

SHANGHAI JIAOTONG UNIVERSITY



计算机系统结构实验报告 – Lab5

姓名：刘一凡

学号：518021910609

完成时间：2020/5/1

目录

[1. 概述 3](#_Toc40977676)

[1.1 实验名称 3](#_Toc40977677)

[1.2 实验目的 3](#_Toc40977678)

[2. 实验描述 3](#_Toc40977679)

[2.1 顶层模块Top 3](#_Toc40977680)

[2.1.1 模块描述 3](#_Toc40977681)

[2.1.2 模块代码 4](#_Toc40977682)

[2.2 加法器模块Adder 8](#_Toc40977683)

[2.2.1 模块描述 8](#_Toc40977684)

[2.2.2 模块代码 8](#_Toc40977685)

[2.3 与门模块AND\_gate 9](#_Toc40977686)

[2.3.1 模块描述 9](#_Toc40977687)

[2.3.2 模块代码 9](#_Toc40977688)

[2.4 指令存取模块inst\_memory 9](#_Toc40977689)

[2.4.1 模块描述 9](#_Toc40977690)

[2.4.2 模块代码 9](#_Toc40977691)

[2.5 选择模块Mux 10](#_Toc40977692)

[2.5.1 模块描述 10](#_Toc40977693)

[2.5.2 模块代码 10](#_Toc40977694)

[2.6 指令模块PC 10](#_Toc40977695)

[2.6.1 模块描述 10](#_Toc40977696)

[2.6.2 模块代码 11](#_Toc40977697)

[2.7 左移模块shift\_left 11](#_Toc40977698)

[2.7.1 模块描述 11](#_Toc40977699)

[2.7.2 模块代码 11](#_Toc40977700)

[3. 仿真验证 12](#_Toc40977701)

[3.1 仿真代码 12](#_Toc40977702)

[3.2 指令内容 12](#_Toc40977703)

[3.3 仿真波形 13](#_Toc40977704)

[4. 实验心得 14](#_Toc40977705)

# 1. 概述

## 1.1 实验名称

类MIPS 单周期处理器的设计与实现

## 1.2 实验目的

1．Instruction memory 等模块的设计

2．单周期CPU 的实现与调试

3．功能仿真

4．上板验证

# 2. 实验描述

## 2.1 顶层模块Top

### 2.1.1 模块描述

顶层模块将类MIPS单周期处理器的各个组成部分联结起来，形成完整的处理器。本次实验中，所有的具体功能都由不同的模块实现，例如寄存器模块Registers，加法模块Adder等。Top模块的主要作用是将不同模块的输入输出连接起来，并通过连接的顺序指定整个单周期处理器的运作顺序，从而实现相关的功能。在这次实验中，除了Top模块以及前几次实现导入的模块之外，我还另外设计了加法器（Adder）模块，与门（AND\_gate）模块，指令（inst\_memory）模块，Mux模块，PC模块，移位（shift\_left）模块。设计多个模块，并且将具体的实现放入模块内部，可以使Top模块的逻辑更加清晰，可读性高，出现错误时也可以及时找出来。

