

USTB

计算机组成原理实验指 导书

2025, Fall

作者

计算机组成原理课程组

USTB, 2025 年 9 月 1 日

目录

0 写在前面	4
0.1 关于 RTFM, STFW, RTFSC	4
RTFM	4
STFW	4
RTFSC	1
1 不考虑冲突的简单五级流水划分	2
1.1 流水级介绍	2
1.2 单周期处理器拆分	4
1.3 信号控制	12
1.4 流水线缓存	12
1.5 实验要求	13
2 指令相关和流水线冲突	14
2.1 指令相关和流水线冲突分析	14
2.2 数据相关引发的冲突	14
2.3 控制相关的冲突	15
2.4 流水线数据前递设计	15
2.4.1 路径设计	15
2.4.2 控制信号调整	15
2.4.3 访存部分设计	16
2.5 实验要求	16
3 算术逻辑转移类指令的添加	18
3.1 实验要求	19
4 访存指令的添加	20
4.1 st.b 和 st.h 指令的添加	20
4.2 ld.b、ld.h、ld.bu、ld.hu 指令的添加	20
5 乘除法指令的添加	22
5.1 运用 Xilinx IP 的方法	22
5.2 实现电路级的乘法器，除法器	24
5.3 实验要求	24
6 异常和中断支持	25
异常处理的开始阶段	25
异常结束阶段	25
6.1 控制状态寄存器	25
6.2 异常产生的条件判断	26
6.2.1 处理器核内部判定接收到中断	26
6.2.2 核内定时器中断的产生	26

6.2.3 取指地址错异常 ADEF	27
6.2.4 地址非对齐异常 ALE	27
6.2.5 指令不存在异常 INE	27
6.2.6 系统调用 SYS 和断点异常 BRK	27
6.3 精确异常的实现	27
6.4 控制状态存储器的实现	27
6.4.1 CRMD 的 PLV 域	28
6.4.2 CRMD 的 IE 域	28
6.4.3 CRMD 的 DA、PG、DATF、DATM 域	29
6.4.4 PRMD 的 PPLV、PIE 域	29
6.4.5 ECFG 的 LIE 域	29
6.4.6 ESTAT 的 IS 域	29
6.4.7 ESTAT 的 Ecode 和 EsubCode 域	30
6.4.8 ERA 的 PC 域	30
6.4.9 BADV 的 VAddr 域	30
6.4.10 EENTRY 的 VA 域	30
6.4.11 SAVE0~3	30
6.4.12 TID	30
6.4.13 TCFG 的 En、Periodic 和 InitVal 域	31
6.4.14 TVAL 的 TimeVal 域	31
6.4.15 TICLR 的 CLR 域	31
6.4.16 CSR 的读出	32
6.5 相关冲突的处理	32
6.6 实验要求	32
 7 结语	34
 A LA 组评分细则	35
A.1 任务要求	35
A.1.1 写在开头	35
A.1.2 个人任务 60 分	35
实验报告——个人部分 10 分	35
A.1.3 团队任务 40 分	35
创新基础部分	35
创新扩展部分	36
实验报告——团队部分 10 分	36
 B MIPS 组评分细则	37
B.1 任务要求	37
B.1.1 写在开头	37
B.1.2 个人任务 60 分	37
CG 测评任务 20 分	37
虚拟仿真实验平台 10 分	37
扩展指令任务 20 分	37

实验报告——个人部分 10 分	39
B.1.3 团队任务 40 分	39
创新基础部分	39
创新扩展部分	39
实验报告——团队部分 10 分	39

0 | 写在前面

第一步 (文森特·梵·高)

人们经常会忘记的是，纵使是一趟没有目的地的旅程，也还是从单纯踏出第一步开始的。



~ . ~

The machine is cold, but COAT is warm.
Computer Organization and Architecture Tutorial
Welcome to the digital world!

计算机组成原理课程，作为计算机专业的基础课，是计算机专业的使用者理解计算机构成的原理和学会用好计算机的关键。在这门课程当中，你会了解到一个基本的计算机系统是如何构成，了解到软件程序与硬件交互，更进一步地了解计算机系统背后的原理。

0.1 | 关于 RTFM, STFW, RTFSC

在学习的过程中，你会使用到我们提供的处理器开发环境，其中包括了对基本的处理器的仿真验证和综合上板的功能。在这一过程中，你可能会遇到一些错误，这些错误，有些可能是由于你对实验内容的不够了解，有些可能是因为你对实验环境提供的工具甚至是实验环境本身的不了解，这些问题不仅是在实验当中，也可能存在于大家未来的学习和工作当中，基于此，我们在这里向大家介绍计算机专业前辈们总结出对付错误的经验。

RTFM 阅读手册。在本实验中提供的手册有很多，比如实验指导书，比如说关于指令集架构的文档。如果你遇到错误，请首先先翻一翻手册，看看是否是因为手册中某个没有注意到细节而导致了错误。手册会对实验环境和实验内容进行简要的介绍，它无法覆盖实验中的方方面面，但是一定会将实验中的重点难点展现给大家，因此查询手册往往能够帮助我们快速解决问题。我们会在后面的资源使用路线中简单介绍一下本课程中可能会用到的各手册的内容。

STFW 上网搜索。手册会尽量将实验中的细节描述清楚，帮助大家完成实验。然而遗憾的是，手册没有办法覆盖到实验中的每一个细节，每一种可能的错误，（比如大魔王 Vivado 的各种报错）所以这种时候可以考虑上网搜索。从效率的角度考虑，我们希望大家尽量使用 Google/Bing 国际版进行搜索，多在国外的一些优秀论坛上寻找解决方案。

RTFSC 阅读源代码。实验环境中有大量代码，他们是整个实验环境的支撑。想要完成好实验，一定需要充分的了解实验环境，手册中已经给出了一部分关于实验环境的描述，但是并不足以支持你理解实验环境的全貌，所以适当的阅读实验环境中的源代码，诸如测试用例的源代码，仿真环境的源代码，通过阅读这些代码，相信你不仅能解决问题，也能加深自己对于整个实验环境的理解。

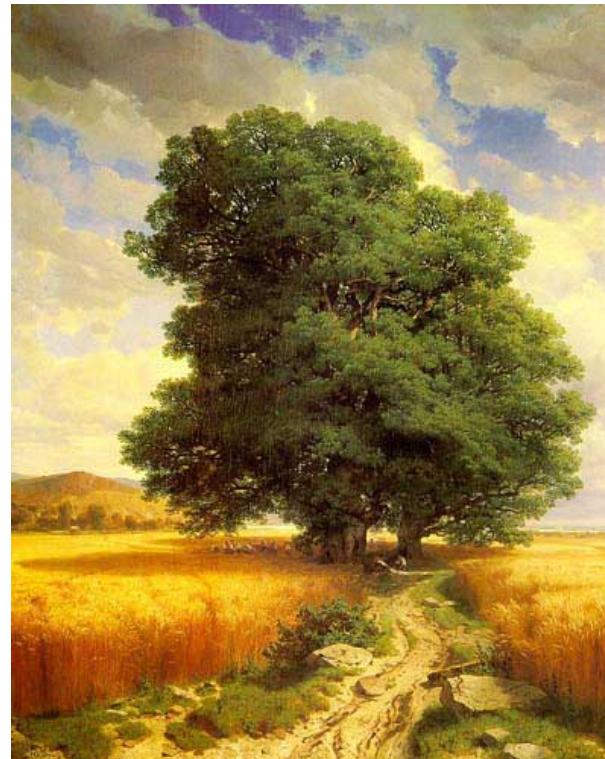
当自己已经充分尝试过以上方法时，你可以向助教，老师，或者是其他同学求助，但是希望你在提问的过程中，能够尽量清晰的描述你的问题，帮助你提问的人能够更快理解你当前所面对的情况，更好的解决问题。

关于如何提问，你可以阅读《提问的智慧》或《别像弱智一样提问》，上面的三板斧和正确提问的方法是计算机世界的答案之书，在本门课程的理论内容的学习之外，我更希望大家能够学会正确的提问，学会正确而高效的寻找答案的方法。

1 | 不考虑冲突的简单五级流水划分

有橡树的风景 (亚历山德拉·卡拉梅)

树干总是一成不变，枝叶却纷披而伸展。



~ . ~

1.1 | 流水级介绍

在计组理论课上大家已经了解过了流水级的概念，不在这里进行过多的解释。在理想情况下，无阻塞的流水级每周期处理一条指令，相比较于单周期 CPU 效率能够有显著的提升。

但是很多时候流水级会被阻塞，这就意味着如果后面的流水级被阻塞，前面的流水级也立即需要被阻塞。因为此时的后面流水级不通，系统不能接受新的数据，所有前面的流水级必须将原有的数据留在本流水级，否则会导致数据的丢失。

