	input WB_wvalid,
16.	
17.	//5 级流水新增
18.	input clk, // 时钟
19.	output [4:0] EXE_wdest, // EXE 级要写回寄存器堆的目标地址号
20.	output [31:0] EXE_wvalue,
21.	output EXE_wvalid,
22.	
23.	//展示 PC
24.	output [31:0] EXE_pc

```

25. );
26. //-----{ID->EXE 总线}begin
27. //EXE 需要用到的信息
28. wire multiply; //乘法
29. wire mthi; //MTHI
30. wire mtlo; //MTLO
31. wire [11:0] alu_control;
32. wire [31:0] alu_operand1;
33. wire [31:0] alu_operand2;
34.
35. //访存需要用到的 load/store 信息
36. wire [3:0] mem_control; //MEM 需要使用的控制信号
37. wire [31:0] store_data; //store 操作的存的数据
38.
39. //写回需要用到的信息
40. wire mfhi;
41. wire mflo;
42. wire mtc0;
43. wire mfc0;
44. wire [7 :0] cp0r_addr;
45. wire syscall; //syscall 和 eret 在写回级有特殊的操作
46. wire eret;
47. wire rf_wen; //写回的寄存器写使能
48. wire [4:0] rf_wdest; //写回的目的寄存器
49.
50. //MOD store inst 旁路
51. wire [4:0] store_rf_addr;
52. //pc
53. wire [31:0] pc;
54. assign {multiply,
55.         mthi,
56.         mtlo,
57.         alu_control,
58.         alu_operand1,
59.         alu_operand2,
60.         mem_control,
61.         store_data,
62.         store_rf_addr,
63.         mfhi,
64.         mflo,
65.         mtc0,
66.         mfc0,
67.         cp0r_addr,
68.         syscall,

```

```

69.             eret,
70.             rf_wen,
71.             rf_wdest,
72.             pc      } = ID_EXE_bus_r;
73. //-----{ID->EXE 总线}end
74.
75. //-----{ALU}begin
76.     wire [31:0] alu_result;
77.
78.     alu alu_module(
79.         .alu_control (alu_control ), // I, 12, ALU 控制信号
80.         .alu_src1     (alu_operand1), // I, 32, ALU 操作数 1
81.         .alu_src2     (alu_operand2), // I, 32, ALU 操作数 2
82.         .alu_result   (alu_result  ) // O, 32, ALU 结果
83.     );
84. //-----{ALU}end
85.
86. //-----{乘法器}begin
87.     wire      mult_begin;
88.     wire [63:0] product;
89.     wire      mult_end;
90.
91.     assign mult_begin = multiply & EXE_valid;
92.     multiply multiply_module (
93.         .clk      (clk      ),
94.         .mult_begin(mult_begin ),
95.         .mult_op1  (alu_operand1),
96.         .mult_op2  (alu_operand2),
97.         .product   (product   ),
98.         .mult_end  (mult_end  )
99.     );
100. //-----{乘法器}end
101.
102. //store 旁路
103.     wire wait_store_data;
104.     assign wait_store_data= (!store_rf_addr) &
105.         ((store_rf_addr==MEM_wdest & ~MEM_wvalid)|(sto
106. re_rf_addr==WB_wdest & WB_wvalid));
107.     wire [31:0] forward_store_data;
108.     assign forward_store_data = (store_rf_addr==MEM_wdest) ? MEM_wvalue:
109.         (store_rf_addr==WB_wdest ) ? WB_wvalue:
110.         store_data;
111. //-----{EXE 执行完成}begin

```



```

112.      //对于 ALU 操作，都是 1 拍可完成，
113.      //但对于乘法操作，需要多拍完成
114.      //MOD
115.      assign EXE_over = EXE_valid & (~multiply | mult_end) & ~wait_store_data;
116.      //-----{EXE 执行完成}end
117.
118.      //-----{EXE 模块的 dest 值}begin
119.      //只有在 EXE 模块有效时，其写回目的寄存器号才有意义
120.      assign EXE_wdest = rf_wdest & {5{EXE_valid}};
121.
122.      //-----{EXE 模块的 dest 值}end
123.
124.
125.      //-----{EXE->MEM 总线}begin
126.      wire [31:0] exe_result;    //在 exe 级能确定的最终写回结果
127.      wire [31:0] lo_result;
128.      wire        hi_write;
129.      wire        lo_write;
130.      //要写入 HI 的值放在 exe_result 里，包括 MULT 和 MTHI 指令，
131.      //要写入 LO 的值放在 lo_result 里，包括 MULT 和 MTLO 指令，
132.      assign exe_result = mthi      ? alu_operand1 :
133.                          mtc0      ? alu_operand2 :
134.                          multiply ? product[63:32] : alu_result;
135.      assign lo_result  = mtlo ? alu_operand1 : product[31:0];
136.      assign hi_write   = multiply | mthi;
137.      assign lo_write   = multiply | mtlo;
138.
139.      assign EXE_MEM_bus = {mem_control,forward_store_data, //load/store 信
                           息和 store 数据
140.                          exe_result,                      //exe 运算结果
141.                          lo_result,                       //乘法低 32 位结
                           果，新增
142.                          hi_write,lo_write,              //HI/LO 写使能，
                           新增
143.                          mfhi,mflo,                      //WB 需用的信号，
                           新增
144.                          mtc0,mfc0,cp0r_addr,syscall,eret,//WB 需用的信号，
                           新增
145.                          rf_wen,rf_wdest,                //WB 需用的信号
146.                          pc};                            //PC
147.      //-----{EXE->MEM 总线}end
148.
149.

