



**计算机组成原理 课 程 设 计**

团队实验报告

|  |  |
| --- | --- |
| 学 院 | 计算机学院 |
| 专 业 | 计算机科学与技术 |
| 指导老师 | 王娟 |
| 组 长 | 张驰 |
| 组 员 | 魏慧聪、王浚哲、曹哲瑀 |
| 组长联系方式 | zhangchi5675@163.com |

二O二二年 九月

目 录

[第一章 项目简述 1](#_Toc113136034)

[第二章 组员分工（团队报告，个人报告不需要） 1](#_Toc113136035)

[第三章 设计目的 1](#_Toc113136036)

[第四章 设计环境 1](#_Toc113136037)

[第五章 设计原理及内容 2](#_Toc113136038)

[5.1 CPU整体架构 2](#_Toc113136039)

[5.1.1 数据通路 3](#_Toc113136040)

[5.1.2 代码模块框架 3](#_Toc113136041)

[5.1.3 控制逻辑 4](#_Toc113136042)

[5.2 流水线冒险问题以及解决方案 5](#_Toc113136043)

[第六章 设计与实现 5](#_Toc113136044)

[第七章 测试 10](#_Toc113136045)

[第八章 问题及解决方法 14](#_Toc113136046)

[第九章 心得体会及总结 15](#_Toc113136047)

[第十章 参考文献有价值的资源推荐 15](#_Toc113136048)

# 项目简述

在本次团队项目中，本团队共计4人，队长张驰，队员魏慧聪、王浚哲、曹哲瑀。在本次团队项目中，我们共同实现了五级流水线CPU（包含从存储器中读取指令，指令译码，执行操作以及计算地址，数据存取，写回）五级阶段，共支持17条MIPS指令，包含冒险冲突处理（数据相关和控制相关的处理均进行了验证），结合汇编与接口设计实验，设计测试用例，进行了仿真测试，测试结果正确，并下板验证成功。

# 组员分工

小组分工见表2.1 小组分工。

|  |  |
| --- | --- |
| **组员** | **分工任务** |
| 组长：张驰 | * 测试代码开发，仿真测试以及代码调试修改 |
| 组员：魏慧聪 | * 运算逻辑开发 |
| 组员：曹哲瑀 | * 流水线开发 |
| 组员：王浚哲 | * 控制逻辑开发，文档ppt编写 |

表2.1 小组分工

# 设计目的

实现支持MIPS指令子集的流水线CPU，并且在仿真模拟中验证成功，仿真正确。

# 设计环境

|  |  |
| --- | --- |
| 操作系统 | Windows10 |
| 编程语言 | Verilog HDL |
| EDA工具 | Vivado2019.2 |
| 汇编语言 | MIPS |
| 汇编程序编辑器 | mars4.5 |

请标注版本号

# 设计原理及内容

## CPU整体架构

本流水线CPU覆盖17条指令(见表5.1-1)，下面将从数据通路和控制逻辑阐述设计思路。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 指令 | opcode | function | 功能 |
| 0 | NOP | X | X | 空指令，用于处理流水线冒险 |
| 1 | ADDI | 1000 | X | rd=rs+im |
| 2 | ADDIU | 1001 | X | rd=rs+im（无符号数） |
| 3 | ADD | 0 | 100000 | rd=rs+rt |
| 4 | ADDU | 0 | 100001 | rd=rs+rt (无符号数) |
| 5 | SUB | 0 | 100010 | rd=rs-rt |
| 6 | SUBU | 0 | 100011 | rd=rs-rt (无符号数) |
| 7 | AND | 0 | 100100 | rd=rs&rt |
| 8 | ANDI | 1100 | X | rd=rs&im |
| 9 | OR | 0 | 100101 | rd=rs|rt |
| 10 | ORI | 1101 | X | rd=rs|im |
| 11 | XOR | 0 | 100110 | rd=rs xor rd |
| 12 | XORI | 1110 | X | rd=rs xor im |
| 13 | NOR | 0 | 100111 | rd=!(rs|rt) |
| 14 | LW | 100011 | X | LW rt, offset(base) |
| 15 | SW | 101011 | X | SW rt, offset(base) |
| 16 | LUI | 1111 | X | rt=im\*65536 |
| 17 | J | 10 | X | PC={(PC+4)[31,28],addr,00} |