我们在实验中采用的策略是：

给每个流水级都安排一个“管理人员”，能与前后级的管理人员进行互相的沟通交流。他会向后一级发送“下个时刻我有东西给你”的请求，同时向前一级发送“下一个时刻我可以接受你传递来的东西”，这样子每级串联起来环环相扣。给出一个示例的设计代码：

```
1 module stallable_pipeline #(
2     parameter WIDTH = 32
3 ) (
4     input wire clk,
5     input wire resetn,
6     input wire validin,
7     input wire [WIDTH-1:0] datain,
8     input wire out_allow,
9     output wire validout,
```

```
10      output wire [WIDTH-1:0] dataout
11  );
12  reg [WIDTH-1:0] pipe1_data;
13  reg [WIDTH-1:0] pipe2_data;
14  reg [WIDTH-1:0] pipe3_data;
15  reg pipe1_valid;
16  reg pipe2_valid;
17  reg pipe3_valid;
18
19  wire pipe1_allowin;
20  wire pipe1_ready_go;
21  wire pipe1_to_pipe2_valid;
22  assign pipe1_ready_go = ....;
23  assign pipe1_allowin = ~pipe1_valid || pipe1_ready_go & pipe2_allowin;
24  assign pipe1_to_pipe2_valid = pipe1_valid & pipe1_ready_go;
25
26  always @ (posedge clk) begin
27      if (~resetn) begin // 复位使得该触发器数据无效
28          pipe1_valid <= 1'b0;
29      end else if (pipe1_allowin) begin
30          pipe1_valid <= validin;
31      end
32
33      if (pipe1_allowin & validin) begin
34          pipe1_data <= datain;
35      end
36  end
37
38  wire pipe2_allowin;
39  wire pipe2_ready_go;
40  wire pipe2_to_pipe3_valid;
41  assign pipe2_ready_go = ....;
42  assign pipe2_allowin = ~pipe2_valid || pipe2_ready_go & pipe3_allowin;
43  assign pipe2_to_pipe3_valid = pipe2_valid & pipe2_ready_go;
44
45  always @ (posedge clk) begin
46      if (~resetn) begin
47          pipe2_valid <= 1'b0;
48      end else if (pipe2_allowin) begin
49          pipe2_valid <= pipe1_to_pipe2_valid;
50      end
51      if (pipe2_allowin & pipe1_to_pipe2_valid) begin
52          pipe2_data <= pipe1_data;
53      end
54  end
55
56  wire pipe3_allowin;
57  wire pipe3_ready_go;
58  wire pipe3_to_out_valid; // 赋值给validout
59  assign pipe3_ready_go = ....;
```

```

60      assign pipe3_allowin = ~pipe3_valid || pipe3_ready_go & out_allow;
61      assign pipe3_to_out_valid = pipe3_valid & pipe3_ready_go;
62
63      always @ (posedge clk) begin
64          if (~resetn) begin
65              pipe3_valid <= 1'b0;
66          end else if (pipe3_allowin) begin
67              pipe3_valid <= pipe2_to_pipe3_valid;
68          end
69          if (pipe3_allowin & pipe2_to_pipe3_valid) begin
70              pipe3_data <= pipe2_data;
71          end
72      end
73      assign validout = pipe3_to_out_valid;
74      assign dataout = pipe3_data;
75  endmodule

```

下面对些许信号进行解释：

`pipex_valid` 表示为第 x 级的有效位，使用触发器实现，为 1 的时候表示当前时钟周期时的数据有效，好处在于，当我们需要将流水线清空的时候，不需要将所有 `data` 置为 0，只需要将 `valid` 设为 0.

`pipex_allowin` 是往前传递的信号，为 1 是表示当前允许接收前一级传入的数据。

`pipex_ready_go` 表示当前拍的状态，为 1 时表示数据在当前级的任务已经完成，可以往后一级传递数据。

`pipex_to_pipex+1_valid` 向后一级传递的信号，为 1 表示有数据需要在下个周期传入后一级。

1.2 | 单周期处理器拆分

我们设计的是经典的单发射五级流水线划分，从前往后依次是：取值（IF），译码（ID），执行（EXE），访存（MEM），写回（WB）。

取指阶段的主要功能是将指令取回；

译码阶段的主要功能是解析指令生成控制信号并读取通用寄存器堆生成源操作数；

执行阶段的主要功能是对源操作数进行算术逻辑类指令的运算或者访存指令的地址计算；

访存阶段的主要功能是取回访存的结果；

写回阶段的主要功能是将结果写入通用寄存器堆。

结合这个流水线阶段的划分方案，我们将之前设计的单发射 CPU 的数据通路拆分为五段，并在各段之间加入触发器作为流水线缓存。

这里给出取指级的详细代码：

```

1 `include "mycpu_head.v"
2
3 module IF_stage(
4     input wire clk,
5     input wire reset,
6     input wire ds_allowin,
7     input wire [`BR_BUS_WD -1 :0] br_bus,
8     output    wire           fs_to_ds_valid,

```

```
9      output wire [`FS_TO_DS_BUS_WD-1 :0] fs_to_ds_bus,
10     // inst sram interface
11     output wire inst_sram_en,
12     output wire [3:0] inst_sram_we,
13     output wire [31:0] inst_sram_addr,
14     output wire [31:0] inst_sram_wdata,
15     input  wire [31:0] inst_sram_rdata
16   );
17   //if级握手信号
18   reg          fs_valid;
19   wire         fs_ready_go;
20   wire         fs_allowin;
21   wire         to_fs_valid;
22
23   wire [31:0] seq_pc; //顺序的地址
24   wire [31:0] next_pc; //下一个执行的地址, 可能会跳转
25
26   wire br_taken; //跳转信号
27   wire [31:0] br_target;//跳转地址
28
29   assign {br_taken,br_target} = br_bus; //跳转信号传递
30
31   wire [31:0] fs_inst; //fs阶段的inst和pc, 往后传递
32   reg [31:0] fs_pc;
33
34   //if向id传递的bus
35   assign fs_to_ds_bus = {fs_inst,fs_pc};
36
37   //pre-if stage
38   assign to_fs_valid = ~reset;
39   assign seq_pc = fs_pc + 3'h4; //顺序
40   assign next_pc = (br_taken == 1 ) ? br_target :seq_pc;
41
42   //IF stage
43   assign fs_ready_go = 1'b1; //可以传输
44   assign fs_allowin = !fs_valid || fs_ready_go && ds_allowin; //允许进入,
45   assign fs_to_ds_valid = fs_valid && fs_ready_go;
46
47   always@(posedge clk)begin
48     if(reset)begin
49       fs_valid <= 1'b0;
50     end else if(fs_allowin)begin
51       fs_valid <= to_fs_valid;
52     end
53   end
54
55   always@(posedge clk)begin
56     if(reset)begin
57       fs_pc <= 32'h1bfffffc;
58     end else if(to_fs_valid && fs_allowin)begin
```

```

59      fs_pc <= next_pc;
60  end
61 end
62
63 assign inst_sram_en = to_fs_valid && fs_allowin; // 
64 assign inst_sram_we = 4'h0; //写
65 assign inst_sram_addr =next_pc;
66 assign inst_sram_wdata = 32'b0;
67
68 assign fs_inst = inst_sram_rdata;
69
70 endmodule

```

关于转移指令更新 PC，由于转移指令需要在译码级才能知道正确的跳转方向和目标，所以我们将转移指令的 pc 修改时间放在译码级，此时我们应该注意，pre-IF 阶段的 `nextpc` 生成逻辑中，来自 PC 相对转移指令跳转目标的那一支，其跳转目标计算所用的 PC 是处在译码阶段的转移指令的 PC，不是此时取指阶段的 PC。

`fs_to_ds_bus` 存放着每一级的缓存，需要往后传递，它的位宽取决于数据内容的大小，这里通过宏定义 (`mycpu_head.v`) 进行统一管理，具体如下，`xx` 为自身传递的位宽大小，根据自身设计而定，没有强制要求：

```

1 `ifndef MYCPU_H
2   `define MYCPU_H
3
4   `define BR_BUS_WD      xx
5   `define FS_TO_DS_BUS_WD  xx
6   `define DS_TO_ES_BUS_WD  xx
7   `define ES_TO_MS_BUS_WD  xx
8   `define MS_TO_WS_BUS_WD  xx
9   `define WS_TO_RF_BUS_WD  xx
10  `endif

```

给出译码级的详细代码：

```

1 `include "mycpu_head.v"
2
3 module ID_stage(
4   input wire    clk,
5   input wire    reset,
6   //allowin
7   input wire    es_allowin,
8   output wire   ds_allowin,
9   //from if stage
10  input wire      fs_to_ds_valid,
11  input wire [`FS_TO_DS_BUS_WD-1 : 0] fs_to_ds_bus,
12  //to ex stage
13  output wire     ds_to_es_valid,
14  output wire [`DS_TO_ES_BUS_WD -1 : 0 ] ds_to_es_bus,
15  //to if stage
16  output wire   [`BR_BUS_WD -1 : 0 ] br_bus ,

