实验 7 报告

组员 1 ：袁欣怡　2018K8009929021
组员 2 ：郑旭舟　2018K8009908047
箱子号：10

一、实验任务

实验目的:

在已有简单的流水线 CPU 上添加转移指令 (BGEZ, BGTZ, BLEZ, BLTZ, J, BLTZAL, BGEZAL, JALR) 和访存指令 (LB, LBU, LH, LHU, LWL, LWR, SB, SH, SWL, SWR) .

主要操作:

配合 CPU 流水级的设计，根据需要调整已有的数据通路或添加新的数据通路。

二、实验设计

（一）总体设计思路

本组设计对应的 CPU 硬件设计框图如下，本次完善了 `BR_UNIT` 等部分的逻辑，故图上未显示显著的新数据通路。

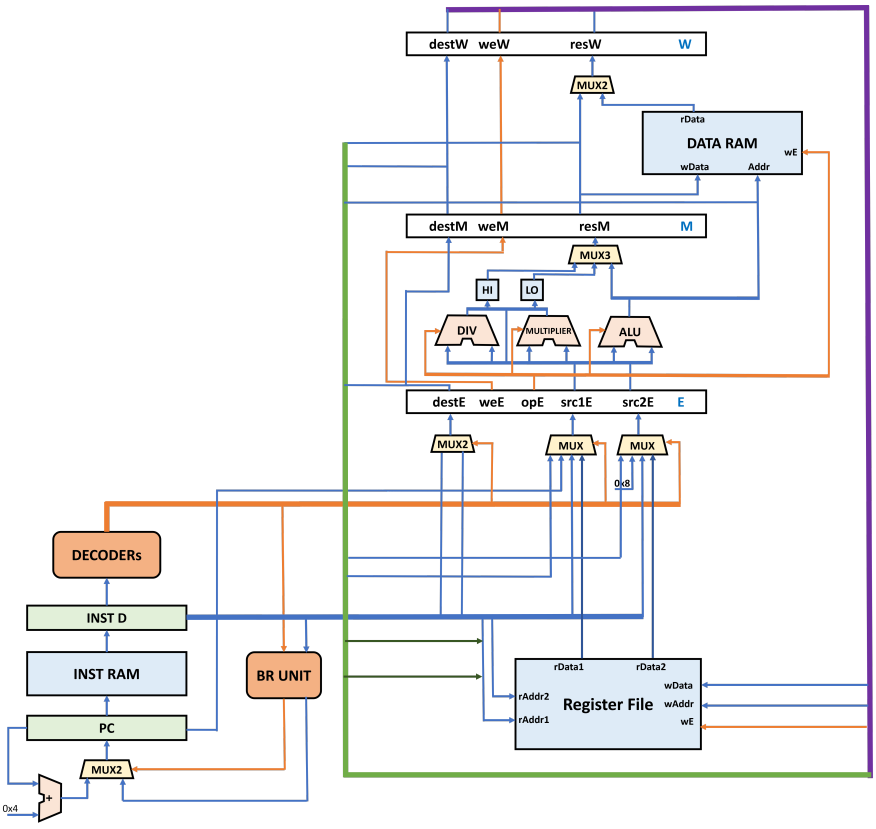

图 1

（二）重要模块 1：译码模块 (ID_stage)

1. 工作原理

译码模块 ID_stage 的主要功能如下：

- 对取指模块 IF_stage 中取到的指令进行译码，得到指令类型、ALU 运算类型、是否阻塞、是否跳转等信息；
- 将译码后的信息通过总线传递给执行模块 EXE_stage；
- 根据写回模块 WB_stage 传入的寄存器写控制信息写寄存器堆。

2. 接口定义

表 1

名称	方向	位宽	功能描述
`clk`	IN	1	时钟信号
`reset`	IN	1	复位信号
`es_allowin`	IN	1	EXE 模块允许接受来自 ID 的数据
`ds_allowin`	OUT	1	ID 模块允许接受来自 IF 的数据
`fs_to_ds_valid`	IN	1	IF 模块向 ID 模块传递的数据有效信号
`fs_to_ds_bus`	IN	64	从 IF 模块到 ID 模块的数据总线
`es_to_ds_bus`	IN	38	从 EXE 模块到 ID 模块的数据总线（前递）
`ms_to_ds_bus`	IN	37	从 MEM 模块到 ID 模块的数据总线（前递）
`ds_to_es_valid`	OUT	1	ID 模块向 EXE 模块传递的数据有效信号
`ds_to_es_bus`	OUT	**152**	从 ID 模块到 EXE 模块的数据总线； **增加了 8 位*
`br_bus`	OUT	34	从 ID 模块到 IF 模块的跳转信息数据总线
`ws_to_rf_bus`	IN	38	从 WB 模块到 ID 模块的写寄存器数据总线