表5.1-1 17条指令

### 数据通路

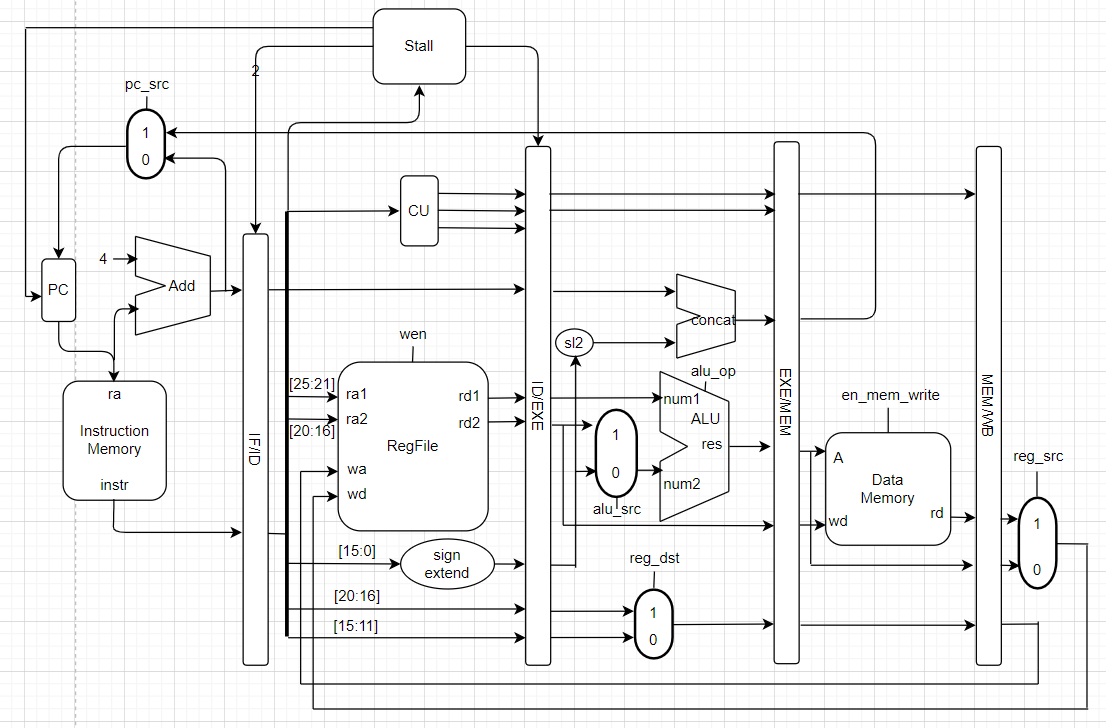


图5.1.1-1 5级流水线CPU数据通路

### 代码模块框架

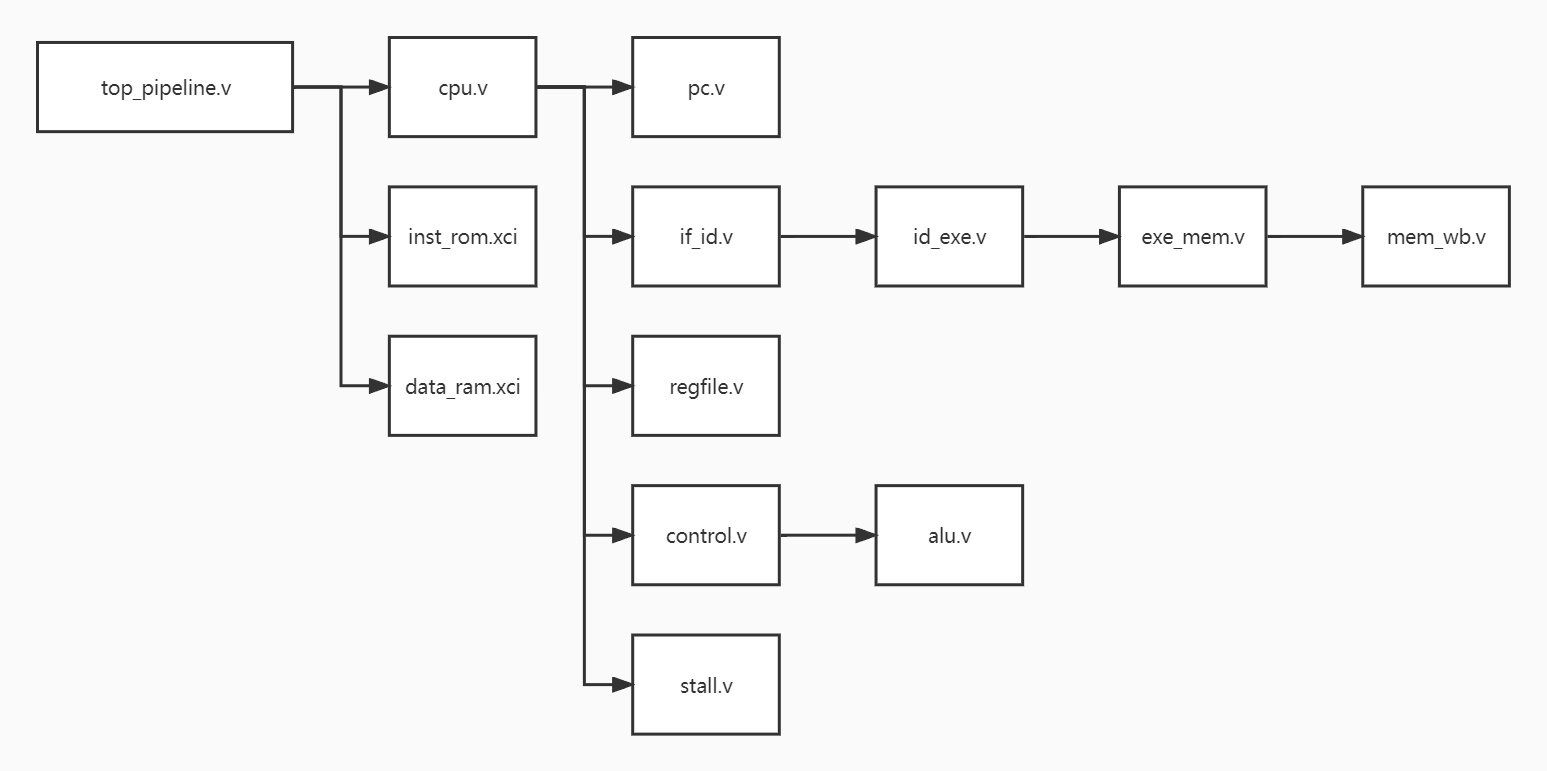


图5.1.2-1 代码模块框架图

### 控制逻辑

**实现控制逻辑采用方式：**

control模块、alu模块使用组合逻辑实现，其余模块使用时序逻辑实现。

**控制信号列表：**

**en\_reg\_write**：寄存器组写使能，0-不能写，1-能写；

**en\_mem\_write**：数据存储器写使能，0-不能写，1-能写；

**alu\_src**：操作数2来源，0-来自扩展后的立即数，1-来自寄存器组读出结果2；

**alu\_op**：alu中执行的操作；

1. null；
2. add；
3. addu；
4. sub；
5. subu；
6. and；
7. or；
8. xor；
9. nor；
10. lui；