```

```

150.    //MOD 旁路
151.    assign EXE_wvalue=exe_result;
152.    assign EXE_wvalid= ~(|mem_control) & EXE_over & rf_wen & ~mfc0;
153.
154.
155.    //-----{展示 EXE 模块的 PC 值}begin
156.        assign EXE_pc = pc;
157.    //-----{展示 EXE 模块的 PC 值}end
158.    endmodule

```

A. 6 mem. v

完整代码

```

1.    `timescale 1ns / 1ps
2.
3.    module mem(                                // 访存级
4.        input          clk,                    // 时钟
5.        input          MEM_valid,              // 访存级有效信号
6.        input  [153:0] EXE_MEM_bus_r, // EXE->MEM 总线
7.        input  [ 31:0] dm_rdata,              // 访存读数据
8.        output [ 31:0] dm_addr,                // 访存读写地址
9.        output reg [ 3:0] dm_wen,              // 访存写使能
10.       output reg [ 31:0] dm_wdata,           // 访存写数据
11.       output          MEM_over,              // MEM 模块执行完成
12.       output  [117:0] MEM_WB_bus,           // MEM->WB 总线
13.
14.       //5 级流水新增接口
15.       input          MEM_allow_in, // MEM 级允许下级进入
16.       output  [ 4:0] MEM_wdest,      // MEM 级要写回寄存器堆的目标地址号
17.       output  [31:0] MEM_wvalue,
18.       output          MEM_wvalid,
19.
20.       //展示 PC
21.       output  [ 31:0] MEM_pc
22.    );
23.    //-----{EXE->MEM 总线}begin
24.        //访存需要用到的 load/store 信息
25.        wire [3 :0] mem_control; //MEM 需要使用的控制信号
26.        wire [31:0] store_data; //store 操作的存的数据
27.
28.        //EXE 结果和 HI/LO 数据
29.        wire [31:0] exe_result;
30.        wire [31:0] lo_result;
31.        wire        hi_write;
32.        wire        lo_write;

```

```

33.
34.    //写回需要用到的信息
35.    wire mfhi;
36.    wire mflo;
37.    wire mtc0;
38.    wire mfc0;
39.    wire [7:0] cp0r_addr;
40.    wire      syscall;    //syscall 和 eret 在写回级有特殊的操作
41.    wire      eret;
42.    wire      rf_wen;    //写回的寄存器写使能
43.    wire [4:0] rf_wdest;  //写回的目的寄存器
44.
45.    //pc
46.    wire [31:0] pc;
47.    assign {mem_control,
48.           store_data,
49.           exe_result,
50.           lo_result,
51.           hi_write,
52.           lo_write,
53.           mfhi,
54.           mflo,
55.           mtc0,
56.           mfc0,
57.           cp0r_addr,
58.           syscall,
59.           eret,
60.           rf_wen,
61.           rf_wdest,
62.           pc      } = EXE_MEM_bus_r;
63.    //-----{EXE->MEM 总线}end
64.
65.    //-----{load/store 访存}begin
66.    wire inst_load;  //load 操作
67.    wire inst_store; //store 操作
68.    wire ls_word;    //load/store 为字节还是字,0:byte;1:word
69.    wire lb_sign;    //load 一字节为有符号 load
70.    assign {inst_load,inst_store,ls_word,lb_sign} = mem_control;
71.
72.    //访存读写地址
73.    assign dm_addr = exe_result;
74.
75.    //store 操作的写使能
76.    always @ (*)    // 内存写使能信号

```

```

77.         begin
78.             if (MEM_valid && inst_store) // 访存级有效时,且为 store 操作
79.                 begin
80.                     if (ls_word)
81.                         begin
82.                             dm_wen <= 4'b1111; // 存储字指令, 写使能全 1
83.                         end
84.                     else
85.                         begin // SB 指令, 需要依据地址底两位, 确定对应的写使能
86.                             case (dm_addr[1:0])
87.                                 2'b00 : dm_wen <= 4'b0001;
88.                                 2'b01 : dm_wen <= 4'b0010;
89.                                 2'b10 : dm_wen <= 4'b0100;
90.                                 2'b11 : dm_wen <= 4'b1000;
91.                                 default : dm_wen <= 4'b0000;
92.                             endcase
93.                         end
94.                     end
95.                 else
96.                     begin
97.                         dm_wen <= 4'b0000;
98.                     end
99.                 end
100.
101.         //store 操作的写数据
102.         always @ (*) // 对于 SB 指令, 需要依据地址底两位, 移动 store 的字节至对应位置
103.             begin
104.                 case (dm_addr[1:0])
105.                     2'b00 : dm_wdata <= store_data;
106.                     2'b01 : dm_wdata <= {16'd0, store_data[7:0], 8'd0};
107.                     2'b10 : dm_wdata <= {8'd0, store_data[7:0], 16'd0};
108.                     2'b11 : dm_wdata <= {store_data[7:0], 24'd0};
109.                     default : dm_wdata <= store_data;
110.                 endcase
111.             end
112.
113.         //load 读出的数据
114.         wire          load_sign;
115.         wire [31:0] load_result;
116.         assign load_sign = (dm_addr[1:0]==2'd0) ? dm_rdata[ 7] :
117.                             (dm_addr[1:0]==2'd1) ? dm_rdata[15] :
118.                             (dm_addr[1:0]==2'd2) ? dm_rdata[23] : dm_rdata[31]
;

```

```

119.         assign load_result[7:0] = (dm_addr[1:0]==2'd0) ? dm_rdata[ 7:0 ] :
120.                                     (dm_addr[1:0]==2'd1) ? dm_rdata[15:8 ] :
121.                                     (dm_addr[1:0]==2'd2) ? dm_rdata[23:16] :
122.                                     dm_rdata[31:24] ;
123.         assign load_result[31:8]= ls_word ? dm_rdata[31:8] : {24{lb_sign & lo
ad_sign}};
124. //-----{load/store 访存}end
125.
126. //-----{MEM 执行完成}begin
127.     //由于数据 RAM 为同步读写的,
128.     //故对 load 指令, 取数据时, 有一拍延时
129.     //即发地址的下一拍时钟才能得到 load 的数据
130.     //故 mem 在进行 load 操作时有需要两拍时间才能取到数据
131.     //而对其他操作, 则只需要一拍时间
132.
133.     assign MEM_over = MEM_valid;
134.     //如果数据 ram 为异步读的, 则 MEM_valid 即是 MEM_over 信号,
135.     //即 load 一拍完成
136. //-----{MEM 执行完成}end
137.
138. //-----{MEM 模块的 dest 值}begin
139.     //只有在 MEM 模块有效时, 其写回目的寄存器号才有意义
140.     assign MEM_wdest = rf_wdest & {5{MEM_valid}};
141. //-----{MEM 模块的 dest 值}end
142.
143. //-----{MEM->WB 总线}begin
144.     wire [31:0] mem_result; //MEM 传到 WB 的 result 为 load 结果或 EXE 结果
145.     assign mem_result = inst_load ? load_result : exe_result;
146.
147.     assign MEM_WB_bus = {rf_wen,rf_wdest,                                // WB 需要使用
的信号
148.                                mem_result,                                // 最终要写回寄
存器的数据
149.                                lo_result,                                // 乘法低 32 位
结果, 新增
150.                                hi_write,lo_write,                        // HI/LO 写使
能, 新增
151.                                mfhi,mflo,                                // WB 需要使用
的信号, 新增
152.                                mtc0,mfc0,cp0r_addr,syscall,eret, // WB 需要使用
的信号, 新增
153.                                pc};                                // PC 值
154. //-----{MEM->WB 总线}begin
155.     //MOD 旁路