3. 功能描述

译码模块 ID_stage 最主要的功能是对指令进行译码，并且判断属于何种指令。

为了解决数据冲突，根据指令的操作数来源，将它们分类成 `type_st`，`type_rs`，`type_rt`，`type_nr` 四类，方便根据指令需要的源寄存器情况判断是否需要进行阻塞。

(1) 支持新增的访存指令

对于新增的访存指令 (load/store)，由于 load 和 store 的主要功能分别要在执行模块 EXE_stage 和访存模块 MEM_stage 中实现，因此需要把当条访存指令的类型和访存地址通过总线传递到后两个模块中。

为此，设计时添加了信号 `ls_type` 和 `ls_laddr` ：

- `ls_type` 指示 load/store 指令的类型:
    - 最高位指示该操作是否要求对立即数做扩展时使用无符号扩展 (仅在 load 指令的后续处理中用到) ;
    - 低五位采用 one-hot 编码，分别表示“字右非对齐访问”、“字左非对齐访问”、“半字访问”、“字节访问”、“整字访问”;
- `ks_laddr` 记录地址的低两位信息;

修改后，对应部分主要代码如下:

```
// lab7 newly added: load/store type & laddr
assign ls_type = {
            inst_lhu | inst_lbu ,          // [5] unsigned extension
            inst_lwr | inst_swr ,          // [4] l/s word right
            inst_lwl | inst_swl ,          // [3] l/s word left
            inst_lh  | inst_lhu | inst_sh, // [2] l/s halfword
            inst_lb  | inst_lbu | inst_sb, // [1] l/s byte
            inst_lw  | inst_sw             // [0] l/s word
               };
assign ls_laddr = {2{mem_re | mem_we}} & func[1:0];
```

(2) 支持新增的跳转指令

为了实现新增的跳转指令 (branch/jump) ， 需要在 `br_taken` 的产生逻辑中增加新的跳转条件判断。

修改后，对应部分主要代码如下:

```
assign rs_be_0 = ~rs_value[31] ;              // rs>=0
assign rs_bt_0 = ~rs_value[31] & ( |rs_value); // rs> 0
assign rs_se_0 =  rs_value[31] | (~|rs_value); // rs<=0
assign rs_st_0 =  rs_value[31] ;              // rs< 0
// if branch?
assign br_taken = ( inst_beq    &  rs_eq_rt    // lab7 logic modified
                  | inst_bne    & ~rs_eq_rt
                  | inst_bgez   &  rs_be_0
                  | inst_bgtz   &  rs_bt_0
                  | inst_blez   &  rs_se_0
                  | inst_bltz   &  rs_st_0
                  | inst_bltzal &  rs_st_0
                  | inst_bgezal &  rs_be_0
                  | inst_jal
                  | inst_jalr
                  | inst_j
                  | inst_jr
                  ) && ds_valid;
```

（三）重要模块 2： 执行模块 (EXE_stage)

1. 工作原理

执行模块 EXE_stage 执行从译码模块 ID_stage 获得的指令，并将得到的结果传递给访存模块 MEM_stage；

该模块在进行一般的算术逻辑运算时需要调用 ALU 模块，在进行乘除法时需要调用乘法器和除法器。

2. 接口定义

表 2

名称	方向	位宽	功能描述
`clk`	IN	1	时钟信号
`reset`	IN	1	复位信号
`ms_allowin`	IN	1	MS 模块允许接受来自 EXE 模块的数据
`es_allowin`	OUT	1	EXE 模块允许接受来自 ID 模块的数据
`ds_to_es_valid`	IN	1	ID 模块向 EXE 模块传递的数据有效信号
`ds_to_es_bus`	IN	145	从 ID 模块到 EXE 模块的数据总线
`es_to_ms_valid`	OUT	1	EXE 模块向 MEM 模块传递的数据有效信号
`es_to_ms_bus`	OUT	**111**	从 EXE 模块到 MS 模块的数据总线; **增加了 30 位*
`es_to_ds_bus`	OUT	38	从 EXE 模块到 ID 模块的数据总线（前递）
`data_sram_en`	OUT	1	data_sram 读使能
`data_sram_wen`	OUT	4	data_sram 写使能
`data_sram_addr`	OUT	32	data_sram 目标地址
`data_sram_wdata`	OUT	32	data_sram 写数据