**reg\_dst**：向寄存器写的地址来源，0-rd[15:11]，1-rt[20:16]；

**reg\_src**：向寄存器写的数据来源，0-alu运算结果，1-数据存储器读出结果；

**pc\_src**：下一个pc地址来源；

1. 延迟槽；
2. 延迟槽前4位与指令后26位左移两位后的拼接；

**sign\_extend\_type**：符号扩展方式，0-无符号扩展，1-带符号扩展；

**instr[5:0] :** 存储的是指令的前五位码，标注了是哪种类型的指令

**func[5:0] :** 针对某些计算类型的指令，在instr[5:0] = 5’b00000时， 根据func[5:0]判断执行的是哪条指令。

## 流水线冒险问题以及解决方案

流水线冒险问题包括结构相关，数据相关和控制相关。

1. 结构相关：结构相关，又称资源相关，是指多条指令进入流水线后在同一机器时钟周期内争用同一个功能部件所发生的冲突。在冯诺依曼结构中，取指和访存可能同时争用同一个功能部件，所以在该五级流水线CPU中，使用哈佛结构，使用一个INST\_ROM存储要运行的指令，和一个DATA\_RAM存储内存空间的数据，从而解决结构相关的问题。
2. 数据相关：数据相关，指在流水线执行的几条指令中，一条指令依赖于前面指令的结果。我们的流水线CPU主要对于寄存器写后读的情况，进行了处理。在译码阶段，我们设计了stall模块用于处理写后读的情况。在程序中，包含三个缓冲寄存器buffer1，buffer2，buffer3来存储上一条，上两条，上三条指令的写回地址。若当前指令需要读的地址和这三个写回地址中的至少一个相同，则表明会出现数据相关。此时，发送pause信号给IF-ID和ID-EXE模块，在接收到pause指令时，插入空指令，让这些模块停止向后流水，也发送pause信号给pc模块，让pc模块停止自增，不再取下一条指令。
3. 控制相关：控制相关，指的是流水线中的分支指令或者其他需要改写PC的指令造成的相关。在本次设计的流水线CPU中，主要针对与J指令。为了避免跳转指令之后的指令被多执行，若当前指令的操作码是6'b000010（当前指令为J指令），则发生跳转。虽然，后一条指令已经在被取出，但是不应执行该指令，所以发送drop信号给IF-ID模块，让IF-ID冲刷流水线，用NOP指令代替不应该被执行的指令。

# 设计与实现

共设计并实现了10个模块，分别为

1. cpu：顶层模块，负责各个模块的连接，信号的选择，PC地址的运算；
2. regfile：通用寄存器堆，共32个寄存器；
3. control：控制核心模块，根据指令的opcode和function字段，确定指令的类型及功能，并产生相应的控制信号，如alu\_op、alu\_src等；
4. alu：计算核心模块，进行算术运算，如加法、减法、拼接(LUI)、或运算等；
5. pc：存储当前指令地址
6. stall：暂停控制模块，存储之前的指令，并根据当前指令判断并添加NOP空周期；
7. if\_id：与其他三个模块都是指令流水寄存器，存储下游流水阶段所需要的信号。
8. id\_exe
9. exe\_mem
10. mem\_wb

此外，对于部分指令所需的符号扩展，由于所需逻辑较为简单，我们在CPU模块中使用组合电路进行实现。

其接口定义如下

1. cpu(
2. input clk,
3. input rst,
4. input [31:0] imem\_rd,
5. input [31:0] dmem\_rd,
7. output dmem\_wen,
8. output [31:0] dmem\_wd,
9. output [31:0] dmem\_ra,
10. output [31:0] imem\_ra
11. );
12. regfile(
13. input clk,
14. input rst,
15. input [4:0] ra1,//read address1
16. input [4:0] ra2,//read address2
17. input en\_write//enable write
18. input [31:0] wd,//write data
19. input [4:0] wa,//write address
21. output [31:0] rd1,//read data1
22. output [31:0] rd2//read data2
23. );
24. control(
25. input [5:0] instr,
26. input [5:0] func,
28. output [`ALU\_OP\_LEN-1:0] alu\_op,
29. output alu\_src,
30. output reg\_dst,
31. output reg\_src,
32. output pc\_src,
33. output en\_reg\_write,
34. output en\_mem\_write,
35. output sign\_extend\_type
36. );
37. alu(
38. input wire[31:0] num1,
39. input wire[31:0] num2,
40. input wire[`ALU\_OP\_LEN - 1:0] alu\_op,
41. output wire zero,
42. output wire[31:0] alu\_res,
43. output wire error
44. );
45. pc(
46. input rst,
47. input clk,
48. input pause,
49. input [31:0] pc\_in,
51. output [31:0] pc\_out
52. );
53. stall(
54. input clk,
55. input rst,
56. input [31:0] instr,
58. output pause,
59. output drop
60. );
61. if\_id(
62. input clk,
63. input rst,
64. input [31:0] \_pc,
65. input [31:0] \_instr,//input instruction
66. input pause,
67. input drop,
69. output [31:0] instr\_,//output instruction
70. output [31:0] pc\_
71. );//IF.ID寄存器
72. id\_exe(
73. input clk,
74. input rst,
76. input wire[31:0] \_pc,
77. input wire[31:0] \_rd1,
78. input wire[31:0] \_rd2,
79. input wire[4:0] \_rt,
80. input wire[4:0] \_rd,
81. input wire[31:0] \_imm,
83. //input control signal
84. input wire[`ALU\_OP\_LEN-1:0] \_alu\_op,
85. input wire \_alu\_src,
86. input wire \_reg\_dst,
87. input wire \_reg\_src,
88. input wire \_pc\_src,
89. input wire \_en\_reg\_write,
90. input wire \_en\_mem\_write,
91. input wire pause,
93. //output data signal
94. output reg[31:0] pc\_,
95. output reg[31:0] rd1\_,
96. output reg[31:0] rd2\_,
97. output reg[4:0] rt\_,
98. output reg[4:0] rd\_,
99. output reg[31:0] imm\_,
100. //output control signal
101. output reg[`ALU\_OP\_LEN-1:0] alu\_op\_,
102. output reg alu\_src\_,
103. output reg reg\_dst\_,
104. output reg reg\_src\_,
105. output reg pc\_src\_,
106. output reg en\_reg\_write\_,
107. output reg en\_mem\_write\_
108. );
109. exe\_mem(
110. input clk,
111. input rst,
112. //input data signal
113. input wire \_zero,
114. input wire[31:0] \_pc,
115. input wire[31:0] \_alu\_res,
116. input wire[31:0] \_reg\_rd2,
117. input wire[4:0] \_reg\_wa,
118. //input control signal
119. input wire \_reg\_src,
120. input wire \_pc\_src,
121. input wire \_en\_reg\_write,
122. input wire \_en\_mem\_write,
123. //output data signal
124. output reg[31:0] pc\_,
125. output reg[31:0] alu\_res\_,
126. output reg[31:0] reg\_rd2\_,
127. output reg[4:0] reg\_wa\_,
128. //output control signal
129. output reg reg\_src\_,
130. output reg pc\_src\_,
131. output reg en\_reg\_write\_,
132. output reg en\_mem\_write\_
133. );
134. mem\_wb(
135. input clk,
136. input rst,
137. //data
138. input wire[31:0] \_mem\_rd,
139. input wire[4:0] \_reg\_wa,
140. input wire[31:0] \_alu\_res,
141. //control
142. input wire \_reg\_src,
143. input wire \_en\_reg\_write,
144. //data
145. output reg[31:0] mem\_rd\_,
146. output reg[31:0] alu\_res\_,
147. output reg[4:0] reg\_wa\_,
148. //control
149. output reg reg\_src\_,
150. output reg en\_reg\_write\_
151. );