```

```

156.         assign MEM_wvalue=mem_result;
157.         assign MEM_wvalid=rf_wen & MEM_over & ~mfc0;
158.
159.         //-----{展示 MEM 模块的 PC 值}begin
160.         assign MEM_pc = pc;
161.         //-----{展示 MEM 模块的 PC 值}end
162.     endmodule

```

A. 7 wb. v

完整代码

```

1.     `timescale 1ns / 1ps
2.
3.     `define EXC_ENTER_ADDR 32'd0      // Exception 入口地址,
4.                                         // 此处实现的 Exception 只有 SYSCALL
5.     module wb(                          // 写回级
6.         input          WB_valid,        // 写回级有效
7.         input  [117:0] MEM_WB_bus_r,    // MEM->WB 总线
8.         output         rf_wen,          // 寄存器写使能
9.         output  [ 4:0] rf_wdest,        // 寄存器写地址
10.        output  [31:0] rf_wdata,         // 寄存器写数据
11.        output         WB_over,          // WB 模块执行完成
12.
13.        //5 级流水新增接口
14.        input          clk,              // 时钟
15.        input          resetn,           // 复位信号, 低电平有效
16.        output  [32:0] exc_bus,          // Exception pc 总线
17.        output  [ 4:0] WB_wdest,         // WB 级要写回寄存器堆的目标地址号
18.        output         cancel,           // syscall 和 eret 到达写回级时会发出 cancel
        信号,
19.                                         // 取消已经取出的正在其他流水级执行的指令
20.        output         WB_wvalid,
21.        output  [31:0] WB_wvalue,
22.
23.        //展示 PC 和 HI/LO 值
24.        output  [31:0] WB_pc,
25.        output  [31:0] HI_data,
26.        output  [31:0] LO_data
27.    );
28.    //-----{MEM->WB 总线}begin
29.        //MEM 传来的 result
30.        wire [31:0] mem_result;
31.        //HI/LO 数据
32.        wire [31:0] lo_result;
33.        wire         hi_write;

```

```

34.      wire      lo_write;
35.
36.      //寄存器堆写使能和写地址
37.      wire wen;
38.      wire [4:0] wdest;
39.
40.      //写回需要用到的信息
41.      wire mfhi;
42.      wire mflo;
43.      wire mtc0;
44.      wire mfc0;
45.      wire [7 :0] cp0r_addr;
46.      wire      syscall;  //syscall 和 eret 在写回级有特殊的操作
47.      wire      eret;
48.
49.      //pc
50.      wire [31:0] pc;
51.      assign {wen,
52.             wdest,
53.             mem_result,
54.             lo_result,
55.             hi_write,
56.             lo_write,
57.             mfhi,
58.             mflo,
59.             mtc0,
60.             mfc0,
61.             cp0r_addr,
62.             syscall,
63.             eret,
64.             pc} = MEM_WB_bus_r;
65.      //-----{MEM->WB 总线}end
66.
67.      //-----{HI/LO 寄存器}begin
68.      //HI 用于存放乘法结果的高 32 位
69.      //LO 用于存放乘法结果的低 32 位
70.      reg [31:0] hi;
71.      reg [31:0] lo;
72.
73.      //要写入 HI 的数据存放在 mem_result 里
74.      always @(posedge clk)
75.      begin
76.          if (hi_write)
77.          begin

```

```

78.             hi <= mem_result;
79.         end
80.     end
81.     //要写入 LO 的数据存放在 lo_result 里
82.     always @(posedge clk)
83.     begin
84.         if (lo_write)
85.         begin
86.             lo <= lo_result;
87.         end
88.     end
89.     //-----{HI/LO 寄存器}end
90.
91.     //-----{cp0 寄存器}begin
92.     // cp0 寄存器即是协处理器 0 寄存器
93.     // 由于目前设计的 CPU 并不完备，所用到的 cp0 寄存器也很少
94.     // 故暂时只实现 STATUS(12.0),CAUSE(13.0),EPC(14.0)这三个
95.     // 每个 CP0 寄存器都是使用 5 位的 cp0 号
96.     wire [31:0] cp0r_status;
97.     wire [31:0] cp0r_cause;
98.     wire [31:0] cp0r_epc;
99.
100.    //写使能
101.    wire status_wen;
102.    //wire cause_wen;
103.    wire epc_wen;
104.    assign status_wen = mtc0 & (cp0r_addr=={5'd12,3'd0});
105.    assign epc_wen    = mtc0 & (cp0r_addr=={5'd14,3'd0});
106.
107.    //cp0 寄存器读
108.    wire [31:0] cp0r_rdata;
109.    assign cp0r_rdata = (cp0r_addr=={5'd12,3'd0}) ? cp0r_status :
110.                        (cp0r_addr=={5'd13,3'd0}) ? cp0r_cause :
111.                        (cp0r_addr=={5'd14,3'd0}) ? cp0r_epc : 32'd0;
112.
113.    //STATUS 寄存器
114.    //目前只实现 STATUS[1]位，即 EXL 域
115.    //EXL 域为软件可读写，故需要 statu_wen
116.    reg status_exl_r;
117.    assign cp0r_status = {30'd0,status_exl_r,1'b0};
118.    always @(posedge clk)
119.    begin
120.        if (!resetn || eret)
121.        begin