3. 功能描述

执行模块 EXE_stage 接收译码模块 ID_stage 传来的指令信号，并执行对应功能。

因为新加的 store 指令涉及写半字等的操作，所以设计了 `write_strb` 信号，用来标识对应地址各字节上的数据是否可写。

为了提升代码的可读性和规范性，本组在设计时添加了 `wtire_strb_swr` , `wtire_strb_swl` , `wtire_strb_sh` , `wtire_strb_sb` 四个信号，以根据低地址信息 `ls_laddr` 给出不同的 store 指令下正确的 `write_strb` ：

```
assign write_strb_swr = {4{ es_ls_laddr_d[0]}} & 4'b1111 // SWR
```

```
                    | {4{ es_ls_laddr_d[1]}} & 4'b1110
                    | {4{ es_ls_laddr_d[2]}} & 4'b1100
                    | {4{ es_ls_laddr_d[3]}} & 4'b1000;
assign write_strb_swl = {4{ es_ls_laddr_d[0]}} & 4'b0001 // SWL
                    | {4{ es_ls_laddr_d[1]}} & 4'b0011
                    | {4{ es_ls_laddr_d[2]}} & 4'b0111
                    | {4{ es_ls_laddr_d[3]}} & 4'b1111;
assign write_strb_sh  = {4{~es_ls_laddr[1]  }} & 4'b0011 // SH
                    | {4{ es_ls_laddr[1]  }} & 4'b1100;
assign write_strb_sb  = {4{ es_ls_laddr_d[0]}} & 4'b0001 // SB
                    | {4{ es_ls_laddr_d[1]}} & 4'b0010
                    | {4{ es_ls_laddr_d[2]}} & 4'b0100
                    | {4{ es_ls_laddr_d[3]}} & 4'b1000;
```

产生这四个信号后，根据总线传输到执行模块 EXE_stage 的 store 指令类型 `ls_type` 判断具体应该选择输出的 `write_strb`：

```
assign write_strb = {4{ es_ls_type[4]}} & write_strb_swr // SWR
                  | {4{ es_ls_type[3]}} & write_strb_swl // SWL
                  | {4{ es_ls_type[2]}} & write_strb_sh  // SH
                  | {4{ es_ls_type[1]}} & write_strb_sb  // SB
                  | {4{ es_ls_type[0]}} & 4'b1111;       // SW
```

最后，`write_strb` 信号会成为 `data_sram_wen` 信号生成逻辑的一部分：

```
assign data_sram_wen = es_mem_we & es_valid ? write_strb : 4'h0;
```

（四）重要模块 3：访存模块 (MEM_stage)

1. 工作原理

访存模块 MEM_stage 从执行模块 EXE_stage 获取访存指令并执行，具体如下：

- 根据 `es_to_ms_bs` 中的 `ms_mem_re` 信号确定是否使用从数据 RAM 中读出的 `mem_result` ；
- 选择相应指令的最终结果，并通过总线传递给写回模块 WB_stage ；
- 将通用寄存器写使能、写地址控制信号等通过总线传递给写回模块 WB_stage ；

2. 接口定义

表 3

名称	方向	位宽	功能描述
`clk`	IN	1	时钟信号
`reset`	IN	1	复位信号
`ws_allowin`	IN	1	WB 模块允许接受来自 MEM 模块的数据
`ms_allowin`	OUT	1	MEM 模块允许接受来自 EXE 模块的数据
`es_to_ms_valid`	IN	1	EXE 模块向 MEM 模块传递数据的有效信号
`es_to_ms_bus`	IN	**111**	EXE 模块到 MEM 模块的数据总线; **增加了 30 位*
`ms_to_ws_valid`	OUT	1	MEM 模块向 EXE 模块传递数据的有效信号
`ms_to_ws_bus`	OUT	70	MEM 模块到WB模块的数据总线
`ms_to_ds_bus`	OUT	37	MEM 模块到 ID 模块的数据总线（前递）
`data_sram_rdata`	OUT	32	data_sram 读出的数据