```

```
17     //to rf :write back
18     input wire [`WS_TO_RF_BUS_WD -1 :0] ws_to_rf_bus
19 );
20
21     wire br_taken;
22     wire [31:0] br_target;
23
24     wire [31:0] ds_pc;
25     wire [31:0] ds_inst;
26
27     reg      ds_valid;
28     wire      ds_ready_go;
29
30     wire [11:0] alu_op;
31
32     wire load_op; //加载信号
33     wire src1_is_pc; //来自src为pc地址
34     wire src2_is_imm; //src2为imm
35     wire res_from_mem; //result来自mem
36     wire dst_is_r1;
37     wire gr_we;
38     wire mem_we;
39     wire src_reg_is_rd;
40     wire [4:0]dest;
41     wire rj_eq_rd;
42     wire [31:0]rj_value;
43     wire [31:0]rkd_value;
44     wire [31:0]imm;
45     wire [31:0]br_offs;
46     wire [31:0]jirl_offs;
47
48 //分段
49     wire [5:0] op_31_26;
50     wire [3:0] op_25_22;
51     wire [ 1:0] op_21_20;
52     wire [ 4:0] op_19_15;
53
54     wire [ 4:0] rd;
55     wire [ 4:0] rj;
56     wire [ 4:0] rk;
57
58     wire [11:0] i12;
59     wire [19:0] i20;
60     wire [15:0] i16;
61     wire [25:0] i26;
62
63     wire [63:0] op_31_26_d;
64     wire [15:0] op_25_22_d;
65     wire [ 3:0] op_21_20_d;
66     wire [31:0] op_19_15_d;
```

```
67      wire      inst_add_w;
68      wire      inst_sub_w;
69      wire      inst_slt;
70      wire      inst_sltu;
71      wire      inst_nor;
72      wire      inst_and;
73      wire      inst_or;
74      wire      inst_xor;
75      wire      inst_slli_w;
76      wire      inst_srli_w;
77      wire      inst_srai_w;
78      wire      inst_addi_w;
79      wire      inst_ld_w;
80      wire      inst_st_w;
81      wire      inst_jirl;
82      wire      inst_b;
83      wire      inst_bl;
84      wire      inst_beq;
85      wire      inst_bne;
86      wire      inst_lu12i_w;
87
88
89      wire      need_ui5;
90      wire      need_si12;
91      wire      need_si16;
92      wire      need_si20;
93      wire      need_si26;
94      wire      src2_is_4;
95
96      wire [ 4:0] rf_raddr1;
97      wire [31:0] rf_rdata1;
98      wire [ 4:0] rf_raddr2;
99      wire [31:0] rf_rdata2;
100
101     wire rf_we ; //写使能
102     wire [4:0] rf_waddr;
103     wire [31:0] rf_wdata;
104
105     wire [31:0] alu_src1 ;
106     wire [31:0] alu_src2;
107     wire [31:0] alu_result;
108
109     wire [31:0] mem_result;
110     wire [31:0] final_result;
111
112     assign op_31_26  = ds_inst[31:26];
113     assign op_25_22  = ds_inst[25:22];
114     assign op_21_20  = ds_inst[21:20];
115     assign op_19_15  = ds_inst[19:15];
116
```

```
117 assign rd = ds_inst[ 4: 0];
118 assign rj = ds_inst[ 9: 5];
119 assign rk = ds_inst[14:10];
120
121 assign i12 = ds_inst[21:10];
122 assign i20 = ds_inst[24: 5];
123 assign i16 = ds_inst[25:10];
124 assign i26 = {ds_inst[ 9: 0], ds_inst[25:10]};
125
126 decoder_6_64 u_dec0(.in(op_31_26 ), .out(op_31_26_d ));
127 decoder_4_16 u_dec1(.in(op_25_22 ), .out(op_25_22_d ));
128 decoder_2_4 u_dec2(.in(op_21_20 ), .out(op_21_20_d ));
129 decoder_5_32 u_dec3(.in(op_19_15 ), .out(op_19_15_d ));
130
131 /////////////////////////////////
132 assign inst_add_w = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] &
133     op_19_15_d[5'h00];
134 assign inst_sub_w = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] &
135     op_19_15_d[5'h02];
136 assign inst_slt = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] &
137     op_19_15_d[5'h04];
138 assign inst_sltu = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] &
139     op_19_15_d[5'h05];
140 assign inst_nor = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] &
141     op_19_15_d[5'h08];
142 assign inst_and = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] &
143     op_19_15_d[5'h09];
144 assign inst_or = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] &
145     op_19_15_d[5'h0a];
146 assign inst_xor = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] &
147     op_19_15_d[5'h0b];
148 assign inst_slli_w = op_31_26_d[6'h00] & op_25_22_d[4'h1] & op_21_20_d[2'h0] &
149     op_19_15_d[5'h01];
150 assign inst_srli_w = op_31_26_d[6'h00] & op_25_22_d[4'h1] & op_21_20_d[2'h0] &
151     op_19_15_d[5'h09];
152 assign inst_srai_w = op_31_26_d[6'h00] & op_25_22_d[4'h1] & op_21_20_d[2'h0] &
153     op_19_15_d[5'h11];
154 assign inst_addi_w = op_31_26_d[6'h00] & op_25_22_d[4'ha];
155 assign inst_ld_w = op_31_26_d[6'h0a] & op_25_22_d[4'h2];
156 assign inst_st_w = op_31_26_d[6'h0a] & op_25_22_d[4'h6];
157 assign inst_jirl = op_31_26_d[6'h13];
158 assign inst_b = op_31_26_d[6'h14];
159 assign inst_bl = op_31_26_d[6'h15];
160 assign inst_beq = op_31_26_d[6'h16];
161 assign inst_bne = op_31_26_d[6'h17];
162 assign inst_lu12i_w= op_31_26_d[6'h05] & ~ds_inst[25];
163
164 /////////////////////////////////
165 assign alu_op[ 0] = inst_add_w | inst_addi_w | inst_ld_w | inst_st_w
166                 | inst_jirl | inst_bl;
```

```
156 assign alu_op[ 1] = inst_sub_w;
157 assign alu_op[ 2] = inst_slt;
158 assign alu_op[ 3] = inst_sltu;
159 assign alu_op[ 4] = inst_and;
160 assign alu_op[ 5] = inst_nor;
161 assign alu_op[ 6] = inst_or;
162 assign alu_op[ 7] = inst_xor;
163 assign alu_op[ 8] = inst_slli_w;
164 assign alu_op[ 9] = inst_srli_w;
165 assign alu_op[10] = inst_srai_w;
166 assign alu_op[11] = inst_lu12i_w;
167 //信号
168 assign need_ui5   = inst_slli_w | inst_srli_w | inst_srai_w;
169 assign need_si12  = inst_addi_w | inst_ld_w | inst_st_w;
170 assign need_si16  = inst_jirl | inst_beq | inst_bne;
171 assign need_si20  = inst_lu12i_w;
172 assign need_si26  = inst_b | inst_bl;
173 assign src2_is_4  = inst_jirl | inst_bl;
174
175 assign imm = src2_is_4 ? 32'h4 : :
176           need_si20 ? {i20[19:0], 12'b0} : :
177           need_ui5 ? rk : :
178           {{20{i12[11]}}, i12[11:0]} ;
179
180 assign br_offs = need_si26 ? {{ 4{i26[25]}}, i26[25:0], 2'b0} :
181           {{14{i16[15]}}, i16[15:0], 2'b0} ;
182
183 assign jirl_offs = {{14{i16[15]}}, i16[15:0], 2'b0};
184
185 assign src_reg_is_rd = inst_beq | inst_bne | inst_st_w;
186
187 assign src1_is_pc   = inst_jirl | inst_bl;
188
189 assign src2_is_imm  = inst_slli_w |
190           inst_srli_w |
191           inst_srai_w |
192           inst_addi_w |
193           inst_ld_w  |
194           inst_st_w  |
195           inst_lu12i_w|
196           inst_jirl  |
197           inst_bl    ;
198
199 assign res_from_mem = inst_ld_w;
200 assign dst_is_r1    = inst_bl;
201 assign gr_we        = ~inst_st_w & ~inst_beq & ~inst_bne & ~inst_b ;
202 assign mem_we       = inst_st_w;
203 assign dest         = dst_is_r1 ? 5'd1 : rd;
204
205 assign rf_raddr1 = rj;
```

```
206     assign rf_raddr2 = src_reg_is_rd ? rd : rk;
207
208     regfile u_Regfile(
209         .clk      (clk      ),
210         .raddr1 (rf_raddr1),
211         .rdata1 (rf_rdata1),
212         .raddr2 (rf_raddr2),
213         .rdata2 (rf_rdata2),
214         .we      (rf_we    ),
215         .waddr   (rf_waddr ),
216         .wdata   (rf_wdata )
217     );
218
219     assign rj_value  = rf_rdata1;
220     assign rkd_value = rf_rdata2;
221
222     assign rj_eq_rd = (rj_value == rkd_value);
223     assign br_taken = ( inst_beq && rj_eq_rd
224                         || inst_bne && !rj_eq_rd
225                         || inst_jirl
226                         || inst_bl
227                         || inst_b
228                         ) && ds_valid;
229     assign br_target = (inst_beq || inst_bne || inst_bl || inst_b) ? (ds_pc +
230                           br_offs) :
231                                     /*inst_jirl*/ (rj_value +
232                                     jirl_offs);
233
234     assign br_bus = {br_taken , br_target};
235
236     reg [`FS_TO_DS_BUS_WD-1 : 0] fs_to_ds_bus_r;
237
238     assign {ds_inst,
239             ds_pc} = fs_to_ds_bus_r;
240
241     assign {rf_we, //37:37
242             rf_waddr, //36:32
243             rf_wdata //31:0
244             } = ws_to_rf_bus;
245
246     assign ds_to_es_bus = {alu_op          ,
247                           load_op        ,
248                           src1_is_pc    ,
249                           src2_is_imm   ,
250                           src2_is_4     ,
251                           gr_we         ,
252                           mem_we        ,
253                           dest          ,
254                           imm           ,
255                           rj_value      ,
```

```

254           rkd_value      ,
255           ds_pc          ,
256           res_from_mem
257       };
258
259
260 ///////////////////////////////////////////////////////////////////
261 assign ds_ready_go    = 1'b1;
262 assign ds_allowin     = !ds_valid || ds_ready_go && es_allowin;
263 assign ds_to_es_valid = ds_valid && ds_ready_go;
264 always @ (posedge clk) begin
265   if (reset) begin
266     ds_valid <= 1'b0;
267   end
268   else if (ds_allowin) begin
269     ds_valid <= fs_to_ds_valid;
270   end
271   if (fs_to_ds_valid && ds_allowin) begin
272     fs_to_ds_bus_r <= fs_to_ds_bus;
273   end
274 end
275
276 endmodule

```

通过分析给出的两级的详细代码，以及此前在上学期我们实现的单周期 CPU 不难发现，代码的主要逻辑部分与单周期 CPU 基本没有差异。现在实现的不考虑冲突的简单五级流水划分的设计就是将单周期 CPU 划分为五级，然后在每级中加入数据缓存，同时使用握手信号对流水级进行控制即可。

1.3 | 信号控制

对于 IF 流水阶段来说，由于目前只从指令 RAM 中取回指令，因此当指令位于取指阶段的时候，指令 RAM 一定可以返回指令码，于是取指阶段的 `ready_go` 信号恒为 1。

对于 ID 流水阶段来说，如果我们暂时不考虑上一节所说的转移指令在译码阶段等待延迟槽指令取回的话，那么由于译码、读寄存器堆都是一拍之内一定可以完成的，所以译码阶段的 `ready_go` 信号恒为 1。

对于 EXE 流水阶段来说，由于目前处理的所有指令在这一阶段均只需要一拍就可以完成，所以 EXE 阶段的 `ready_go` 信号恒为 1。

对于 MEM 流水阶段来说，由于目前只从数据 RAM 中取回数据，因此当 load 类指令位于 MEM 阶段的时候，数据 RAM 一定可以返回数据，于是 MEM 阶段的 `ready_go` 信号恒为 1。

对于 WB 流水阶段来说，由于写回寄存器堆在一拍之内一定可以完成，因此 WB 阶段的 `ready_go` 信号恒为 1。

1.4 | 流水线缓存

根据译码级的例子：

```
1 assign ds_to_es_bus = {alu_op ,
```

```

2      load_op      ,
3      src1_is_pc   ,
4      src2_is_imm  ,
5      src2_is_4    ,
6      gr_we       ,
7      mem_we      ,
8      dest        ,
9      imm         ,
10     rj_value    ,
11     rkd_value   ,
12     ds_pc       ,
13     res_from_mem
14   };

```

这是译码阶段向执行阶段发送的流水级缓存，不难看出这里面包括了数据信号以及控制信号，换而言之，包括了往后流水级需要使用的所有信号，举例说明：

- 上面的写使能信号（gr_we）在译码级已经生成，但在写回级才会被使用，这就意味着它被三级流水线缓存隔断，需要一级一级的往后传递，直至写回级被使用。
- 上面的 alu_op 信号在下个阶段（EXE）就被使用了，那么当执行级向缓存级发送缓存时，便不需要将此信号继续传递下去。