```



```

122.         status_exl_r <= 1'b0;
123.     end
124.     else if (syscall)
125.     begin
126.         status_exl_r <= 1'b1;
127.     end
128.     else if (status_wen)
129.     begin
130.         status_exl_r <= mem_result[1];
131.     end
132. end
133.
134. //CAUSE 寄存器
135. //目前只实现 CAUSE[6:2]位, 即 ExcCode 域, 存放 Exception 编码
136. //ExcCode 域为软件只读, 不可写, 故不需要 cause_wen
137. reg [4:0] cause_exc_code_r;
138. assign cp0r_cause = {25'd0, cause_exc_code_r, 2'd0};
139. always @(posedge clk)
140. begin
141.     if (syscall)
142.     begin
143.         cause_exc_code_r <= 5'd8;
144.     end
145. end
146.
147. //EPC 寄存器
148. //存放产生例外的地址
149. //EPC 整个域为软件可读写的, 故需要 epc_wen
150. reg [31:0] epc_r;
151. assign cp0r_epc = epc_r;
152. always @(posedge clk)
153. begin
154.     if (syscall)
155.     begin
156.         epc_r <= pc;
157.     end
158.     else if (epc_wen)
159.     begin
160.         epc_r <= mem_result;
161.     end
162. end
163.
164. //syscall 和 eret 发出的 cancel 信号
165. assign cancel = (syscall | eret) & WB_over;

```

```

166. //-----{cp0 寄存器}begin
167.
168. //-----{WB 执行完成}begin
169.     //WB 模块所有操作都可在一拍内完成
170.     //故 WB_valid 即是 WB_over 信号
171.     assign WB_over = WB_valid;
172. //-----{WB 执行完成}end
173.
174. //-----{WB->regfile 信号}begin
175.     assign rf_wen    = wen & WB_over;
176.     assign rf_wdest = wdest;
177.     assign rf_wdata = mfhi ? hi :
178.                     mflo ? lo :
179.                     mfc0 ? cp0r_rdata : mem_result;
180. //-----{WB->regfile 信号}end
181.     //旁路
182.     assign WB_wvalid=rf_wen;
183.     assign WB_wvalue=rf_wdata;
184. //-----{Exception pc 信号}begin
185.     wire      exc_valid;
186.     wire [31:0] exc_pc;
187.     assign exc_valid = (syscall | eret) & WB_valid;
188.     //eret 返回地址为 EPC 寄存器的值
189.     //SYSCALL 的 excPC 应该为{EBASE[31:10],10'h180},
190.     //但作为实验，先设置 EXC_ENTER_ADDR 为 0，方便测试程序的编写
191.     assign exc_pc = syscall ? `EXC_ENTER_ADDR : cp0r_epc;
192.
193.     assign exc_bus = {exc_valid,exc_pc};
194. //-----{Exception pc 信号}end
195.
196. //-----{WB 模块的 dest 值}begin
197.     //只有在 WB 模块有效时，其写回目的寄存器号才有意义
198.     assign WB_wdest = rf_wdest & {5{WB_valid}} ;
199. //-----{WB 模块的 dest 值}end
200.
201. //-----{展示 WB 模块的 PC 值和 HI/LO 寄存器的值}begin
202.     assign WB_pc = pc;
203.     assign HI_data = hi;
204.     assign LO_data = lo;
205. //-----{展示 WB 模块的 PC 值和 HI/LO 寄存器的值}end
206. endmodule

```

附录 B：实现的所有指令

指令类型	汇编指令	指令码	源操作数 1	源操作数 2	源操作数 3	目的寄存器	功能描述
R 型指令	addu rd,rs,rt	000000 rs rt rd 00000 100001	[rs]	[rt]		rd	GPR[rd]=GPR[rs]+GPR[rt]
	subu rd,rs,rt	000000 rs rt rd 00000 100011	[rs]	[rt]		rd	GPR[rd]=GPR[rs]-GPR[rt]
	slt rd,rs,rt	000000 rs rt rd 00000 101010	[rs]	[rt]		rd	GPR[rd]=(sign(GPR[rs])<sign(GPR[rt]))
	sltu rd,rs,rt	000000 rs rt rd 00000 101011	[rs]	[rt]		rd	GPR[rd]=(zero(GPR[rs])<zero(GPR[rt]))
	jalr rs	000000 rs 00000 11111 00000 001001	[rs]			31	GPR[31]=PC,PC=GPR[rs]
	jr rs	000000 rs 000000000 00000 001000	[rs]				PC=GPR[rs]
	and rd,rs,rt	000000 rs rt rd 00000 100100	[rs]	[rt]		rd	GPR[rd]=GPR[rs]&GPR[rt]
	nor rd,rs,rt	000000 rs rt rd 00000 100111	[rs]	[rt]		rd	GPR[rd]=~(GPR[rs] GPR[rt])
	or rd,rs,rt	000000 rs rt rd 00000 100101	[rs]	[rt]		rd	GPR[rd]=GPR[rs] GPR[rt]
	xor rd,rs,rt	000000 rs rt rd 00000 100110	[rs]	[rt]		rd	GPR[rd]=GPR[rs]^GPR[rt]
	sll rd,rt,shf	000000 00000 rt rd shf 000000		[rt]		rd	GPR[rd]=zero(GPR[rt])<<shf
	sllv rd,rt,rs	000000 rs rt rd 00000 000100	[rs]	[rt]		rd	GPR[rd]=zero(GPR[rt])<<(GPR[rs]%32)
	sra rd,rt,shf	000000 00000 rt rd shf 000011		[rt]		rd	GPR[rd]=sign(GPR[rt])>>shf
	srav rd,rt,rs	000000 rs rt rd 00000 000111	[rs]	[rt]		rd	GPR[rd]=sign(GPR[rt])>>(GPR[rs]%32)
	srl rd,rt,shf	000000 00000 rt rd shf 000010		[rt]		rd	GPR[rd]=zero(GPR[rt])>>shf
	srlv rd,rt,rs	000000 rs rt rd 00000 000110	[rs]	[rt]		rd	GPR[rd]=zero(GPR[rt])>>GPR[rs]
	mult rs,rt	000000 rs rt 0000000000 011000	[rs]	[rt]		(HI,LO)	(HI,LO)=sign(GPR[rs])*sign(GPR[rt])
	mflo rd	000000 0000000000 rd 00000 010010	[LO]			rd	GPR[rd]=[LO]
	mfhi rd	000000 0000000000 rd 00000 010000	[HI]			rd	GPR[rd]=[HI]
I 型指令	addiu rt,rs,imm	001001 rs rt imm	[rs]	sign_ext (imm)		rt	GPR[rt]=GPR[rs]+sign_ext (imm)
	slti rt,rs,imm	001010 rs rt imm	[rs]	sign_ext (imm)		rt	GPR[rt]=(sign(GPR[rs])<sign_ext (imm))
	sltiu rt,rs,imm	001011 rs rt imm	[rs]	sign_ext (imm)		rt	GPR[rt]=(zero(GPR[rs])<sign_ext (imm))
	beq rs,rt,offset	000100 rs rt offset	[rs]	[rt]			if GPR[rs]=GPR[rt] then PC= next_pc+ sign_ext (offset)<<2
	bgez rs,offset	000001 rs 00001 offset	[rs]				if GPR[rs]≥0 then PC= next_pc + sign_ext (offset)<<2
	bgtz rs,offset	000111 rs 00000 offset	[rs]				if GPR[rs]>0