### 2.1.2 模块代码

1. module Top(
2. Clock,reset
3. );
4. input Clock;
5. input reset;
7. wire Clock;
9. //control signal
10. wire REG\_WRITE;
11. wire ALU\_SRC;
12. wire MEM\_WRITE;
13. wire [1:0] ALU\_OP;
14. wire MEM\_to\_REG;
15. wire MEM\_READ;
16. wire BRANCH;
17. wire JUMP;
18. wire REG\_DST;
20. //pc
21. wire[31:0] PC\_in;
22. wire [31:0] PC\_out;
23. wire [31:0] INSTRUCTION;
25. //alu control
26. wire [3:0] ALU\_CTR;
28. //register
29. wire [31:0] WRITE\_DATA;
30. wire [4:0] WRITE\_DEST;//write to which register
31. wire [31:0] READ\_DATA1;
32. wire [31:0] READ\_DATA2;
34. //shifted and extended
35. wire [31:0] INST\_SHIFTED;
36. wire [31:0] SIGN\_EXTEND;
37. wire [31:0] EXTEND\_SHIFTED;
39. //adder result
40. wire [31:0] ADD\_RES1;//pc adder
41. wire [31:0] ADD\_RES2;//another
43. //mux result
44. wire [31:0] MUX\_RES\_REG;
45. wire [31:0] MUX\_RES\_BRANCH;
47. //ALU MODULE
48. wire ZERO;
49. wire [31:0] ALU\_RES;//alu result
51. //memory
52. wire [31:0] READ\_DATA\_MEM;//read data from the memory
54. wire  AND\_RES;//and result
55. //----------------------------------------
56. //PC part
57. PC counter(
58. //input
59. .clock(Clock),
60. .reset(reset),
61. .INPUT(PC\_in),
62. //output
63. .OUTPUT(PC\_out)
64. );
66. //get instruction for memory
67. inst\_memory get\_ins(
68. .inst\_address(PC\_out),
69. .instruction(INSTRUCTION)
70. );
72. //generate control signal
73. Ctr gen(
74. //input
75. .opcode(INSTRUCTION[31:26]),
76. //output
77. .RegDst(REG\_DST),
78. .ALUSrc(ALU\_SRC),
79. .MemToReg(MEM\_to\_REG),
80. .RegWrite(REG\_WRITE),
81. .MemRead(MEM\_READ),
82. .MemWrite(MEM\_WRITE),
83. .Branch(BRANCH),
84. .ALUOp(ALU\_OP),
85. .Jump(JUMP)
86. );
88. //pc adder
89. Adder a1(
90. .add1(32'b00000000000000000000000000000001),
91. .add2(PC\_out),
92. .result(ADD\_RES1)
93. );
95. //instruction shift 2
96. Shift\_left s1(
97. .INPUT(INSTRUCTION),
98. .OUTPUT(INST\_SHIFTED)
99. );
100. //instrcution mux
101. Mux5 m1(
102. .SEL(REG\_DST),
103. .INPUT1(INSTRUCTION[20:16]),
104. .INPUT2(INSTRUCTION[15:11]),
105. .OUTPUT(WRITE\_DEST)
106. );
108. Registers re(
109. .clock\_in(Clock),
110. .reset(reset),
111. .readReg1(INSTRUCTION[25:21]),
112. .readReg2(INSTRUCTION[20:16]),
113. .regWrite(REG\_WRITE),
114. .writeReg(WRITE\_DEST),
115. .writeData(WRITE\_DATA),
116. //output
117. .readData1(READ\_DATA1),
118. .readData2(READ\_DATA2)
119. );
121. signext se(
122. //input
123. .inst(INSTRUCTION[15:0]),
124. //OUTPUT
125. .data(SIGN\_EXTEND)
126. );
128. Mux32 m2(
129. .SEL(ALU\_SRC),
130. .INPUT1(READ\_DATA2),
131. .INPUT2(SIGN\_EXTEND),
132. .OUTPUT(MUX\_RES\_REG)
133. );
135. AluCtr actr(
136. .aluOp(ALU\_OP),
137. .funct(INSTRUCTION[5:0]),
138. //output
139. .aluCtr(ALU\_CTR)
140. );
141. //alu
142. Alu alu(
143. .input1(READ\_DATA1),
144. .input2(MUX\_RES\_REG),
145. .aluCtr(ALU\_CTR),
146. //output
147. .zero(ZERO),
148. .aluRes(ALU\_RES)
149. );
150. //signextend shift
151. Shift\_left s2(
152. .INPUT(SIGN\_EXTEND),
153. .OUTPUT(EXTEND\_SHIFTED)
154. );
155. //another add
156. Adder a2(
157. .add1(ADD\_RES1),
158. .add2(EXTEND\_SHIFTED),
159. .result(ADD\_RES2)
160. );
161. //and gate
162. AND\_gate ag(
163. .INPUT1(BRANCH),
164. .INPUT2(ZERO),
165. .OUTPUT(AND\_RES)
166. );
167. //and gate control
168. Mux32 m3(
169. .SEL(AND\_RES),
170. .INPUT1(ADD\_RES1),
171. .INPUT2(ADD\_RES2),
172. .OUTPUT(MUX\_RES\_BRANCH)
173. );
174. //update pc
175. Mux32 m4(
176. .SEL(JUMP),
177. .INPUT1(MUX\_RES\_BRANCH),
178. .INPUT2({ADD\_RES1[31:28], 2'b00, INST\_SHIFTED[25:0]}),
179. .OUTPUT(PC\_in)
180. );
182. DataMemory data\_mem(
183. .Clk(Clock),
184. .address(ALU\_RES),
185. .writeData(READ\_DATA2),
186. .memWrite(MEM\_WRITE),
187. .memRead(MEM\_READ),
188. //output
189. .readData(READ\_DATA\_MEM)
190. );
191. //memory mux
192. Mux32 m5(
193. .SEL(MEM\_to\_REG),
194. .INPUT1(ALU\_RES),
195. .INPUT2(READ\_DATA\_MEM),
196. .OUTPUT(WRITE\_DATA)
197. );
198. endmodule