根据给出的两个流水级示例，拆分你的单周期 CPU，实现一个不考虑冲突的五级流水 CPU。

1.5 | 实验要求

将上学期的单周期处理器代码转移到给出的 myCPU 文件夹中，直接覆盖代码即可，使用 `soc_bram/` 子目录使用 tcl 命令创建工程，在此基础上开始进行第一个实验的实现：

- 调整 CPU 顶层接口，增加指令 RAM 的片选信号 `inst_sram_en` 和数据 RAM 的片选信号 `data_sram_en`。
- 调整 CPU 顶层接口，将 `inst_sram_we` 和 `data_sram_we` 都从 1 比特的写使能调整为 4 比特的字节写使能。
- 设计一个不考虑相关引发的冲突的单发射五级流水 CPU。
- 运行对应的 func，要求成功通过仿真和上板验证。

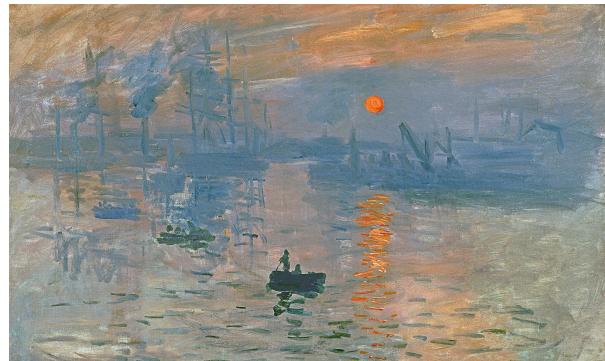
从现在开始的指令 RAM 和数据 RAM 均采用 block RAM 实现，其访问时需要给出片选信号。为此 myCPU 顶层接口中增加指令 RAM 的片选信号 `inst_sram_en` 和数据 RAM 的片选信号 `data_sram_en`。两个信号均为 1 比特，均为高电平有效。

尽管目前存数指令仅实现了 st.w，但考虑到后续实践任务的需求，myCPU 顶层接口中的 `inst_sram_we` 和 `data_sram_we` 都从 1 比特改为 4 比特，其含义也从 RAM 的写使能调整为 RAM 的字节写使能。

2 | 指令相关和流水线冲突

日出·印象（克劳德·莫奈）

逆境是人类获得知识的最高学府，难题是人们取得智慧之门。



~ · ~

2.1 | 指令相关和流水线冲突分析

前面的设计不考虑流水线冲突，在执行简单指令序列的时候很理想通畅，可以在每个周期都完成一条指令，但是在现实中并不是每时每刻都如此的称心如意的，通常都会存在着指令间的相关，对导致指令的实现出错，在下面简单进行举例：

```
add.w $r2, $r1, $r1  
add.w $r3, $r2, $r2
```

在这个指令序列中，第一条的结果会写入到 r_2 中进行，第二条指令需要使用 r_2 寄存器的值进行运算。

根据我们设计的五级流水结构来看，数据只有在最后的写回阶段才会把结果返回到寄存器堆中，但是第一条指令在执行阶段的时候，第二条指令（此时处于译码阶段）就需要读取寄存器的堆的值，但是数据并还没写回到寄存器堆中，就会出现写后读的错误——读取到的是 r_2 寄存器的旧值，进而造成输出的错误。

指令相关分为三类：**数据相关，控制相关，结构相关**。

如果两条指令访问同一个寄存器或者内存，且至少有一条是写寄存器或者内存的指令，那么便是存在数据相关。（上面的例子便是一种）

如果两个指令一条是转移指令且另外一个是否执行取决于转移指令的执行结果，这便是控制相关。

如果两个指令都使用同一个硬件资源，便是存在结构相关。

2.2 | 数据相关引发的冲突

第一种是写后读（RAW），后面的指令要用到前面指令所写的数据，这便是真相关。

第二种是写后写（WAW），两个指令写同一个单元。

第三种是读后写（WAR），后面指令写入前面所读的单元，即反相关。

在我们设计的五级流水结构中，只有真相关会导致流水线冲突，并不需考虑后两种。（如果有同学想要实现乱序执行流水的话则需要考虑）

那么如何解决此冲突呢？这里给出的最直观简单的解决方法——**阻塞**，让需要结果的指令一直阻塞在译码级，等到上一条将结果写回到寄存器堆种，才让其进入执行阶段。

关键在于判断处于译码级的指令是需要等待还是前进的条件如何生成。

我们知道，RAW 的出现是因为译码级的指令与后面流水级指令访问同一个寄存器导致的，那么我们只需要判断：**此时译码级指令中具有来自非 0 号寄存器的源操作数，如果这些操作数中任何一个的寄存器号与当前这个时刻的执行，访存和写回级的目的操作数的非 0 号寄存器号是否相同**。

那么如何阻塞呢，使用 `ready_go` 信号。只需调整译码级的信号即可，他此时不再恒为 1，当存在写后读相关的时候，需要把此信号变为 0.

2.3 | 控制相关的冲突

我们带入一个情景帮大家去更好的理解：

假设有一个转移指令，PC 为 0×1000 ，在第一个时钟周期，它位于取指阶段，此时的预取指阶段同时已经在计算 `nexpc` 了，但是此时是不可能根据转移指令的情况来判断，因为此时 CPU 仍不知道此为何指令，更不清楚这是否跳转。`nexpc` 只能顺序设置为 0×1004 。

在第二个周期， 0×1000 的指令来到了译码阶段， 0×1004 来到了取值阶段，此时如果不跳转，那么预取指仍是顺序取 0×1008 ， 0×1004 也是需要仍然留在执行路径上，继续顺序执行即可。

如果发生跳转(假设跳转到 0×3000)，那么需要直接调整预取指阶段的为跳转地址为 0×3000 ，但是无法改变存在于 0×1004 的指令存在流水线的事实，他是不可能被执行的。那么如何对待这条指令，下面给出一种方法：

回忆流水线的设计参考，我们如何表示每个流水级上有没有指令——用 `valid` 信号来实现的，所以其实非常的简单，只需要用 `valid` 信号判断即可。

取消一条指令，就是把伴随这个指令的 `valid` 变为 0

2.4 | 流水线数据前递设计

我们在上面实现的是用阻塞的方法来解决真相关的问题，其实我们不难发现，其实数据的结果在译码之后的阶段已经算出，只是还没到写回级进行写回，那我们是不是可以进行一个判断，然后将数据直接送到译码级，这样不就不需要等待了？

这种方法便是前递 (forward)，也称为旁路 (bypass)

2.4.1 | 路径设计

需要将执行，访存，写回三级的数据前递到译码级，主要优先级：EXE>MEM>WB

2.4.2 | 控制信号调整

并不是使用前递就能保证不需要任何阻塞，这个时候就可以把 `ready_go` 信号变为恒为 1 了吗，显然不行，现在只需要考虑一种问题——`load_delay` 问题：

```
ld.w $r2, $r1,0x0;
add.w $r4, $r3, $r2;
```

假设此时 `ld` 指令位于执行级，`add.w` 便位于译码级，请问下一拍 `add.w` 能进入下一级吗？

显然不能，因为此时 `ld` 指令还没正确的生成最后结果，仍还是需要将其阻塞一个周期，此时 `ready_go` 信号还是得置为 0。

2.4.3 | 访存部分设计

在前面的设计中，我们默认 RAM 的读写使能、片选使能和字节使能为高电平有效，但实际访问 BaseRAM 的时候为低电平有效。另外 RAM 地址为 20 位，所以应当选取原地址中 [21:2] 的部分作为实际访存地址。下面给出简单的访存部分代码：

```

1 module BaseRAM_control(
2     input wire          clk,
3     input wire          reset,
4     // from cpu
5     input wire          inst_sram_en,
6     input wire [31: 0]   inst_sram_addr,
7     input wire [31: 0]   inst_sram_wdata,
8     output reg [31: 0]  inst_sram_rdata,
9     input wire [ 3: 0]  inst_sram_we,
10
11    // to base ram
12    inout wire [31: 0]  base_ram_data,
13    output wire [19: 0]  base_ram_addr,
14    output wire [ 3: 0]  base_ram_be_n,
15    output wire          base_ram_ce_n,
16    output wire          base_ram_oe_n,
17    output wire          base_ram_we_n,
18 );
19    assign base_ram_be_n = ( | inst_sram_we && inst_sram_en) ? ~inst_sram_we : 4'h0
20        ;
21    assign base_ram_ce_n = ~inst_sram_en;
22    assign base_ram_we_n = ~( | inst_sram_we && inst_sram_en);
23    assign base_ram_oe_n = ( | inst_sram_we && inst_sram_en);
24
25    assign base_ram_addr = inst_sram_addr[21:2];
26
27    assign base_ram_data = ~base_ram_we_n ? inst_sram_wdata : 32'hz;
28
29    always @ (posedge clk) begin
30        if (reset) begin
31            inst_sram_rdata <= 32'hz;
32        end else if (~base_ram_oe_n) begin
33            inst_sram_rdata <= base_ram_data;
34        end
35    end
endmodule