						then PC= next_pc + sign_ext (offset)<<2
	blez rs,offset	000110 rs 00000 offset	[rs]			if GPR[rs]≤0 then PC= next_pc + sign_ext (offset)<<2
	bltz rs,offset	000001 rs 00000 offset	[rs]			if GPR[rs]<0 then PC= next_pc + sign_ext (offset)<<2
	bne rs,rt,offset	000101 rs rt offset	[rs]	[rt]		if GPR[rs]≠GPR[rt] then PC= next_pc + sign_ext (offset)<<2
	lw rt,offset(b)	100011 b rt offset	[b]	sign_ext (offset)	rt	GPR[rt]=Mem[GPR[b]+sign_ext (offset)]
	sw rt,offset(b)	101011 b rt offset	[b]	sign_ext (offset)	[rt]	Mem[GPR[b]+sign_ext (offset)]=GPR[rt]
	lb rt,offset(b)	100000 b rt offset	[b]	sign_ext (offset)	rt	GPR[rt]=sign(Mem[GPR[b]+sign_ext (offset)])
	lbu rt,offset(b)	100100 b rt offset	[b]	sign_ext (offset)	rt	GPR[rt]=zero(Mem[GPR[b]+sign_ext (offset)])
	sb rt,offset(b)	101000 b rt offset	[b]	sign_ext (offset)	[rt]	Mem[GPR[b]+sign_ext (offset)]=GPR[rt]
	andi rt,rs,imm	001100 rs rt imm	[rs]	zero_ext (imm)	rt	GPR[rt]=GPR[rs]&zero_ext (imm)
	lui rt,imm	001111 00000 rt imm		{imm, 16'd0}	rt	GPR[rt]= {imm, 16'd0}
	ori rt,rs,imm	001101 rs rt imm	[rs]	zero_ext (imm)	rt	GPR[rt]=GPR[rs] zero_ext (imm)
	xori rt,rs,imm	001110 rs rt imm	[rs]	zero_ext (imm)	rt	GPR[rt]=GPR[rs]^zero_ext (imm)
J 型 指令	j target	000010 target				[PC]={next_pc[31:28],target<<2}
	jal target	000011 target				GPR[31]=PC,PC={next_pc[31:28],target<<2}
cp0 指 令	mfc0 rt,cs	010000 00000 rt cs 00000000 sel	CPR[cs.sel]		rt	GPR[rt]=CPR[cs.sel]
	mtc0 rt,cd	010000 00100 rt cd 00000000 sel		GPR[rt]	CPR[cd.sel]	CPR[cd.sel]=GPR[rt]
	syscall	000000 code 001100				CPR[14.0]=PC,CPR[13.0][6:2]=01000,C PR[12.0][1]=1,PC=CPR[15.1]+0x180 并 跳转
	Eret	010000 1000000000000000000000 011000				CPR[12.0][1]=0,PC=CPR[14.0]并跳转