3. 功能描述

对于新增的多种 load 指令，访存模块 MEM_stage 需要分析从执行模块 EXE_stage 传递过来的 `ls_addr` ，并且向写回阶段传递正确的寄存器堆写控制信号（写使能和写地址等）。

`mem_result` 根据指令类型和低位地址来处理从 data_sram 得到的数据。

对于低位地址，本组的设计借鉴了本实验 CPU 框架中译码模块 ID_stage 的信号译码思路，在访存模块 MEM_stage 对 `ls_laddr` 做 one-hot 编码处理得到信号 `ls_laddr_d[3:0]` ：

```
// lab7 newly added: ls_laddr decoded(one-hot)
assign ms_ls_laddr_d[3] = (ms_ls_laddr==2'b11);
assign ms_ls_laddr_d[2] = (ms_ls_laddr==2'b10);
assign ms_ls_laddr_d[1] = (ms_ls_laddr==2'b01);
assign ms_ls_laddr_d[0] = (ms_ls_laddr==2'b00);
```

为了提升代码的可读性和规范性，设计者在访存模块 MEM_stage 中设计了 `mem_res_<load_type>` 信号 (load_type = lwr, lwl, lhg, lbg)，以在不同的 load 指令下根据低地址信息 `ls_laddr` 计算正确的待载入数据：

```
// generate result under each situation of instructions
assign mem_res_lwr = {32{ms_ls_laddr_d[0]}} &  data_sram_rdata[31:0]                     // LWR
                   | {32{ms_ls_laddr_d[1]}} & {ms_rt_value[31:24], data_sram_rdata[31: 8]}
                   | {32{ms_ls_laddr_d[2]}} & {ms_rt_value[31:16], data_sram_rdata[31:16]}
                   | {32{ms_ls_laddr_d[3]}} & {ms_rt_value[31: 8], data_sram_rdata[31:24]};
assign mem_res_lwl = {32{ms_ls_laddr_d[0]}} & {data_sram_rdata[ 7:0], ms_rt_value[23:0]}   // LWL
                   | {32{ms_ls_laddr_d[1]}} & {data_sram_rdata[15:0], ms_rt_value[15:0]}
                   | {32{ms_ls_laddr_d[2]}} & {data_sram_rdata[23:0], ms_rt_value[ 7:0]}
                   | {32{ms_ls_laddr_d[3]}} &  data_sram_rdata;
assign mem_res_lhg = {32{~ms_ls_laddr[1] }} & {{16{mem_res_s_15}}, data_sram_rdata[15: 0]} // LH/LHU
                   | {32{ ms_ls_laddr[1] }} & {{16{mem_res_s_31}}, data_sram_rdata[31:16]};
assign mem_res_lbg = {32{ms_ls_laddr_d[0]}} & {{24{mem_res_s_07}}, data_sram_rdata[ 7: 0]} // LB/LBU
                   | {32{ms_ls_laddr_d[1]}} & {{24{mem_res_s_15}}, data_sram_rdata[15: 8]}
                   | {32{ms_ls_laddr_d[2]}} & {{24{mem_res_s_23}}, data_sram_rdata[23:16]}
                   | {32{ms_ls_laddr_d[3]}} & {{24{mem_res_s_31}}, data_sram_rdata[31:24]};
// generate mem_result
assign mem_result = {32{ms_type_lwr}} & mem_res_lwr       // LWR
                  | {32{ms_type_lwl}} & mem_res_lwl       // LWL
                  | {32{ms_type_lhg}} & mem_res_lhg       // LH/LHU
                  | {32{ms_type_lbg}} & mem_res_lbg       // LB/LBU
                  | {32{ms_type_lw }} & data_sram_rdata ; // LW
```

值得注意的是，在处理 LH/LHU, LB/LBU 指令时，`ls_type` 的最高位发挥了作用。以 LB/LBU 指令为例，它们的区别在于对从 data_sram 得到的数据做的扩展是有符号扩展还是无符号扩展，因此只需要根据具体的指令选择高位。

对一个字 (word) 中每个字节 (Byte) 的高位设置一个位宽为 1 的信号 `mem_res_s_<bit-location>` (bit-location= 7, 15, 23, 31)：