```

2.5 | 实验要求

在上一章的基础上实现流水线冲突的处理：

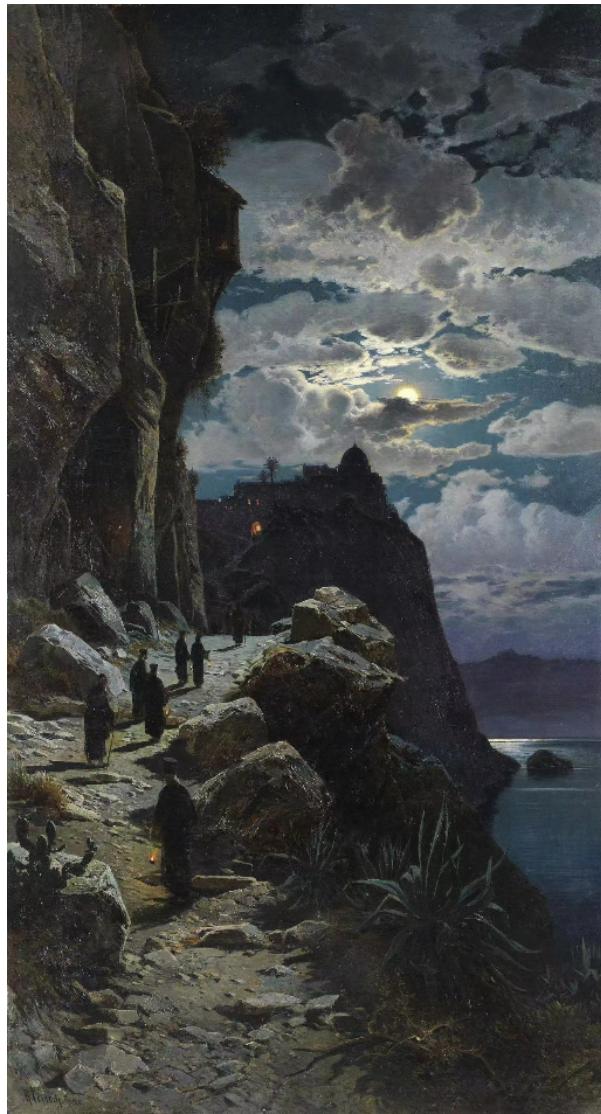
- 使用阻塞的方法，加入适当的逻辑处理寄存器写后读数据相关引发的流水线冲突，运行对应的 func，要求成功通过仿真和上板验证。

- 使用前递的方法，加入适当的逻辑处理寄存器写后读数据相关引发的流水线冲突，运行对应的 func，要求成功通过仿真和上板验证。
- 完成访存部分的设计，将代码接入模板工程，通过一级测试。

3 | 算术逻辑转移类指令的添加

僧侣步行到阿索斯山修道院 (赫尔曼·科洛迪)

路途中的一切，有些与我擦肩而过从此天各一方，有些便永久驻进我的心魂，雕琢我，塑造我，锤炼我，融入我而成为我。



~ · ~

添加指令的设计过程：

- 认真阅读指令系统规范，明确待实现指令的功能定义。
- 根据指令的功能定义考虑数据通路的设计调整，能复用的尽量复用，该新增的就新增。
- 根据调整后的数据通路，梳理所有指令（包括原有的指令和新增的指令）对应的控制信号。

需要添加的指令为

`slti, sltui, andi, ori, xori, sll.w, srl.w, sra.w, pcaddu12i, blt, bge, bltu, bgeu.`

通过比较已经完成的 20 条指令，不难发现，这些指令在各个阶段均可以复用原本已经实现的指令的数据通路，然后对流水线上的所需使用的控制信号进行调整，增加部分信号即可，不在此进行过多的描述。

3.1 | 实验要求

- 添加算术逻辑运算类指令 slti、sltui、andi、ori、xori、sll、srl、sra、pcaddu12i。运行对应的 func，要求成功通过仿真和上板验证。
- 添加转移指令 blt、bge、bltu、bgeu，运行对应的 func，要求成功通过仿真和上板验证。

4 | 访存指令的添加

跨越阿尔卑斯山圣伯纳隘道的拿破仑（雅克·路易·大卫）

就像一只回旋镖，你已经忘记曾经把他掷出，却在意想不到的时候飞回你手中。



~ . ~

4.1 | st.b 和 st.h 指令的添加

实现的关键在于控制如何实现字节或者半字的写入。

显然是通过字节写使能来实现的。例如 **st.b** 是写入一个字节的数据，所以他的写使能需要根据 **vaddr** 的最后两位进行判断。

```
00 -> 0001  
01 -> 0010  
10 -> 0100  
11 -> 1000
```

st.h 指令同理，最后只需要生成写数据即可。

```
00 -> 0011  
10 -> 1100
```

4.2 | ld.b、ld.h、ld.bu、ld.hu 指令的添加

分析与 **ld.w** 指令之间的异同：

- 他们之间的计算虚地址，操作数来源，虚实地址映射的方法规则是完全一样的
- 访存结果都写回 **rd** 项寄存器

- 差异仅仅是写回的数据的位宽是不同的

这四条指令的设计可以参考 `ld.w` 指令，并无太大的难度，唯一需要考虑的是如何从数据 RAM 中选择出所需要的内容，可以通过一个多路选择器去实现，选择信号是根据指令的访存地址的 **最低两位** 以及访存操作的类型信息共同决定的。可结合上面的 `st.b` 和 `st.h` 指令的添加进行理解实现，两者的实现方法大同小异。

不要忘记将取回内容扩展至 32 位，记得判断是符号扩展还是零扩展

5 | 乘除法指令的添加

记忆的永恒 (萨尔瓦多·达利)

时间是不断运动的永恒影像。



~ · ~

关于乘除法指令，在上学期的计组实验中已经有过初步的了解，但是当时直接使用了“*”与“/”直接实现，

关于“*”运算符，对于 Artix-7 系列的 FPGA，目前 Vivado 实现“*”运算符时会默认采用 DSP48 器件（内含固化的 16 位乘法器电路），所以最终实现的电路的时序通常不错，也几乎不消耗 LUT 资源。采用这种方式推导出的乘法器电路有两个 32 位数输入、一个 64 位输出，具有单周期延迟，即输入数据之后当拍就能输出结果。

关于“/”和“%”，我们这里禁止直接使用完成乘除法的运算，其主要的原因是如果直接使用这类运算符的话，Vivado 的综合工具将现出一个用 LUT 实现的单周期除法运算部件，时序会非常非常的差。

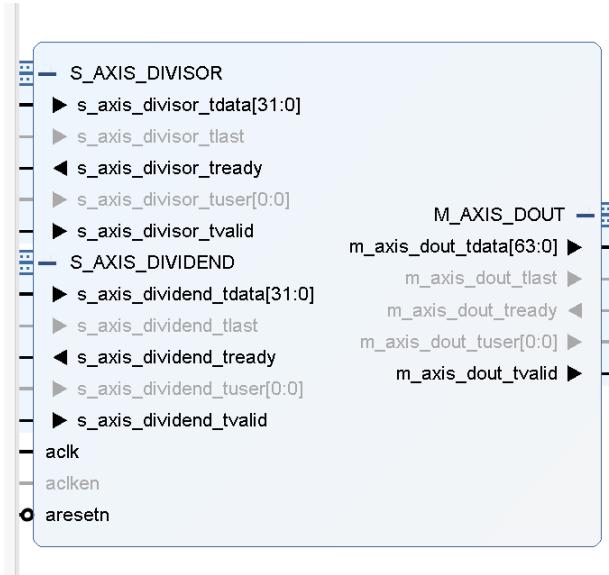
在这里我们给出大家两种关于乘除法指令实现的方法。

5.1 | 运用 Xilinx IP 的方法

对于乘法运算部件，根据上面所述，直接使用运算符实现即可。

对于除法运算部件，这里调用 IP 核实现，在工程栏的“IP Catalog”中，然后搜索“Divider Generator”，进入除法器的配置页面，在下面我们给出一点关于除法器配置的建议：

- 建议采用 Radix2 算法，它的优点是消耗的资源少，缺点是完成运算需要的迭代周期数较多。
- 需要实现有符号除和无符号除，建议定义两个除法器 IP。
- 将 Remainder Type 选择为 Remainder，确保余数的类型为整数型。不需要除 0 检测
- AXI 接口的流控这里可以选择 Non Blocking



- **s_axis_dividend_tdata** 对应计算时输入的被除数数据, **s_axis_divisor_tdata** 对应计算时输入的除数数据, **m_axis_dout_tdata** 中输出得到的商和余数, [63: 32] 位存放商, [31: 0] 存放余数。(上图中其他使用的信号, 可能大家并不会完全用到, 请根据自身使用情况进行调整, 不需要跟途中的除法器配置完全保证一致, 只需要保证功能可以正常实现即可)

对于两个输入通道和一个输出通道, 除了有上述的数据信号外, 每个通道都有一对 tvalid、tready 信号。这是一对“握手”控制信号, 其工作原理类似于我们在 CPU 流水线之间使用的 valid、allowin 信号。tvalid 是请求信号, tready 是应答信号。在时钟上升沿到来时, 如果采样得到 tvalid 和 tready 都等于 1, 则请求发起方和接收方之间完成一次成功的握手。如果我们假想接收方有一组触发器缓存, 那么所谓的成功握手是指发送方的数据写入接收方的缓存中, 也就是在握手成功的这个上升沿之后, 触发器缓存会变为发送方的数据。

我们假设在执行流水阶段调用所生成的除法器 IP。在除法指令处于执行流水级且没有对除法器成功输入数据的时候, 同时将 **s_axis_dividend_tvalid** 和 **s_axis_divisor_tvalid** 置为 1。当发现 **s_axis_dividend_tready** 和 **s_axis_divisor_tready** 反馈为 1 后(此时在一个时钟上升沿同时看到 tvalid 和 tready 为 1, 表示握手成功), 需要将 **s_axis_dividend_tvalid** 和 **s_axis_divisor_tvalid** 清 0, 也就是确保握手成功的那个时钟上升沿之后的 **s_axis_dividend_tvalid** 一定同时置为 1, 那么这里生成的除法器 IP 反馈的 **s_axis_dividend_tready** 和 **s_axis_divisor_tready** 一定也同时置为 1。这里再次强调, 被除数和除数输入的 tready 置起后 (也就是握手成功后), tvalid 一定要撤销, 否则对于除法器 IP 来说, 它会认为又有一个新的除法运算。

完成输入数据的握手之后, 除法指令就需要在执行流水级等待除法器 IP 最终输出结果。当 **m_axis_dout_tvalid** 置为 1 时, 表示除法计算完成。此时除法指令就可以从 **m_axis_dout_tdata** 上取出计算结果, 进入流水线的后续阶段。由于前面我们将流控策略设置为 Non Blocking, 因此输出通道上不需要外部反馈 tready 信号, 换言之, 不管产生多少结果、什么时候产生, 外部都要能够及时处理。