注：分支跳转指令参与运算的不在是当前 PC 值，而是 next_pc,即当前 PC+4，即延迟槽指令的 PC。

附录 C：验证程序代码

指令地址	汇编指令		结果描述	机器指令的机器码	
				16 进制	二进制
Exception 入口地址，在 SYSCALL 指令执行后进入此处执行					
00H	sw	\$1,#0(\$0)	Mem[0000_0000H] = 0000_0008H	AC010000	1010_1100_0000_0001_0000_0000_0000_0000
04H	sw	\$2,#4(\$0)	Mem[0000_0004H] = 0000_0010H	AC020004	1010_1100_0000_0010_0000_0000_0000_0100
08H	sw	\$3,#8(\$0)	Mem[0000_0008H] = 0000_0011H	AC030008	1010_1100_0000_0011_0000_0000_0000_1000
0CH	sw	\$4,#12(\$0)	Mem[0000_000CH] = 0000_0004H	AC04000C	1010_1100_0000_0100_0000_0000_0000_1100
10H	sw	\$5,#16(\$0)	Mem[0000_0010H] = 0000_000DH	AC050010	1010_1100_0000_0101_0000_0000_0001_0000
14H	sw	\$6,#24(\$0)	Mem[0000_0018H] = FFFF_FFE2H	AC060018	1010_1100_0000_0110_0000_0000_0001_1000
18H	sw	\$7,#112(\$0)	Mem[0000_0070H] = FFFF_FFF3H	AC070070	1010_1100_0000_0111_0000_0000_0111_0000
1CH	sw	\$25,#116(\$0)	Mem[0000_0074H] = 0000_0001H	AC190074	1010_1100_0001_1001_0000_0000_0111_0100
20H	sw	\$13,#24(\$0)	Mem[0000_0078H] = 0000_0000H	AC0D0078	1010_1100_0000_1101_0000_0000_0111_1000
24H	mfc0	\$1, cp0(14.0)	[\$1] = 0000_0104H	40017000	0100_0000_0000_0001_0111_0000_0000_0000
28H	addiu	\$1,\$1,#4	[\$1] = 0000_0108H	24210004	0010_0100_0010_0001_0000_0000_0000_0100
2CH	mtc0	\$1, cp0(14.0)	cp0(14.0) = 0000_0108H	40817000	0100_0000_1000_0001_0111_0000_0000_0000
30H	eret		返回 108H	42000018	0100_0010_0000_0000_0000_0000_0001_1000
CPU 复位地址 0000_0034H					
34H	addiu	\$1, \$0,#1	[\$1] = 0000_0001H	24010001	0010_0100_0000_0001_0000_0000_0000_0001
38H	sll	\$2, \$1,#4	[\$2] = 0000_0010H	00011100	0000_0000_0000_0001_0001_0001_0000_0000
3CH	addu	\$3, \$2,\$1	[\$3] = 0000_0011H	00411821	0000_0000_0100_0001_0001_1000_0010_0001
40H	srl	\$4, \$2,#2	[\$4] = 0000_0004H	00022082	0000_0000_0000_0010_0010_0000_1000_0010
44H	slti	\$25,\$4,#5	[\$25] = 0000_0001H	28990005	0010_1000_1001_1001_0000_0000_0000_0101
48H	bgez	\$25,#14	跳转到 84H	0721000E	0000_0111_0010_0001_0000_0000_0000_1110
4CH	subu	\$5, \$3,\$4	[\$5] = 0000_000DH	00642823	0000_0000_0110_0100_0010_1000_0010_0011
50H	sw	\$5, #20(\$0)	Mem[0000_0014H] = 0000_000DH	AC050014	1010_1100_0000_0101_0000_0000_0001_0100
54H	nor	\$6, \$5,\$2	[\$6] = FFFF_FFE2H	00A23027	0000_0000_1010_0010_0011_0000_0010_0111
58H	or	\$7, \$6,\$3	[\$7] = FFFF_FFF3H	00C33825	0000_0000_1100_0011_0011_1000_0010_0101

5CH	xor	\$8, \$7, \$6	[\$8] = 0000_0011H	00E64026	0000_0000_1110_0110_0100_0000_0010_0110
60H	beq	\$8, \$3, #2	跳转到 6CH	11030002	0001_0001_0000_0011_0000_0000_0000_0010
64H	sw	\$8, #28(\$0)	Mem[0000_001CH] = 0000_0011H	AC08001C	1010_1100_0000_1000_0000_0000_0001_1100
68H	slt	\$9, \$1, \$2	不执行	0022482A	0000_0000_0010_0010_0100_1000_0010_1010
6CH	lw	\$10, #28(\$0)	[\$10] = 0000_0011H	8C0A001C	1000_1100_0000_1010_0000_0000_0001_1100
70H	bne	\$10, \$5, #2	跳转到 7CH	15450002	0001_0101_0100_0101_0000_0000_0000_0010
74H	and	\$11, \$2, \$1	[\$11] = 0000_0000H	00415824	0000_0000_0100_0001_0101_1000_0010_0100
78H	sw	\$11, #28(\$0)	不执行	AC0B001C	1010_1100_0000_1011_0000_0000_0001_1100
7CH	jal	#38	跳转到 98H, [\$31] = 0000_0084H	0C000026	0000_1100_0000_0000_0000_0000_0010_0110
80H	sw	\$4, #16(\$0)	Mem[0000_0010H] = 0000_0004H	AC040010	1010_1100_0000_0100_0000_0000_0001_0000
84H	lui	\$12, #12	[\$12] = 000C_0000H	3C0C000C	0011_1100_0000_1100_0000_0000_0000_1100
88H	srav	\$26, \$12, \$2	[\$26] = 0000_000CH	004CD007	0000_0000_0100_1100_1101_0000_0000_0111
8CH	addiu	\$27, \$26, #68	[\$27] = 0000_0050H	275B0044	0010_0111_0101_1011_0000_0000_0100_0100
90H	jalr	\$27	跳转到 50H, [\$31] = 0000_0098H	0360F809	0000_0011_0110_0000_1111_1000_0000_1001
94H	addiu	\$1, \$0, #8	[\$1] = 0000_0008H	24010008	0010_0100_0000_0001_0000_0000_0000_1000
98H	sb	\$26, #5(\$3)	MEM[0000_0014H] = 000C_000DH	A07A0005	1010_0000_0111_1010_0000_0000_0000_0101
9CH	sltu	\$13, \$10, \$3	[\$13] = 0000_0000H	0143682B	0000_0001_0100_0011_0110_1000_0010_1011
A0H	bgtz	\$13, #2	不跳转	1DA00002	0001_1101_1010_0000_0000_0000_0000_0010
A4H	sllv	\$14, \$6, \$4	[\$14] = FFFF_FE20H	00867004	0000_0000_1000_0110_0111_0000_0000_0100
A8H	sra	\$15, \$14, #2	[\$15] = FFFF_FF88H	000E7883	0000_0000_0000_1110_0111_1000_1000_0011
ACH	srlv	\$16, \$15, \$1	[\$16] = 00FF_FFFFH	002F8006	0000_0000_0010_1111_1000_0000_0000_0110
B0H	blez	\$16, #7	不跳转	1A000007	0001_1010_0000_0000_0000_0000_0000_0111
B4H	srav	\$16, \$15, \$1	[\$16] = FFFF_FFFFH	002F8007	0000_0000_0010_1111_1000_0000_0000_0111
B8H	bltz	\$16, #6	跳转到 D4H	06000006	0000_0110_0000_0000_0000_0000_0000_0110
BCH	sll	\$11, \$26, #4	[\$11] = 0000_00C0H	001A5900	0000_0000_0001_1010_0101_1001_0000_0000
C0H	lw	\$28, #3(\$10)	[\$28] = 000C_000DH / 000C_880DH	8D5C0003	1000_1101_0101_1100_0000_0000_0000_0011
C4H	bne	\$28, \$29, #7	不跳转/跳转 E4H	179D0007	0001_0111_1001_1101_0000_0000_0000_0111
C8H	sb	\$15, #8(\$5)	Mem[0000_0014H] = 000C_8800H	A0AF0008	1010_0000_1010_1111_0000_0000_0000_1000
CCH	lb	\$18, #8(\$5)	[\$18] = FFFF_FF88H	80B20008	1000_0000_1011_0010_0000_0000_0000_1000
D0H	lbu	\$19, #8(\$5)	[\$19] = 0000_0088H	90B30008	1001_0000_1011_0011_0000_0000_0000_1000
D4H	sltiu	\$24, \$15, #0xFFFF	[\$24] = 0000_0001H	2DF8FFFF	0010_1101_1111_1000_1111_1111_1111_1111
D8H	or	\$29, \$12, \$5	[\$29] = 000C_000DH	0185E825	0000_0001_1000_0101_1110_1000_0010_0101