## 2.2 加法器模块Adder

### 2.2.1 模块描述

加法器模块实现了简单的加法运算，主要用于指令指针PC的加法以及Branch操作的加法。

### 2.2.2 模块代码

1. module Adder(add1,add2,result);
2. input wire [31:0] add1;
3. input wire [31:0] add2;
4. output wire [31:0] result;
5. assign result=add1+add2;
6. endmodule

## 2.3 与门模块AND\_gate

### 2.3.1 模块描述

与门模块实现了按位的and操作，主要用于branch和zero控制信号的and操作。

### 2.3.2 模块代码

1. module AND\_gate(INPUT1,INPUT2,OUTPUT);
2. input INPUT1;
3. input INPUT2;
4. output OUTPUT;
5. assign OUTPUT=INPUT1 & INPUT2;
6. endmodule

## 2.4 指令存取模块inst\_memory

### 2.4.1 模块描述

指令存取模块实现了从txt文件中读取下一条指令并将其输出的功能。在此实验中我用的是绝对路径来寻找txt文件，所以在验证时需要重新指定路径。此外，需要注意的是，由于文件中的指令是一行一条，每行32位。而不是正常的四行一条，每行8位，所以需要稍微修改原来的逻辑，将PC的顺序更新从PC+4变为PC+1，不过基本的思想还是一致的。

### 2.4.2 模块代码

1. module inst\_memory(inst\_address,instruction );
2. input [31:0] inst\_address;
3. output [31:0] instruction;
4. reg [31:0] insFile[0:255];
5. initial begin
6. $readmemb("C:/Archlabs/Lab05/mem\_inst.txt", insFile);
7. end
8. assign instruction = insFile[inst\_address];
9. endmodule

## 2.5 选择模块Mux

### 2.5.1 模块描述

选择模块根据指令SEL从若干输入中选择一个作为输出，用于若干需要选择一个作为输出的环节。Mux模块有两个小模块，分别对应5位和32位这两个不同长度的输入。

### 2.5.2 模块代码

1. //2 kinds of mux operation
2. //used to define write back address
3. module Mux5(SEL,INPUT1,INPUT2,OUTPUT );
4. input SEL;
5. input [4:0] INPUT1;
6. input [4:0] INPUT2;
7. output [4:0] OUTPUT;
8. assign OUTPUT=(SEL) ? INPUT2 : INPUT1;
9. endmodule
11. //used other mux oprations
13. module Mux32(SEL,INPUT1,INPUT2,OUTPUT );
14. input SEL;
15. input [31:0] INPUT1;
16. input [31:0] INPUT2;
17. output [31:0] OUTPUT;
18. assign OUTPUT=(SEL) ? INPUT2 : INPUT1;
19. endmodule