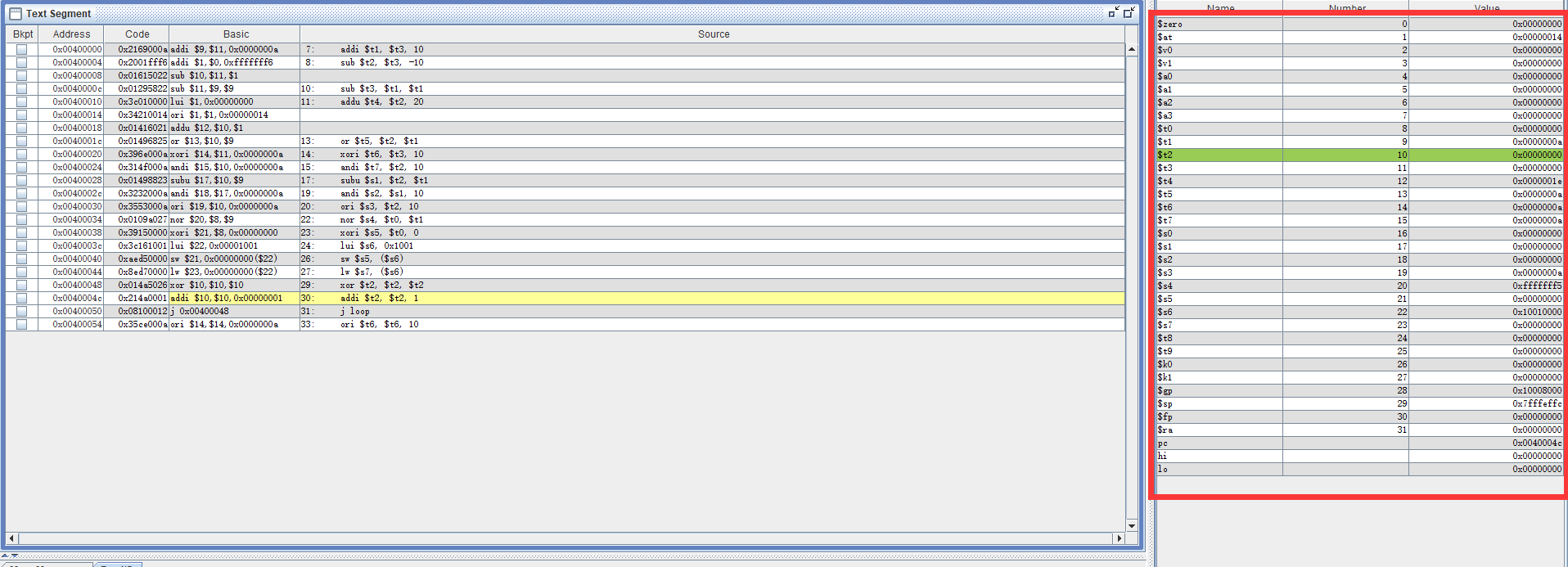
实现的逻辑的关键部分均已在前文中说明，具体实现细节可参考源码验证。

# 测试

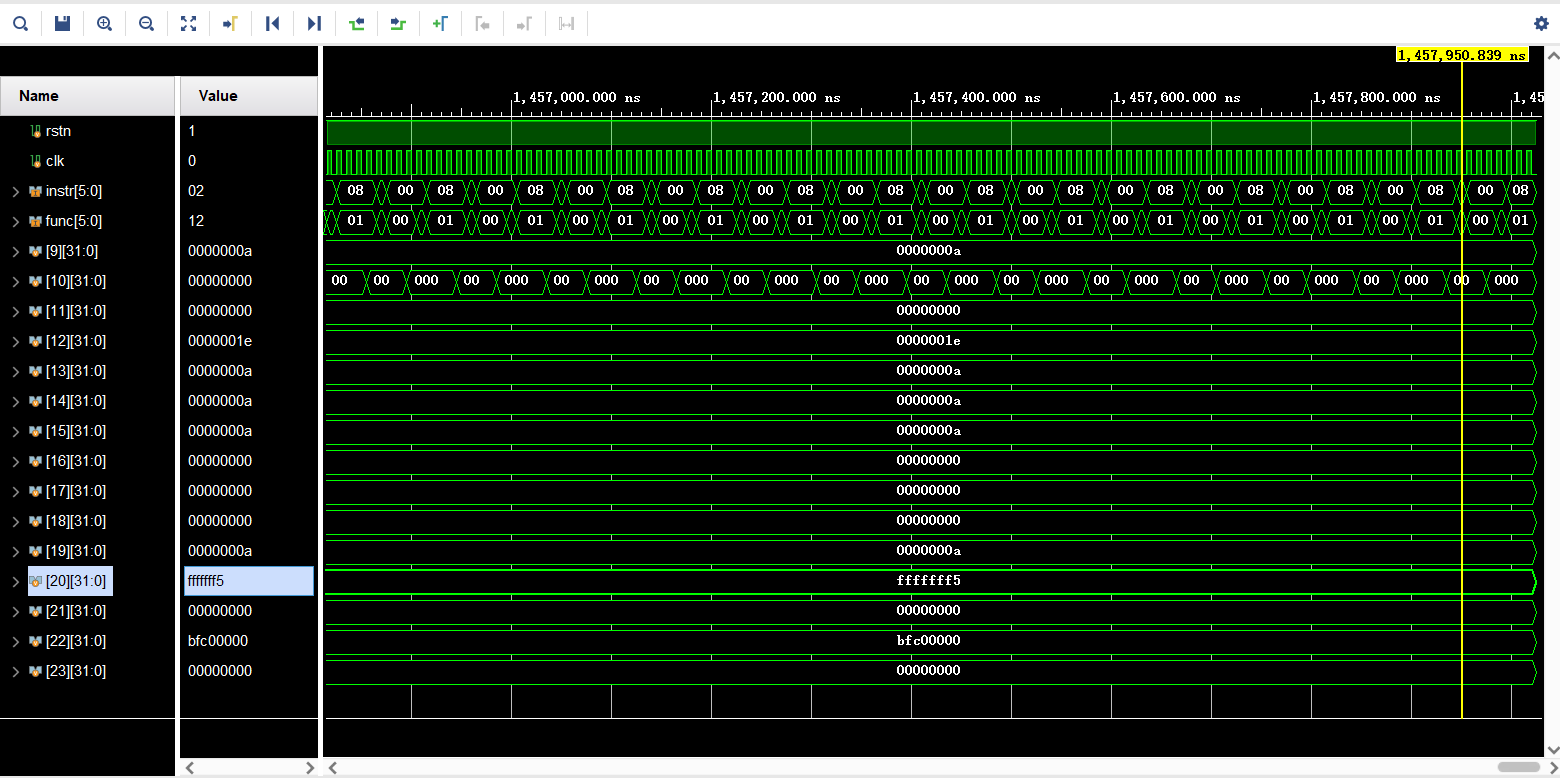
编写以下汇编程序进行仿真测试，该程序覆盖了我们设计的17条指令。

1. .org 0x0
2. .set noat
3. .set noreorder
4. .set nomacro
5. .global \_start
6. \_start:
7. addi $t1, $t3, 10
8. sub $t2, $t3, -10
9. *# 数据相关 此处插入两条空指令*
10. sub $t3, $t1, $t1
11. addu $t4, $t2, 20
13. or $t5, $t2, $t1
14. xori $t6, $t3, 10
15. andi $t7, $t2, 10
17. subu $s1, $t2, $t1
18. *# 数据相关 插入三条空指令*
19. andi $s2, $s1, 10
20. ori $s3, $t2, 10
22. nor $s4, $t0, $t1
23. xori $s5, $t0, 0
24. lui $s6, 0x1001
25. *# 数据相关 插入三条空指令*
26. sw $s5, ($s6)
27. lw $s7, ($s6)
28. loop:
29. xor $t2, $t2, $t2
30. addi $t2, $t2, 1
31. j loop
32. *# 控制相关 冲刷流水线 下一条指令不会被执行*
33. ori $t6, $t6, 10