DCH	jr \$11	跳转到 C0H	01600008	0000_0001_0110_0000_0000_0000_1000
E0H	andi \$20,\$15,#0xFFFF	[\$20] = 0000_FF88H	31F4FFFF	0011_0001_1111_0100_1111_1111_1111
E4H	ori \$21,\$15,#0xFFFF	[\$21] = FFFF_FFFFH	35F5FFFF	0011_0101_1111_0101_1111_1111_1111
E8H	xori \$22,\$15,#0xFFFF	[\$22] = FFFF_0077H	39F6FFFF	0011_1001_1111_0110_1111_1111_1111
ECH	mult \$12,\$29	[HI] = 0000_0090H, [LO] = 009C_0000H	019D0018	0000_0001_1001_1101_0000_0000_1000
F0H	mflo \$23	[\$23] = 009C_0000H	0000B812	0000_0000_0000_0000_1011_1000_0001_0010
F4H	mfhi \$30	[\$30] = 0000_0090H	0000F010	0000_0000_0000_0000_1111_0000_0001_0000
F8H	mtlo \$26	[LO] = 0000_000CH	03400013	0000_0011_0100_0000_0000_0000_0001_0011
FCH	mthi \$27	[HI] = 0000_0050H	03600011	0000_0011_0110_0000_0000_0000_0001_0001
100H	mtc0 \$0,c14	cp0[14.0] = 0000_0000H	40807000	0100_0000_1000_0000_0111_0000_0000_0000
104H	syscall	cp0[14.0] =0000_0104H, cp0[13.0][6..2] =01000B, cp0[12.0][1]=1, 跳转 Exception 入口 地址, 00H	0000000C	0000_0000_0000_0000_0000_0000_0000_1100
108H	mfc0 \$2, cp0(14.0)	[\$2] = 0000_0108H	40027000	0100_0000_0000_0010_0111_0000_0000_0000
10CH	mfc0 \$3, cp0(13.0)	[\$3] = 0000_0020H	40036800	0100_0000_0000_0011_0110_1000_0000_0000
110H	mfc0 \$4, cp0(12.0)	[\$4] = 0000_0000H	40046000	0100_0000_0000_0100_0110_0000_0000_0000
114H	addiu \$1, \$0,#32	[\$1] = 0000_0020H	24010020	0010_0100_0000_0001_0000_0000_0010_0000
118H	slt \$17,\$15,\$14	[\$17] = 0000_0000H	01EE882A	0000_0001_1110_1110_1000_1000_0010_1010
11CH	lui \$17,#1234H	[\$17] = 1234_0000H	3C111234	0011_1100_0001_0001_0001_0010_0011_0100
120H	addiu \$17,\$17,#5678	[\$17] = 1234_5678H	26315678	0010_0110_0011_0001_0101_0110_0111_1000
124H	sw \$17,#0(\$1)	Mem[0000_0020H] =1234_5678H / Mem[0000_0024H] =2345_6780H / Mem[0000_0028H] =3456_7800H / Mem[0000_002CH] =4567_8000H / Mem[0000_0030H]	AC310000	1010_1100_0011_0001_0000_0000_0000_0000

		=5678_0000H / Mem[0000_0034H] =6780_0000H / Mem[0000_0038H] =7800_0000H		
128H	sll \$17,\$17,#4	[\$17] = 2345_6780H / 3456_7800H / 4567_8000H / 5678_0000H / 6780_0000H / 7800_0000H / 8000_0000H	00118900	0000_0000_0001_0001_1000_1001_0000_0000
12CH	bgtz \$17,#-3	跳转到 124H /跳转/ 跳转/跳转/跳转/跳转 124H/不跳转	1E20FFFD	0001_1110_0010_0000_1111_1111_1111_1101
130H	addiu \$1,\$1,#4	[\$1] = 0000_0024H / 0000_0028H / 0000_002CH / 0000_0030H / 0000_0034H / 0000_0038H / 0000_003CH	24210004	0010_0100_0010_0001_0000_0000_0000_0100
134H	addiu \$2,\$0,#60	[\$2] = 0000_003CH	2402003C	0010_0100_0000_0010_0000_0000_0011_1100
138H	lw \$17,#-28(\$1)	[\$17] = 1234_5678H	8C31FFE4	1000_1100_0011_0001_1111_1111_1110_0100
13CH	srl \$17,\$17,#4	[\$17] = 0123_4567H / 0012_3456H / 0001_2345H / 0000_1234H / 0000_0123H / 0000_0012H / 0000_0001H /	00118902	0000_0000_0001_0001_1000_1001_0000_0010