控制信号调整: 当执行流水阶段上是一条除法指令时, 仅当除法运算部件返回结果完成的信号之后这一级流水的 ready_go 才能置为有效。

5.2 | 实现电路级的乘法器，除法器

可以自行学习 Booth 编码和华莱士数实现乘法器，以及学习实现迭代除法器，这里不给出过多的介绍，有兴趣的同学可自行实现。

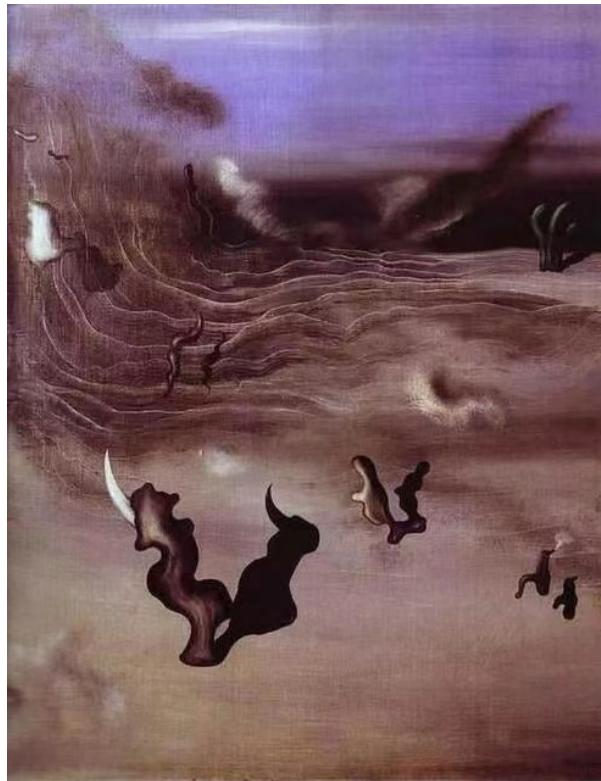
5.3 | 实验要求

- 添加乘除运算类指令 mul.w、mulh.w、mulh.wu、div.w、mod.w、div.wu、mod.wu, 运行对应的 func，要求成功通过仿真和上板验证。

6 | 异常和中断支持

风 (伊夫·唐吉)

我站在今天设想过去又幻想未来，过去和未来在今天随意交叉，因而过去和未来都刮着现在的风。



~ · ~

强调：本节内容请搭配手册使用

关于异常和中断的概念，在这里不进行过多的阐述，在 CPU 设计的角度来看，中断可以看成是一种特殊的异常，下文中统一使用异常来指代中断和异常。

异常处理的绝大部分工作是由异常处理程序（软件）的完成的，但在异常处理的开始和结束阶段则是硬件需要考虑的，这正是本节中我们需要考虑的。

异常处理的开始阶段 首先硬件需要判断异常的触发条件，随后硬件需要自动保存异常的类型，触发异常的指令，地址等信息以用来给异常处理程序。此外，硬件需要保证跳转到异常处理程序的入口执行，并且处于高特权等级。

异常结束阶段 需要跳转回到发生异常的指令处重新开始执行，并且保证特权等级回到最初环境的特权等级。

6.1 | 控制状态寄存器

上面提到，异常需要进行软硬件的协同处理，所以难以避免的需要进行硬件逻辑电路和异常处理软件的信息交互。为了保证精确异常的实现，LoongArch 指令系统中定义了一组独立的寄存器用于这类信息的交互，统称为控制状态寄存器（Control Status Register，简称 CSR）。

在本节内容中需要实现,设计的异常和中断的处理来说,相关的 CSR 有: CRMD、PRMD、ECFG、ESTAT、ERA、BADV、EENTRY、SAVE0~3、TID、TCFG、TVAL、TICLR。这些寄存器的详细定义请参看指令手册。

6.2 | 异常产生的条件判断

6.2.1 | 处理器核内部判定接收到中断

根据 LoongArch 指令系统的定义,每个处理器核内部可以记录 12 个线中断。除去应用于多核场景下的核间中断目前暂不考虑,还包括 8 个硬中断、2 个软中断和 1 个定时器中断。

硬中断可以理解为处理器核上有八个中断输入引脚,中断信号通过这八个引脚输入。ESTAT 控制状态寄存器 IS (Interrupt State) 域的 9 到 2 这八位 (RTL 上对应 8 个触发器) 则直接对中断输入引脚的信号采样。

软件中断顾名思义是由软件来设置的,通过 CSR 写指令对 ESTAT 状态控制寄存器 IS 域的 1..0 这两位写 1 或写 0 就可以完成两个软件中断的置起和撤销。

定时器中断的状态记录在 ESTAT 控制状态寄存器 IS 域的第 11 位。

从上面描述可知,ESTAT 的 IS 域的 11, 9..0 这 11 位记录了中断的 11 个状态,均为高电平表示有效。但是处理器核中是否接受中断还需要看中断的使能情况,分为两个层次:

低层次-局部中断使能,由 ECFG 的 LIE(local interrupt enable) 域的 11, 9..0 位依次对应控制
高层次-全局中断使能,通过 CRMD 的 IE(interrupt enable) 位来控制

综上可以给出处理器内部判断中断信号的 has_int 可以这样实现:

```
1 assign has_int = ((csr_estat_is[12:0] & csr_ecfg_lie[12:0]) != 13'b0)
2   && (csr_crmd_ie == 1'b1);
```

注: 在 la32r 中,不考虑多个中断的先后顺序,无论是接收到一个外部中断还是多个外部中断,此信号都可以被置为有效

6.2.2 | 核内定时器中断的产生

定时器中断经常用于操作系统的调度和计时功能的实现。该中断源来自于定时器,通常分为核外和核内两种实现方式。LoongArch 指令系统采用了核内实现定时器中断源的方式。简单来说,在每个 LoongArch32 处理器核内部实现一个 32 位的计数器,在开启定时功能后每个时钟周期减 1,当减到 0 值即可触发一次定时器中断。接下来介绍其定义细节:

- 定时器的软件配置集中在 TCFG (Timer Config) 控制状态寄存器,包括它的启动使能、倒计时初始值和倒计时模式。其中倒计时模式分为两种,一种是减到 0 后即停止计数,另一种是减到 0 后自动装载倒计时初始值再次开始新一轮倒计时。
- 定时器的时钟与 rdcontvl/h.w 指令所访问的计时器使用同一时钟,这个时钟要求频率固定,我们尚不考虑处理器核的变频和关停,所以可以采用处理器核流水线的时钟。
- 定时器当前的计数值仅可以通过读取 TVAL 状态寄存器近似获得,它与 rdcontvl/h.w 指令读取的计时器值来自于两个截然不同的对象。
- 当定时器倒计时到 0 时硬件将 ESTAT 控制状态寄存器 IS 域的第 11 位置 1,软件通过对 TICLR 控制寄存器的 CLR 位写 1 将 ESTAT 控制状态寄存器 IS 域的第 11 位清 0。

6.2.3 | 取指地址错异常 ADEF

要求所有指令的 PC 都是字对齐的（地址最低两比特全为 0），否则触发 ADEF。

检测逻辑：对取指所用的 PC 的最低两位进行判断，如果不是 $2'b00$ 的话，则置起取指地址错异常标志。我们推荐在 pre-IF 级就进行这一判断。从严格意义上讲，出现异常的取指地址不应该用来发起取指的请求，因为此时这个 PC 可能已经完全不正确，所以它的访存行为也不再软件人员的预想之内，最严重时可能导致死机等错误。

6.2.4 | 地址非对齐异常 ALE

地址非对齐仅针对 load, store 这类的访存指令。当 lh, lhu 和 sh 指令的地址最低位不为 0 时，或者 lw, sw 指令的地址最低两位不为全 0 时，触发 ALE。

检测逻辑：需对 load、store 指令的地址进行判断，当访存地址出现非对齐情况时，则置起地址非对齐异常标志。与前一节的设计思路一致，我们推荐在发起访存请求的 EXE 级进行上述判断，这样可以在发现异常情况的时候停止用错误地址发起访存请求。

6.2.5 | 指令不存在异常 INE

当取回的指令码不属于任何一条已实现的指令时，将触发 INE。

检测逻辑：可知指令不存在异常需要对指令译码后才得到检测结果。由于我们需要在译码阶段对每一条实现的指令进行解码生成控制信号，所以自然就可得到“不是任何一条实现的指令”的条件。

6.2.6 | 系统调用 SYS 和断点异常 BRK

当执行 SYSCALL 指令时触发系统调用异常，当执行 BREAK 指令时触发断点异常。

检测逻辑：译码发现是 syscall 指令就置起系统调用异常标志，译码发现 break 指令就置起断点异常标志。

6.3 | 精确异常的实现

为了实现精确异常，我们并不需要一发生异常就马上去修改控制状态寄存器和 PC。发生异常时仅需要考虑该如何处理那些在流水线中的指令。

这里给出一种处理思路：异常发生的判断逻辑分布在各流水级，靠近与之相关的数据通路；发现异常后将异常信息附着在指令上沿流水线一路携带下去，直至写回级才真正报出异常，此时才会根据所携带的异常信息更新控制状态寄存器；写回指令报出异常的同时，清空所有流水级缓存的状态，并将 nextPC 置为异常入口地址。

当然不一定在写回级报出异常，但必须保证该级之后的流水级不能产生新的异常，否则这就不能满足精确异常的要求。

6.4 | 控制状态存储器的实现

控制寄存器需要被 csrrd, csrwr, csrxchg 这样的 CSR 指令访问，又需要和各级流水端口交互，看起来又集中又分散，下面给出一些设计的建议：

- 把所有的控制状态寄存器集中到一个模块中实现；

- 模块接口分为用于指令访问的接口和与处理器核内部硬件电路逻辑直接交互的控制、状态信号接口两类；
- 指令访问接口包含读使能 (`csr_re`)、寄存器号 (`csr_num`)、寄存器读返回值 (`csr_rvalue`)、写使能 (`csr_we`)、写掩码 (`csr_wmask`) 和写数据 (`csr_wvalue`)；
- 与硬件电路逻辑直接交互的接口信号视需要各自独立定义，无须再统一编码，如送往预取指 (pre-IF) 流水级的异常处理入口地址 `ex_entry`、送往译码流水级的中断有效信号 `has_int`、来自写回流水级的 `ertn` 指令执行的有效信号 `ertn_flush`、来自写回流水级的异常处理触发信号 `wb_ex` 以及异常类型类型 `wb_ecode`、`wb_esubcode` 等。