## 2.6 指令模块PC

### 2.6.1 模块描述

指令模块将从指令读取模块输入到此模块中的指令作为输出，以生成接下来需要的控制信号和地址等。PC需要被初始化为0，表示从第一个指令开始读。

### 2.6.2 模块代码

1. //get the instruction
2. module PC(
3. clock,
4. reset,
5. INPUT,
6. OUTPUT
7. );
8. //definition
9. input clock;
10. input reset;
11. input [31:0] INPUT;
12. output reg [31:0] OUTPUT;
14. initial begin
15. OUTPUT=0;
16. end
18. always @ (posedge clock)
19. begin
20. **if** (!reset) OUTPUT=32'b00000000000000000000000000000000;
21. **else** OUTPUT=INPUT;
22. end
23. endmodule

## 2.7 左移模块shift\_left

### 2.7.1 模块描述

左移模块将输入的数据左移两位后输出，主要用于符号位扩展和PC=4的左移，目的是保证下一步处理的指令开始地址是4的倍数，从而能够正确读到下一个指令。但是由于前面指令文件一行一条的特点，PC会更新为PC+1而非PC+4，所以此模块不需要做操作，直接输出原来的数据即可。

### 2.7.2 模块代码

1. module Shift\_left(
2. INPUT,OUTPUT
3. );
4. input [31:0] INPUT;
5. output [31:0] OUTPUT;
6. assign OUTPUT=INPUT;
7. endmodule

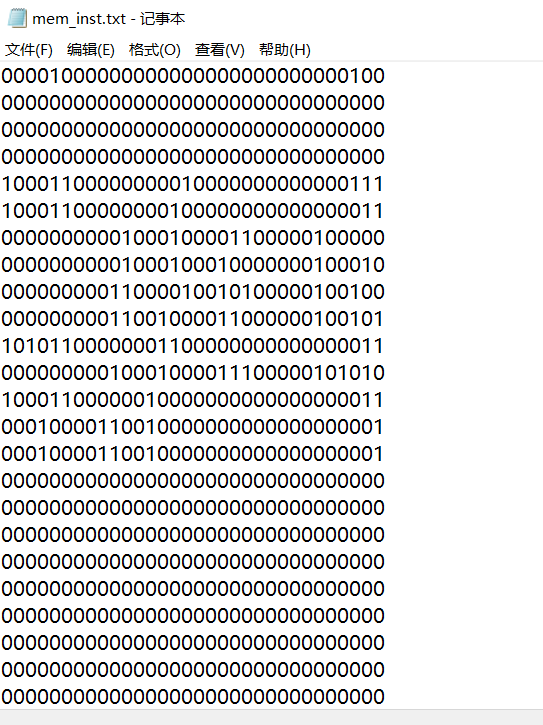
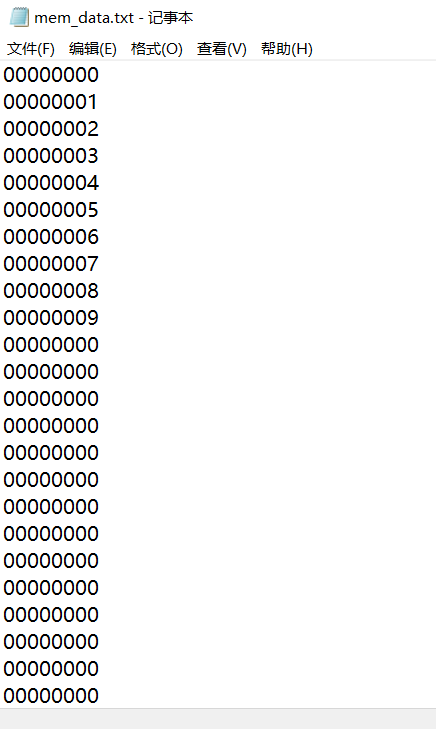
# 3. 仿真验证