		0000_0000H		
140H	sw \$17,#0(\$2)	Mem[0000_003CH] =0123_4567H / Mem [0000_0040H] =0012_3456H / Mem [0000_0044H] =0001_2345H / Mem [0000_0048H] =0000_1234H / Mem [0000_004CH] =0000_0123H / Mem [0000_0050H] =0000_0012H / Mem [0000_0054H] =0000_0001H / Mem [0000_0058H] =0000_0000H	AC510000	1010_1100_0101_0001_0000_0000_0000_0000
144H	bne \$17,\$0,#-3	跳转到 13CH /跳转/ 跳转/跳转/跳转/跳转 /跳转 13CH/不跳转	1620FFFD	0001_0110_0010_0000_1111_1111_1111_1101
148H	addiu \$2,\$2,#4	[\$2] = 0000_0040H / 0000_0044H / 0000_0048H / 0000_004CH / 0000_0050H / 0000_0054H / 0000_0058H / 0000_005CH	24420004	0010_0100_0100_0010_0000_0000_0000_0100
14CH	addiu \$6,\$0,#68	[\$6] = 0000_0044H	24060044	0010_0100_0000_0110_0000_0000_0100_0100
150H	addiu \$7,\$0,#100	[\$7] =0000_0064H	24070064	0010_0100_0000_0111_0000_0000_0110_0100

154H	lw	\$3,#-28(\$1)	[\$3] = 1234_5678H / 2345_6780H / 3456_7800H / 4567_8000H / 5678_0000H / 6780_0000H / 7800_0000H	8C23FFE4	1000_1100_0010_0011_1111_1111_1110_0100
158H	lw	\$4,#-4(\$2)	[\$4] = 0000_0000H / 0000_0001H / 0000_0012H / 0000_0123H / 0000_1234H / 0001_2345H / 0012_3456H	8C44FFFC	1000_1100_0100_0100_1111_1111_1111_1100
15CH	or	\$5,\$3,\$4	[\$5] = 1234_5678H / 2345_6781H / 3456_7812H / 4567_8123H / 5678_1234H / 6781_2345H / 7812_3456H	00642825	0000_0000_0110_0100_0010_1000_0010_0101
160H	sb	\$5,#0(\$7)	Mem[0000_0064H] =0000_0078H / Mem [0000_0064H] =0000_8178H / Mem[0000_0064H] =0012_8178H	A0E50000	1010_0000_1110_0101_0000_0000_0000_0000

		/ Mem[0000_0064H] =2312_8178H / Mem[0000_0068H] =0000_0034H / Mem[0000_0068H] =0000_4534H / Mem[0000_0068H] =0056_4534H		
164H	addiu \$7,\$7,#1	[\$7] = 0000_0065H / 0000_0066H / 0000_0067H / 0000_0068H / 0000_0069H / 0000_006AH / 0000_006BH	24E70001	0010_0100_1110_0111_0000_0000_0000_0001
168H	addiu \$1,\$1,#4	[\$1] = 0000_0040H / 0000_0044H / 0000_0048H / 0000_004CH / 0000_0050H / 0000_0054H / 0000_0058H	24210004	0010_0100_0010_0001_0000_0000_0000_0100
16CH	bne \$2,\$6,#-7	跳转到 154H/跳转/跳转/跳转/跳转/跳转/跳转/跳转/154H/不跳转	1446FFF9	0001_0100_0100_0110_1111_1111_1111_1001
170H	ADDIU \$2,\$2,#-4	[\$2] = 0000_0058H / 0000_0054H / 0000_0050H /	2442FFFC	0010_0100_0100_0010_1111_1111_1111_1100

		0000_004CH / 0000_0048H / 0000_0044H / 0000_0040H		
174H	addiu \$9,\$0,#100	[\$9] = 0000_0064H	24090064	0010_0100_0000_1001_0000_0000_0110_0100
178H	lbu \$9,#3(\$9)	[\$9] = 0000_0023H	91290003	1001_0001_0010_1001_0000_0000_0000_0011
17CH	addiu \$13,\$0,#104	[\$13] = 0000_0068H	240D0068	0010_0100_0000_1101_0000_0000_0110_1000
180H	lw \$13,#0(\$13)	[\$13] = 0056_4534H	8DAD0000	1000_1101_1010_1101_0000_0000_0000_0000
184H	ll \$9,\$9,#24	[\$9] = 2300_0000H	00094E00	0000_0000_0000_1001_0100_1110_0000_0000
188H	xori \$13,\$13,#9	[\$13] = 0056_453DH	39AD0009	0011_1001_1010_1101_0000_0000_0000_1001
18CH	sw \$13,#1(\$7)	MEM[0000_006CH]= 0056_453DH	ACED0001	1010_1100_1110_1101_0000_0000_0000_0001
190H	lw \$1,#0(\$0)	[\$1] = 0000_0008H	8C010000	1000_1100_0000_0001_0000_0000_0000_0000
194H	lw \$2,#4(\$0)	[\$2] = 0000_0010H	8C020004	1000_1100_0000_0010_0000_0000_0000_0100
198H	lw \$3,#8(\$0)	[\$3] = 0000_0011H	8C030008	1000_1100_0000_0011_0000_0000_0000_1000
19CH	lw \$4,#12(\$0)	[\$4] = 0000_0004H	8C04000C	1000_1100_0000_0100_0000_0000_0000_1100
1A0H	lw \$5,#16(\$0)	[\$5] = 0000_000DH	8C050010	1000_1100_0000_0101_0000_0000_0001_0000
1A4H	lw \$6,#24(\$0)	[\$6] = FFFF_FFE2H	8C060018	1000_1100_0000_0110_0000_0000_0001_1000
1A8H	lw \$7,#112(\$0)	[\$7] = FFFF_FFF3H	8C070070	1000_1100_0000_0111_0000_0000_0111_0000
1ACH	lw \$25,#116(\$0)	[\$25] = 0000_0001H	8C190074	1000_1100_0001_1001_0000_0000_0111_0100
1B0H	lw \$13,#120(\$0)	[\$13] = 0000_0000H	8C0D0078	1000_1100_0000_1101_0000_0000_0111_1000
1B4H	j #34H	跳转到 34H	0800000D	0000_1000_0000_0000_0000_0000_0000_1101