使用Mars编译以上程序，单步执行至循环处，得到以下寄存器情况。



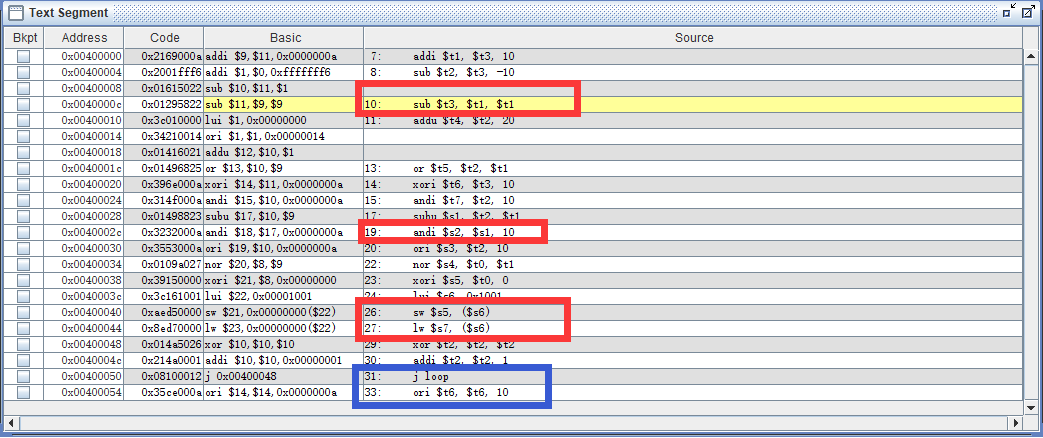
仿真得到的波形图如下所示：

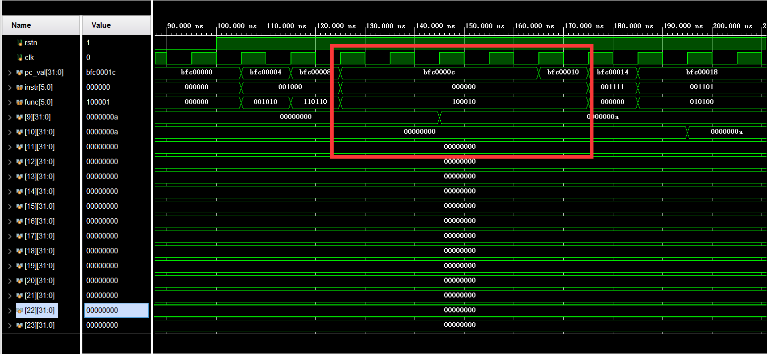


在该图中，将波形图的结果与在Mars中寄存器一致，说明CPU运行汇编测试代码的运行结果无误。

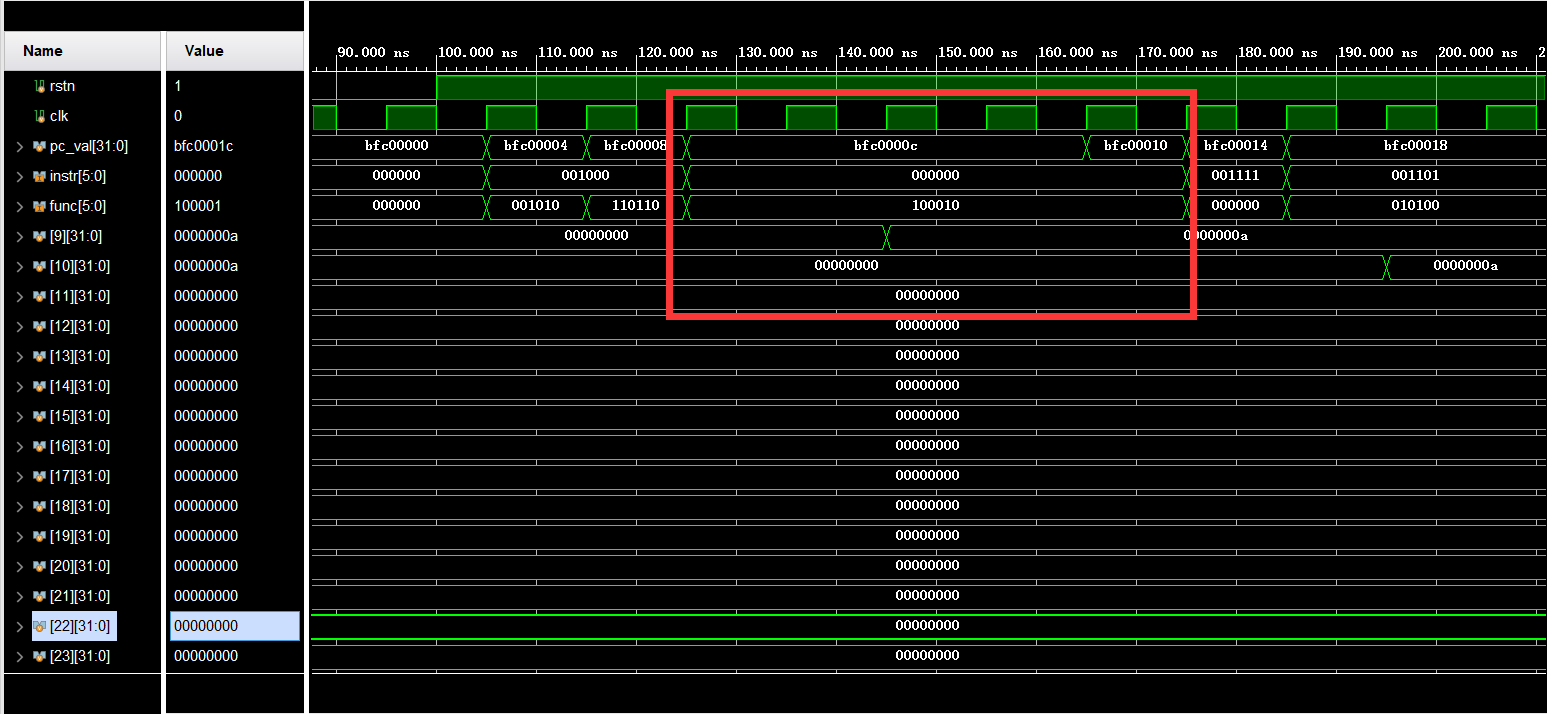