## 3.1 仿真代码

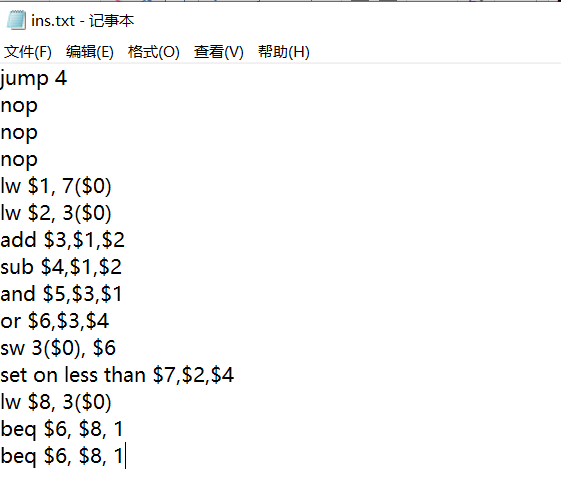
1. module top\_tb(
3. );
5. reg Clock;
6. reg reset;
7. Top uut(
8. .Clock(Clock),
9. .reset(reset)
10. );
11. always #50 Clock <= ~ Clock;
12. initial begin
13. Clock <= 0;
14. reset <= 0;
15. #1100 reset <= 1;
16. end
17. endmodule

## 3.2 指令内容

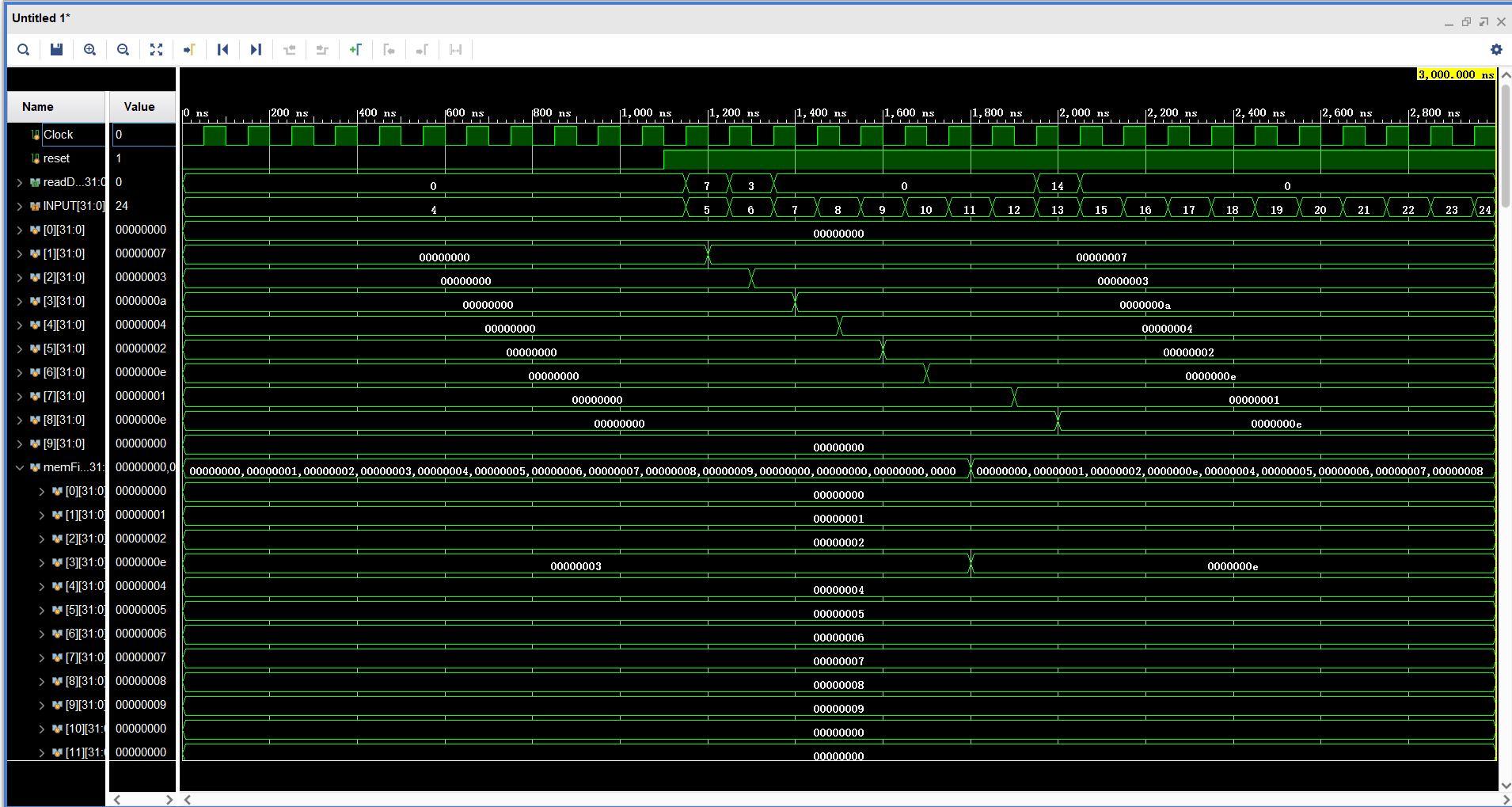
下面列出了初始状态下memory中的数据以及指令txt文件中的指令内容，具体数据在Lab5工程目录下的两个txt文件中：