下面给出一个具体的 CSR 端口设计：

接下来给出每个域的具体分析和部分实现举例：

6.4.1 | CRMD 的 PLV 域

从指令手册的定义可知 CRMD 的 PLV 域是可以通过 CSR 指令更新，而且在触发异常和 `ertn` 指令执行时也将被更新。

```

1  always @(posedge clock) begin
2      if (reset)
3          csr_crmd_plv <= 2'b0;
4      else if (wb_ex)
5          csr_crmd_plv <= 2'b0;
6      else if (ertn_flush)
7          csr_crmd_plv <= csr_prmd_pplv;
8      else if (csr_we && csr_num==`CSR_CRMD)
9          csr_crmd_plv <= csr_wmask[`CSR_CRMD_PLV]&csr_wvalue[`CSR_CRMD_PLV]
10             | ~csr_wmask[`CSR_CRMD_PLV]&csr_crmd_plv;
11

```

这个代码应该是比较好理解的，每一个写入条件和写入的值基本上都可以与指令手册中的定义一一对应上。实现时，重点关注 CSR 模块的各个输入信号没有接错即可。

6.4.2 | CRMD 的 IE 域

```

1  always @(posedge clock) begin
2      if (reset)
3          csr_crmd_ie <= 1'b0;
4      else if (wb_ex)
5          csr_crmd_ie <= 1'b0;
6      else if (ertn_flush)
7          csr_crmd_ie <= csr_prmd_pie;
8      else if (csr_we && csr_num==`CSR_CRMD)
9          csr_crmd_ie <= csr_wmask[`CSR_CRMD_PIE]&csr_wvalue[`CSR_CRMD_PIE]
10             | ~csr_wmask[`CSR_CRMD_PIE]&csr_crmd_ie;
11

```

请根据手册自行理解，对比 CRMD 的 IE 和 PLV 域的代码，可以发现分支条件基本一样，代码处理操作也是相似的。

6.4.3 | CRMD 的 DA、PG、DATF、DATM 域

目前我们设计的处理器还没有实现 MMU 的全部功能，仅支持直接地址翻译模式，所以 CRMD 的 DA、PG、DATF、DATM 域可以暂时置为常值。具体何值请根据手册自行判断。

6.4.4 | PRMD 的 PPLV、PIE 域

```

1  always @(posedge clock) begin
2      if (wb_ex) begin
3          csr_prmd_pplv <= csr_crmd_plv;
4          csr_prmd_pie <= csr_crmd_ie;
5      end
6      else if (csr_we && csr_num==`CSR_PRMD) begin
7          csr_prmd_pplv <= csr_wmask[`CSR_PRMD_PPLV]&csr_wvalue[`CSR_PRMD_PPLV]
8              | ~csr_wmask[`CSR_PRMD_PPLV]&csr_prmd_pplv;
9          csr_prmd_pie <= csr_wmask[`CSR_PRMD_PIE]&csr_wvalue[`CSR_PRMD_PIE]
10             | ~csr_wmask[`CSR_PRMD_PIE]&csr_prmd_pie;
11      end
12  end

```

6.4.5 | ECFG 的 LIE 域

请根据手册和上面示例代码自行实现

6.4.6 | ESTAT 的 IS 域

ESTAT 的 IS 域中，1..0 位、9..2 位、11 位、12 位的更新来源存在区别，其依次仅被 CSR 指令更新、仅通过采样处理器核硬件中断输入引脚更新、仅根据定时器计数器和 TICLR.CLR 域的写更新、仅通过采样处理器核的核间中断输入引脚更新。第 10 位没有定义，保险起见我们将其恒置为 0。

```

1  always @(posedge clock) begin
2      if (reset)
3          csr_estat_is[1:0] <= 2'b0;
4      else if (csr_we && csr_num==`CSR_ESTAT)
5          csr_estat_is[1:0] <= csr_wmask[`CSR_ESTAT_IS10]&csr_wvalue[
6              `CSR_ESTAT_IS10]
7                  | ~csr_wmask[`CSR_ESTAT_IS10]&csr_estat_is[1:0];
8
9          csr_estat_is[9:2] <= hw_int_in[7:0];
10
11         csr_estat_is[10] <= 1'b0;
12
13         if (timer_cnt[31:0]==32'b0)
14             csr_estat_is[11] <= 1'b1;
15         else if (csr_we && csr_num==`CSR_TICLR && csr_wmask[`CSR_TICLR_CLR]
16             && csr_wvalue[`CSR_TICLR_CLR])
17             csr_estat_is[11] <= 1'b0;
18
19         csr_estat_is[12] <= ipi_int_in;

```

20 end

6.4.7 | ESTAT 的 Ecode 和 EsubCode 域

ESTAT 的 Ecode 和 EsubCode 域需要在触发异常时填入异常的类型代号。见手册，我们推荐采用每个异常单独一个标志信号的传递方式，最后在写回级编码为 Ecode 和 EsubCode 值送到 CSR 模块。

```

1  always @(posedge clock) begin
2      if (wb_ex) begin
3          csr_estat_ecode <= wb_ecode;
4          csr_estat_esubcode <= wb_esubcode;
5      end
6  end

```

6.4.8 | ERA 的 PC 域

当位于写回级指令触发异常时，需要记录到 ERA 寄存器的 PC 就是当前写回级的 PC。根据手册和以上示例代码自行实现

6.4.9 | BADV 的 VAddr 域

BADV 的 VAddr 域和 ERA 的 PC 域的维护有相似之处，都是在写回级指令触发异常时，记录该指令的一些信息。

这里需要注意一点，在支持异常处理之前，处理器流水线中在访存级和写回级是不需要保存 load, store 指令完整的虚地址的。但是为了 BADV 的 VAddr 域的正常维护，我们需要在流水线中进行保存。

根据手册和以上示例代码自行实现

6.4.10 | EENTRY 的 VA 域

根据手册和以上示例代码自行实现

6.4.11 | SAVE0~3

SAVE0~3 就是提供给特权态软件临时存放数值用的，给出部分示例实现，自行补全完整

```

1  always @(posedge clock) begin
2      if (csr_we && csr_num==`CSR_SAVE0)
3          csr_save0_data <= csr_wmask[`CSR_SAVE_DATA]&csr_wvalue[`CSR_SAVE_DATA]
4          | ~csr_wmask[`CSR_SAVE_DATA]&csr_save0_data;
5      if (csr_we && csr_num==`CSR_SAVE1)
6          csr_save1_data <= csr_wmask[`CSR_SAVE_DATA]&csr_wvalue[`CSR_SAVE_DATA]
7          | ~csr_wmask[`CSR_SAVE_DATA]&csr_save1_data;
8  end

```

6.4.12 | TID

TID 寄存器的维护也是比较简单的，根据手册和示例自行实现

6.4.13 | TCFG 的 En、Periodic 和 InitVal 域

根据手册和示例自行实现

6.4.14 | TVAL 的 TimeVal 域

TVAL 的 TimeVal 域是一个软件只读域，它返回定时器计数器的值即可，所以可以将其实现为 wire 而非 reg。此处的设计关键点在于用作定时器的计数器 timer_cnt 的实现。

```

1  reg csr_tcfg_en;
2  reg csr_tcfg_periodic;
3  reg [29:0] csr_tcfg_initval;
4  wire [31:0] tcfg_next_value;
5  wire [31:0] csr_tval;
6
7  reg [31:0] timer_cnt;
8  assign tcfg_next_value = csr_wmask [31:0]&csr_wvalue[31:0]
9          | ~csr_wmask[31:0]&{csr_tcfg_initval,
10                           csr_tcfg_periodic, csr_tcfg_en};
11
12 always @ (posedge clock) begin
13     if (reset)
14         timer_cnt <= 32'hffffffff;
15     else if (csr_we && csr_num==`CSR_TCFG && tcfg_next_value[`CSR_TCFG_EN])
16         timer_cnt <= {tcfg_next_value[`CSR_TCFG_INITVAL], 2'b0};
17     else if (csr_tcfg_en && timer_cnt!=32'hffffffff) begin
18         if (timer_cnt[31:0]==32'b0 && csr_tcfg_periodic)
19             timer_cnt <= {csr_tcfg_initval, 2'b0};
20         else
21             timer_cnt <= timer_cnt - 1'b1;
22     end
23 end
24
25 assign csr_tval = timer_cn[31:0];

```

- 我们在软件对 timer 进行配置(也就是更新 TCFG 控制状态寄存器的时候)的同时发起 timer_cnt 的更新操作。具体来说，就是当软件开启 timer 的使能时(即将 TCFG 的 En 域置 1)，将此时写入的 timer 配置寄存器的定时器初始值更新到 timer_cnt 中；当软件关闭 timer 的使能时，timer_cnt 不更新。因为是在软件写 TCFG 的同时更新 timer，所以要看当前写入 TCFG 寄存器的值(tcfg_next_value)，而不是用 TCFG 寄存器已有的值。
- 当 timer_cnt 减到全 0 且定时器不是周期性工作模式的情况下，代码中没有专门处理的逻辑，所以 timer_cnt 会继续减 1 变成 32'hffffffff。不过，因为此时定时器是非周期性的，所以它应该停止计数，这就是为何 timer_cnt 自减的使能条件除了看 csr_tcfg_en 是否为 1 外还会看 timer_cnt!=32'hffffffff 这个条件

6.4.15 | TICLR 的 CLR 域

TICLR 的 CLR 域很特殊，它的读写属性是“W1”，意味着软件只有对它写 1 才会产生执行效果(即硬件只捕获对 TICLR 的 CLR 域写 1 这个动作)，写 0 无效，同时软件读出的值永远是 0。

所以，TICLR 的 CLR 域并不需要定义一个 reg 与之对应，我们只需要定义一个恒为 0 的 wire 用于后面的 CSR 读出即可。

6.4.16 | CSR 的读出

给出示例如：

```

1  wire [31:0] csr_prmd_rvalue = {29'b0, csr_prmd_pie, csr_prmd_pplv};
2  wire [31:0] csr_ecfg_rvalue = {19'b0, csr_ecfg_lie};
3
4  assign csr_rvalue = {32{csr_num==`CSR_CRMD}} & csr_crmd_rvalue
5          | {32{csr_num==`CSR_PRMD}} & csr_prmd_rvalue