下面主要针对波形图中的数据相关与结构相关部分，对仿真结果的指令功能进行验证：

在下图中，包含三个数据相关部分（红色框）以及一个控制相关部分（蓝色框）部分

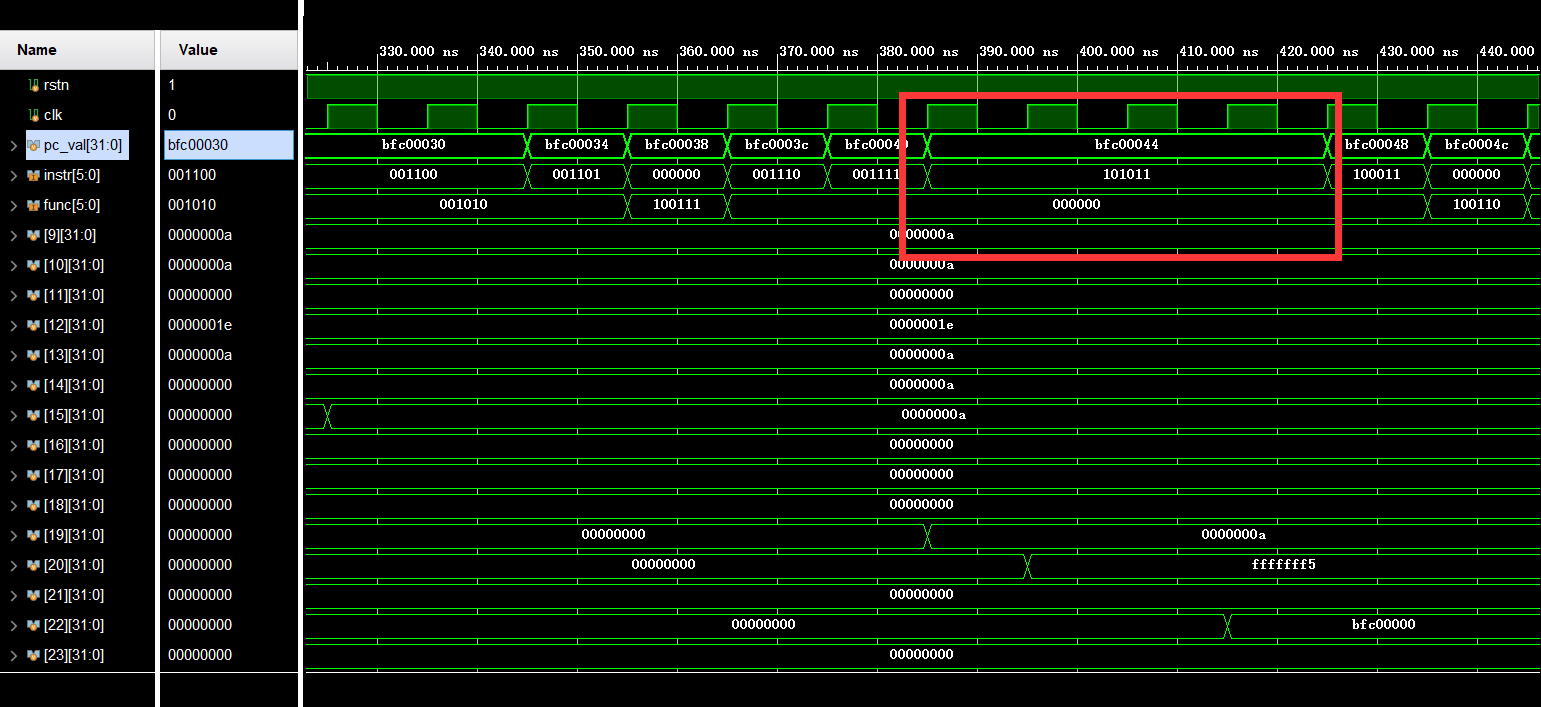




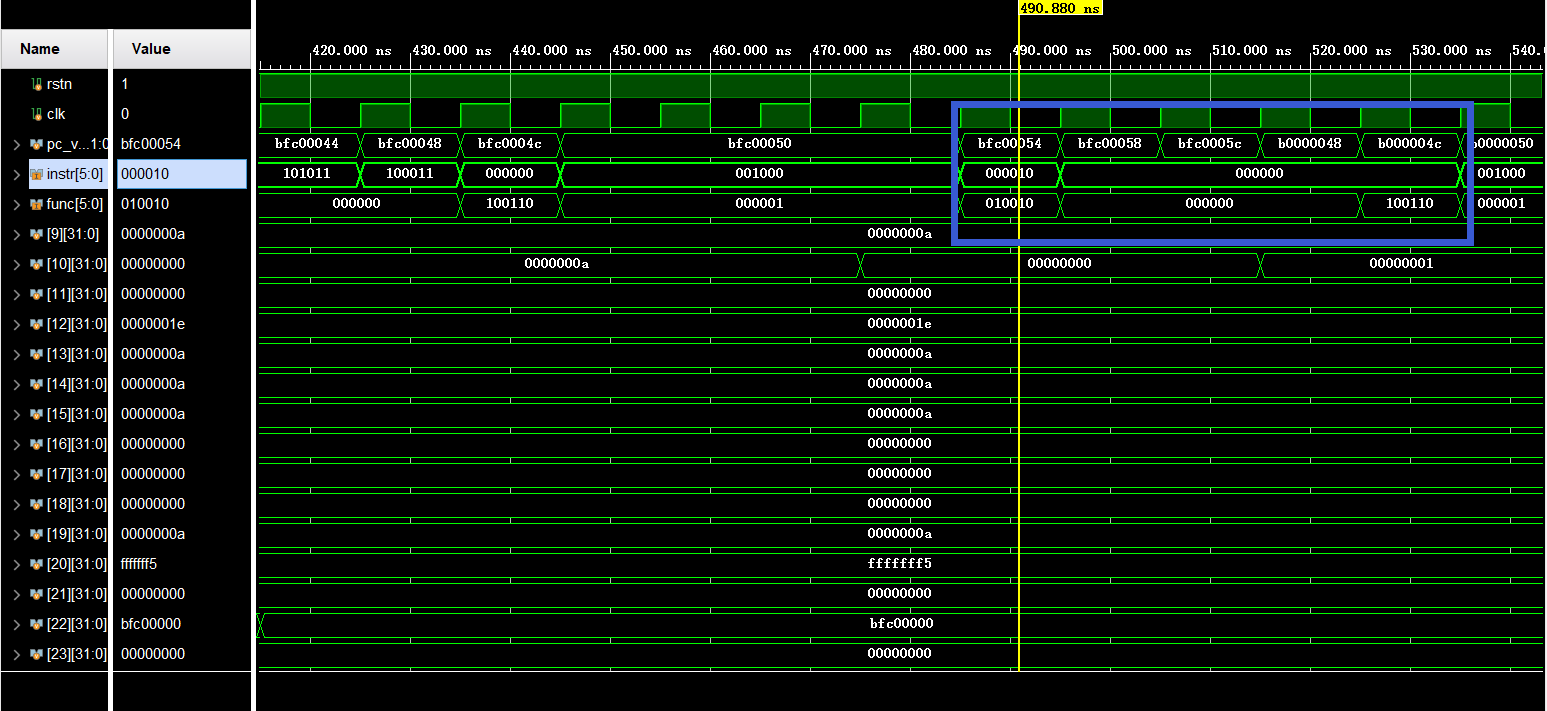
这一部分运行的是SUB指令，与之前的指令出现的数据相关，所以在执行这条SUB指令时，发现这条SUB指令持续了4个周期。这说明，在执行该指令时，出现了3个空指令。这证明了在处理第一个数据相关时，成功向其中插入了3条空指令。



这一部分运行的是ANDI指令，与之前的指令出现的数据相关，所以在执行这条ANDI指令时，发现这条ANDI指令持续了4个周期。这说明，在执行该指令时，出现了3个空指令。这证明了在处理第二个数据相关时，成功向其中插入了3条空指令。



这一部分运行的是SW指令，与之前的指令出现的数据相关，所以在执行这条SW指令时，发现这条SW指令持续了4个周期。这说明，在执行该指令时，出现了3个空指令。这证明了在处理第三个数据相关时，成功向其中插入了3条空指令。