下面列出了指令的具体内容：



## 3.3 仿真波形



结合指令的具体内容以及波形图中寄存器和内存数据的变化可知，模拟的类MIPS单周期处理器能从指令存储中读取指令，生成相应的控制信号并且对寄存器内存进行读取操作，并可以根据指令正确进行ALU运算，也能够正确取出下一步操作对应的指令。从测试的这些指令以及指令执行完的相应结果来看，本次实验成功。

# 4. 实验心得

本次实验是将前面几次实验做的模块进行连接，并增加一些必要的模块，设计出类似于MIPS的单周期处理器。本次实验在代码量和逻辑上比前几次实验都要难，因此我用了两次实验课的时间才将其基本完成。

本次实验的第一个难点是模块的设计，要另外设计哪些模块，这些模块的输入输出是什么样的，前面的实验大多已经设计好了接口，但是这次的实验中这些都需要根据处理器的工作原理自己想出来，因此我在设计模块的时候花了一些时间。

可能左移模块，与门模块这些模块在此次实验中似乎可以不作为模块来处理，而把它们直接写到Top模块中。但是，模块化处理有很多好处。首先可以使Top模块的整体逻辑变清晰，可读性更好；其次可以在程序出问题时方便地定位到问题处并及时修改；最后，在工程中，这些模块可以被重复使用，大大降低了工作负担。

本次实验的第二个难点是模块之间数据和信号的交互和连接，也就是各模块之间的“连线”。在进行连线操作时，需要十分清楚各个模块之间传输的数据是什么，受到什么控制信号的影响，不同的影响对应的操作有什么不同，这些都是要在连线时搞清楚的。在这一阶段，我设定了许多有意义的变量来表示不同的线，这样就可以根据这些变量名并结合原理图进行连线了，从而避免变量名太多因而搞混的情况出现。事实证明，这种有意义的变量名可读性高，也能体现出交互的原理和作用。此外，在具体实例化模块的时候，我将输入和输出用注释分隔开，这样更能体现每个模块的作用，使Top模块更容易理解。

本次实验的第三个难点是仿真时的验证，需要根据输入的指令以及输出的波形检验自己的代码逻辑是否正确。具体说来，需要先把具体的指令转换成32位的MIPS指令，再将它保存在指令文件中作为输入，最后根据波形图判断结果。在验证的过程中，我一开始发现我的代码能够正确执行指令的解码，内存的读取，指令的更新等操作，但是寄存器却无法更新。为了解决这个问题，我进行了单步调试，通过反复比对理想结果与实际结果以及控制信号，最终发现我的控制寄存器写的信号输出有误，从而导致这一控制信号永远是0，自然也就无法更新寄存器了。在改正了这一错误后，得到了理想中的仿真波形图，完成了验证。

在本次实验中，我对于类MIPS单周期处理器的整体结构和工作原理有了全面的掌握，并且能够熟练地运用模块实现功能，掌握了实例化的具体方法和流程。此外，我还掌握了从外部导入其他Verilog文件以及txt文件的方法，对工程的单步调试也有了一定的认识。感谢老师和同学对我的帮助，让我能在两次实验课上顺利完成实验。