```

6.5 | 相关冲突的处理

最经典的 CSR 写后读相关是 `csrwr` 或者 `csrxchg` 这两个指令修改一个 CSR 后又有 `csrrd`, `csrwr` 或者 `csrxchg` 指令读取同一个 CSR，为了解决这种冲突，最简单有效的方法就是将所有的 CSR 读写指令访问 CSR 的操作都放到同一流水级进行处理，虽然会损失一些性能，但是影响并不是很大。

写者	相关对象	读者
<code>csrwr</code> 或 <code>csrxchg</code>	CRMD.IE、ECFG.LIE、ECFG.IS[1:0]、TCFG.En、TICLR.CLR ERA、PRMD.PPLV、PRMD.PIE	译码级的指令（标记中断） <code>ertn</code>
<code>ertn</code>	CRMD.IE CRMD.PLV	译码级的指令 取指的 PC

我们这里处理冲突最主要的还是用阻塞，并不考虑前递。

前面三种情况的的阻塞仍是直截了当，只需要判断 EXE, MEM, WB 级的时候有没有这几种情况的写相关性的写者，如果有就直接阻塞（阻塞在译码级）。

第四种情况，在流水线中最早只能在译码的时候能知道写者和相关对象，但是这个时候取值级已经取出来一个指令了，现在单靠阻塞是不能解决问题了，这里给出一种特殊的处理方法：`ertn` 指令直到写回级才修改 CRMD，与此同时清空流水线并更新取指 PC。这也就是前面提到的 `ertn_flush` 信号的由来。

6.6 | 实验要求

- 为 CPU 增加 `csrrd`、`csrwr`、`csrxchg` 和 `ertn` 指令
- 为 CPU 增加控制状态寄存器 CRMD、PRMD、ESTAT、ERA、EENTRY、SAVE0~3。
- 为 CPU 增加 syscall 指令，实现系统调用异常支持。
- 运行对应的 func，要求成功通过仿真和上板验证。
- 为 CPU 增加取指地址错（ADEF）、地址非对齐（ALE）、断点（BRK）和指令不存在（INE）异常的支持。
- 为 CPU 增加中断的支持，包括 2 个软件中断、8 个硬件中断和定时器中断。

- 为 CPU 增加 rdcontvl.w、rdcntvh.w 和 rdcontid 指令。
- 为 CPU 增加控制状态寄存器 ECFG、BADV、TID、TCFG、TVAL、TICLR。
- 运行对应的 func，要求成功通过仿真和上板验证。

7 | 结语

希望大家能感受，享受来自数字逻辑的魅力。

A | LA 组评分细则

A.1 | 任务要求

A.1.1 | 写在开头

个人任务需要个人独立完成验收，创新扩展部分在答辩环节进行展示，需要在答辩中展示创新扩展的正确性。

- 关于答辩会提的问题：

- 主要会围绕着创新内容部分进行考察，考察原理和实现思路。

- 关于评分与难度的非线性关系：

- 之所以存在这种非线性关系，主要是为了一般的同学能够拿到分数，对此感兴趣的同學可以深入。不喜欢这个方向就没必要卷计组课设了，抓紧时间学好其他专业课。

- 关于分数分布

- 本课程的总分 100 分，难度最大的部分其实在创新扩展部分，而其他的部分都只要认真把握课程原理，即可实现，请同学们重点把握这部分知识，充分理解；
 - 不建议大家死磕创新扩展的实现，**更禁止通过抄袭的方式获得这一部分分数**，请大家实事求是。

A.1.2 | 个人任务 60 分

根据给出的实验指导书完成前五个章节：

- 不考虑冲突的简单五级流水划分；
- 指令相关和流水线冲突；
- 算术逻辑转移类指令的添加；
- 访存指令的添加；
- 乘除法指令的添加。

实验报告——个人部分 10 分

A.1.3 | 团队任务 40 分

创新基础部分 异常相关指令扩展（根据指导书的最后一章，实现异常和中断的支持）：

- 第一阶段：

- 增加 csrrd、csrwr、csrxchg 和 ertn 指令；
 - 为 CPU 增加控制状态寄存器 CRMD、PRMD、ESTAT、ERA、EENTRY、SAVE0-3；
 - 为 CPU 增加 syscall 指令，实现系统调用异常支持。

- 第二阶段：

- 增加取指地址错（ADEF）、地址非对齐（ALE）、断点（BRK）和指令不存在（INE）异常的支持；
- 增加中断的支持，包括 2 个软件中断、8 个硬件中断和定时器中断；
- 增加 rdcontvl.w、rdcntvh.w 和 rdcontid 指令；
- 增加控制状态寄存器 ECFG、BADV、TID、TCFG、TVAL、TICLR。

创新扩展部分 给出多个选项参考，并不需要大家全部完成，如有其他自行设计的创新扩展不在本范围内也可。

■ AXI 总线接口设计：

- 将 CPU 顶层接口修改为 AXI 总线接口。

■ 缓存模块设计：

Cache 是现代计算机中不可缺少的技术。

- Cache 基本实现为一路直接映射 Cache，二者都实现时取 DataCache 成绩；
- Cache 指令实现；
- 多路组相连 Cache；
- Cache 流水化。

■ 外设功能扩展

- 通过汇编语言代码对 SoC 上已经提供实现了的外设进行操控；
- 集成串口外设，并能通过汇编语言代码操纵外设。

实验报告——团队部分 10 分

B | MIPS 组评分细则

B.1 | 任务要求

B.1.1 | 写在开头

个人任务需要个人独立完成验收，创新扩展部分在答辩环节进行展示，需要在答辩中展示创新扩展的正确性。

- 关于答辩会提的问题：
 - 主要会围绕着创新内容部分进行考察，考察原理和实现思路。
- 关于评分与难度的非线性关系：
 - 之所以存在这种非线性关系，主要是为了一般的同学能够拿到分数，对此感兴趣的同学可以深入。不喜欢这个方向就没必要卷计组课设了，抓紧时间学好其他专业课。
- 关于分数分布
 - 本课程的总分 100 分，难度最大的部分其实在创新扩展部分，而其他的部分都只要认真把握课程原理，即可实现，请同学们重点把握这部分知识，充分理解；
 - 不建议大家死磕创新扩展的实现，**更禁止通过抄袭的方式获得这一部分分数**，请大家实事求是。

B.1.2 | 个人任务 60 分

CG 测评任务 20 分 完成 CG 上评测题，需要提交的代码已经在 CDE 工程中给出。请阅读实验指导书《计算机组成原理课程设计实验指导书下》，学习 TinyMIPS 的基本结构。并提交相应代码。

虚拟仿真实验平台 10 分

- 完成虚拟仿真实验平台上的四个实验。包括模拟器运行和代码运行；
- 本部分实验重点在于解决流水线的前递与暂停问题，与实验指导书的第二章结尾和第三章结尾有关。请注意体会其原理，在进行乘除法器的扩展过程中，同样需要利用到前递和暂停的知识。

扩展指令任务 20 分

- 要求选择 TinyMIPS CPU 中未扩展的 22 条指令中的若干条进行扩展，具体包含的指令在表 B.1，对于 ADD, SUB 等涉及异常的指令，暂时可以不用扩展异常处理的内容。
- 对于每一条指令，都有若干其所属的分组和一个基础分值。
- 每个指令的基础分值根据扩展该指令时的工作量进行设置，具体见表 B.1。
- 如果选择了多条同分组的指令，自第二条起，其计算分值减少为基础分值的 1/2。
- 要求最终同学们完成计算分值总计 10 分的指令扩展。多做不加分。
- 个人任务需要到现场进行验收，演示对指令的测试过程。验收现场会进行提问，提问内容不仅涉及下面的指令扩展的思路过程，也包括 Trace 机制，测试用例，MIPS 指令集架构相关的知识。

表 B.1: 指令分值

序号	指令名称	指令分值	指令分组
1	ADD	2	R 型运算指令/加法指令
2	SUB	2	R 型运算指令
3	NOR	2	R 型运算指令
4	ADDI	2	I 型运算指令/加法指令
5	ANDI	2	I 型运算指令
6	ORI	2	I 型运算指令
7	XORI	2	I 型运算指令
8	SLTI	2	I 型运算指令
9	SLTIU	2	I 型运算指令
10	SRL	2	Shamt 移位指令
11	SRA	2	Shamt 移位指令
12	J	2	直接跳转指令
13	JR	3	直接跳转指令
14	LH	3	内存载入指令
15	LHU	3	内存载入指令
16	SH	3	内存存储指令
17	BGEZ	3	条件分支指令/BGE 跳转
18	BGTZ	3	条件分支指令
19	BLEZ	3	条件分支指令
20	BLTZ	3	条件分支指令/BLT 跳转
21	BGEZAL	3	条件分支指令/BGE 跳转
22	BLTZAL	3	条件分支指令/BLT 跳转

例如一个同学选择了 ADD 指令, ORI 指令, ANDI 指令, JR 指令, LH 指令, 其总分值为 2, 2, 1, 3, 3 共计 11 分, 可以达到验收要求。

实验报告——个人部分 10 分

B.1.3 | 团队任务 40 分

创新基础部分 创新部分基础实现包括《计算机组成原理课程设计指导书补充内容》所介绍的全部内容:

■ 乘除法指令扩展:

- 通过乘除法测试点, 包括 n44_div, n45_divu, n46_mult, n47_multu, n48_mfhi, n49_mflo, n50_mthi, n51_mflo;
- 乘除法模块可以采用 IP 核实现。

■ 异常相关指令扩展:

- 第一阶段:
 - 实现特权指令 ERET, MTC0, MFC0
 - 自陷指令 BREAK, SYSCALL
 - 实现 A02 文件中规定的所有 CP0 寄存器
- 第二阶段:
 - 继续扩展异常, 通过完整 89 个功能点的测试

创新扩展部分 给出多个选项参考, 并不需要大家全部完成, 如有其他自行设计的创新扩展不在本范围内也可。

■ 缓存模块设计:

Cache 是现代计算机中不可缺少的技术。

- Cache 基本实现为一路直接映射 Cache, 二者都实现时取 DataCache 成绩;
- Cache 指令实现;
- 多路组相连 Cache;
- Cache 流水化。

■ 外设功能扩展

- 通过汇编语言代码对 SoC 上已经提供实现了的外设进行操控;
- 集成串口外设, 并能通过汇编语言代码操纵外设。

实验报告——团队部分 10 分