- 当需要做有符号扩展时，该信号置为当字节的最高位；
- 当需要做无符号扩展时，该信号置为 0；

对应的代码如下：

```
assign mem_res_s_07 = ~ms_ls_type[5] & data_sram_rdata[ 7];
assign mem_res_s_15 = ~ms_ls_type[5] & data_sram_rdata[15];
assign mem_res_s_23 = ~ms_ls_type[5] & data_sram_rdata[23];
assign mem_res_s_31 = ~ms_ls_type[5] & data_sram_rdata[31];
```

三、实验过程

（一）实验流水账

- 10 月 24 日，上午优化了一部分译码逻辑，下午 debug，调试完一处 bug 后测试跑通，开始撰写实验报告；
- 10 月 26 日，补充设计相关的信息，完善实验报告；
- 10 月 27 日，进一步完善报告中的硬件框图。

（二）错误记录

1. 错误 1 : 赋值符号两侧信号位宽不等导致的错误

(1) 错误现象

行为仿真时， tcl console 输出如下：

```
----[1643115 ns] Number 8'd67 Functional Test Point PASS!!!
--------------------------------------------------------------
[1645607 ns] Error!!!
    reference: PC = 0xbfc2b6d0, wb_rf_wnum = 0x02, wb_rf_wdata = 0x8c79c07b
    mycpu    : PC = 0xbfc2b6d0, wb_rf_wnum = 0x02, wb_rf_wdata = 0x0079c07b
```

发现 `rf_wdata` 最低的字节被错误地置为了 8'h0 .

(2) 分析定位过程

在反汇编文件 `test.s` 中找到该 `PC` 对应的指令：

```
bfc2b6d0:   8d02760c    lw  v0,30220(t0)
```

这是一条 load 指令，因此先检查该指令在 MEM 阶段和 WB 阶段的波形：

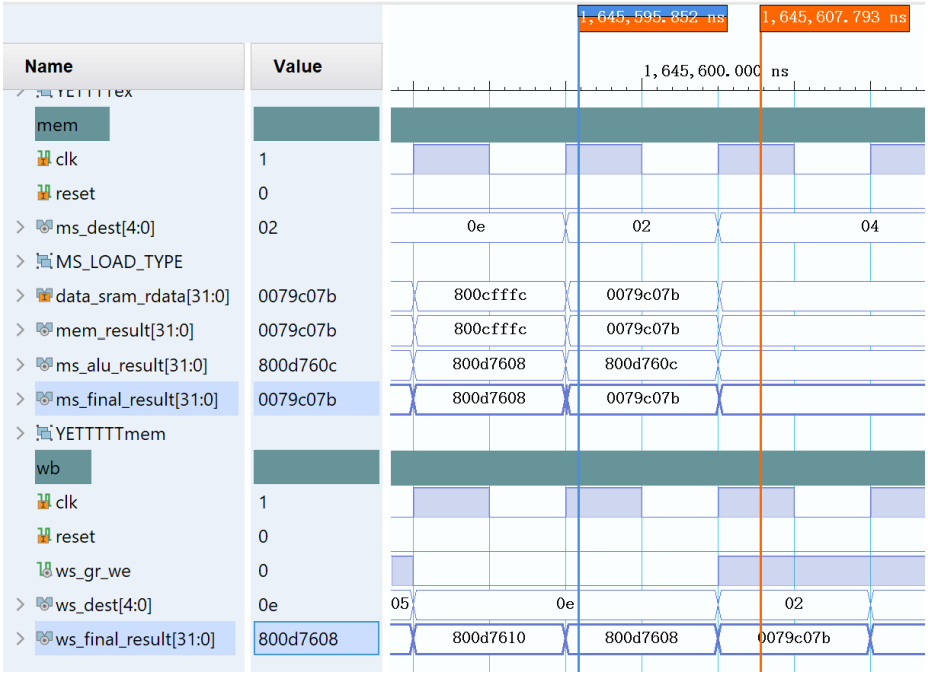

图 2

从图中可以看到，不论是访存阶段的 `mem_result` 和 `ms_final_result` ，还是写回阶段的 `ws_final_result` ，都是这个错误的数据，可以推断出是前一次在内存的 `30220(t0)` 位置存错了数据。

在反汇编中找到前一条改动内存的 `30220(t0)` 位置的字 (Word) 的指令：

```
bfc2b6bc:   b909760f    swr t1,30223(t0)
```