在这一部分，展示了处理控制相关的部分。当func\_code为100110时，即执行xori指令（loop循环的起点）。在这之前，为了冲刷流水线，而可以看到中间添加了三个周期的空指令，对流水线进行冲刷，从而避免了控制相关的问题。

冒险问题测试：为了验证数据相关的操作，我们设计了三处先写后读冲突，通过波形验证，发现确实插入了三个周期的空指令，说明数据相关被成功解决；对于控制相关，我们在跳转指令之后写了一条多余指令，通过波形验证，发现流水线CPU没有执行该指令，说明其对流水线进行了冲刷，控制相关被成功解决。

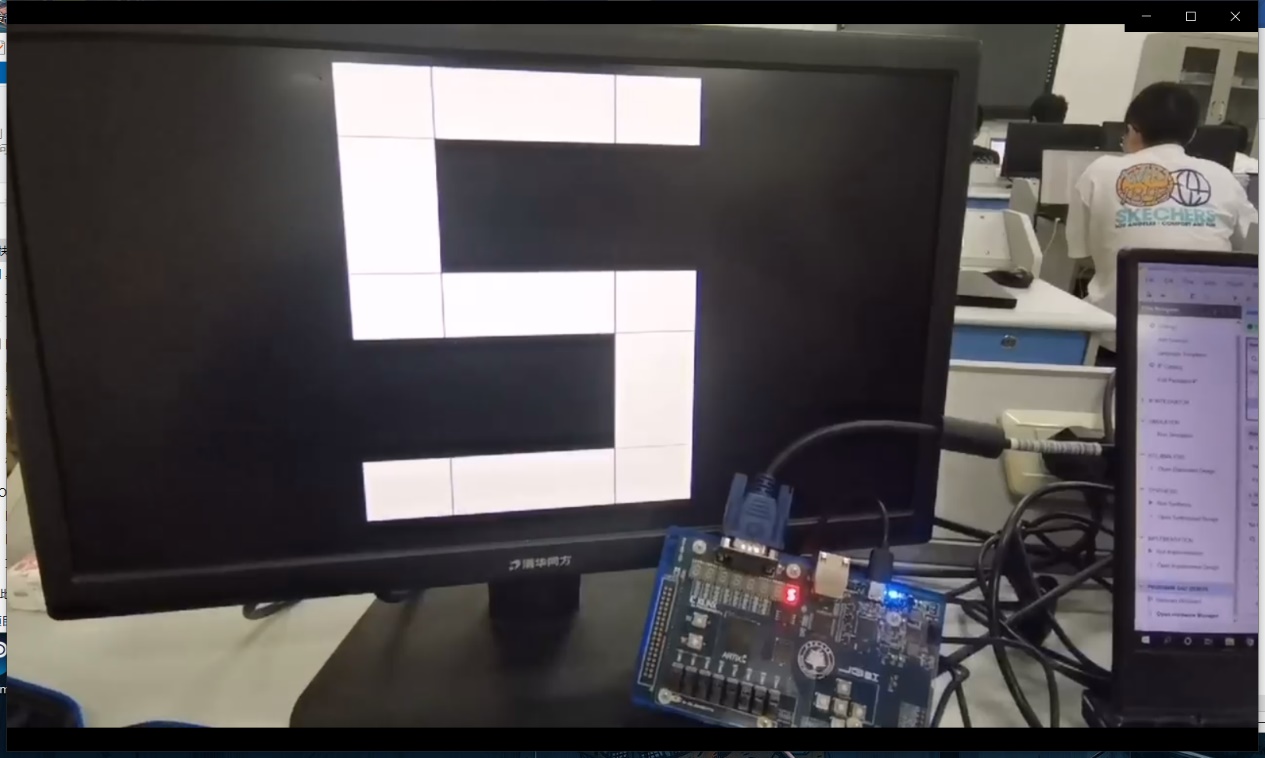
完整的仿真演示过程可以参考视频的后半部分。

对于该CPU的下板测试，采用与接口集成的方式，具体测试情况可参见另一份报告，CPU负责执行以下指令，其具体效果是循环输出一个递增的数到数据存储器的固定位置，具体效果可参考演示视频的前半部分：

1. .org 0x0
2. .set noat
3. .set noreorder
4. .set nomacro
5. .global \_start
6. \_start:
7. lui $v0,0xbfaf
8. addiu $v0,$v0,0x8000
10. sw $t0,0x0($v0)
12. j loop
13. addi $t4,$t4,0x0000
14. loop:
15. addiu $t0,$t0,0x0001
17. sw $t0,0x0($v0)
18. j loop
19. addi $t5,$t5,0x0000

该程序是一个简单的自增程序。同时在使用该程序时，连接VGA显示屏，使显示屏与板上计数器显示值一致。

下板成功图如下所示：



# 问题及解决方法

1. 问题：由于控制信号基本是没有规律的，所以直接写二进制会导致可读性很差，给设计和调试代码麻烦。

解决：采用宏的形式对各个控制信号进行编码。保证各文件的控制信号统一，并且可读性高。

1. 问题：部分指令需要进行符号扩展

解决：在CPU模块设置专用的组合电路与信号进行符号扩展的处理，用以解决特定指令问题。

1. 问题：在对CPU进行验证时，发现在数据相关是没有插入指定要求的空指令数量。

解决：通过调整指令的顺序，Mars在执行指令时，会将有些指令分成两条指令来执行，而这些被分成的指令之间可能存在数据相关，我们对这些指令进行了一定的改进。

# 心得体会及总结

团队齐心协力，共同设计实现了流水线CPU，并成功上版验证。通过本次设计流水线CPU的过程，我们对CPU的结构和流水线的工作原理有了更清楚的认知。同时，在流水线CPU中，我们处理了流水线冒险问题，并且针对数据相关和控制相关进行了仿真验证，加深了对流水线冒险问题等流水线CPU关键技术问题的理解。

# 参考文献有价值的资源推荐

无