这是一条非对齐访存的 SWR 指令，在小端架构下，它访问的是该字的最低字节。

因为 store 指令是在 EXE 阶段向数据 RAM 输入有效的写内存信息的，所以检查这条 SWR 指令在 EXE 阶段的波形：

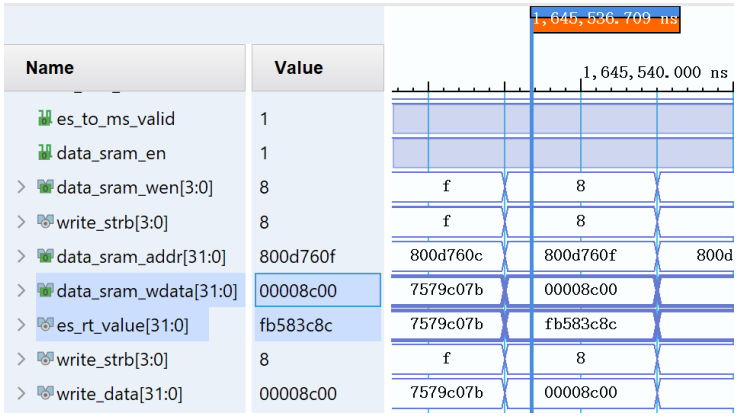

图 3

可以看到，源数据 `es_rt_value` 是 0xfb583c8c，`write_strb` 是 `1000`，但是 `write_data` 并不是预期的 0x8c000000，'8c' 这个字节被诡异地放在了 `write_data` 的中部，所以必然是 `write_data` 的中 SWR 指令对应的赋值逻辑出错了。

检查对应的代码：

```
assign write_data = ... &  ...                                  // SWR
                    ... & {es_rt_value[ 7:0],  8'b0} // ERROR
                    ...
                   ...;
```

发现原本应该是 `{es_rt_value[7:0],24'b0}` 的地方写成了 `{es_rt_value[7:0],  8'b0}`，位宽对不上，在仿真中高位就自动被置 0 了。

(3) 错误原因

给 EXE 阶段的 `write_data` 赋值时，赋值符号右侧信号位宽不够，导致最终产生的数据出错。

(4) 修正效果

将 `write_data` 中错误的赋值逻辑改正 (8'b0 改为 24'b0)，接着仿真就 PASS 了。

```
----[1658115 ns] Number 8'd68 Functional Test Point PASS!!!
================================================================
Test end!
----PASS!!!
$finish called at time : 1658575 ns : File
"C:/labs_ca/ucas_ca_lab_2020/lab7/CPU_CDE/mycpu_verify/testbench/mycpu_tb.v" Line 267
run: Time (s): cpu = 00:01:39 ; elapsed = 00:01:36 . Memory (MB): peak = 1215.117 ; gain = 0.781
```

从波形中也可看到，出错当拍的 `write_data` 波形现在显示了正确的值。

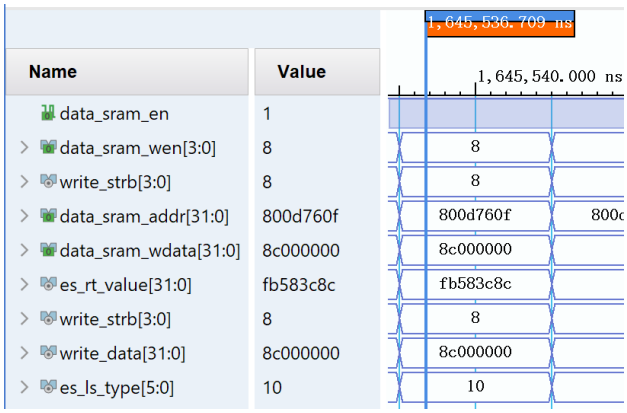

图 4

随后进行上板测试，测试运行完毕后板上显示 `0x44000044` ，全部通过。

(5) 归纳总结

在接线时，应先仔细对照接口宽度等信息再进行赋值。

四、实验总结

实验进行中，我们比较了两个人的思路和代码，进一步熟悉了框架的设计。

在动手操作中，我们逐渐了解框架的一些设计是如何做到 coding-friendly 的，同时也在自己的实验中借鉴了这些特性；这帮助我们简化了一些代码工作，并且优化了代码可读性和工程的时序。