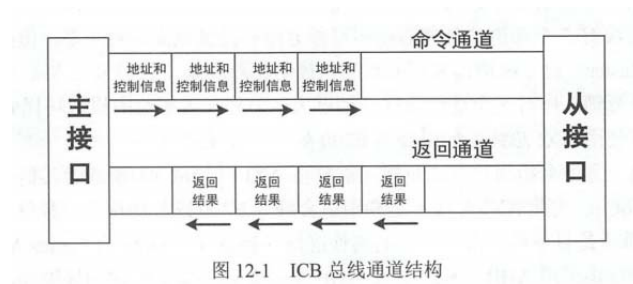


1，ICB 总线

考虑到目前主流的总线协议对于蜂鸟这种低功耗的处理器来说不太适用，比如 AXI 协议分离的读写通道虽然有很高的吞吐率但是控制相对复杂，会增大硬件开销；比如 AHB 虽然作为高性能低功耗总线的最佳选择，但是 AHB 有不容易添加流水线级数，无法支持多个滞外交易等缺点；至于 APB 作为低速设备总线，不适合作为主总线用。所以蜂鸟采用的是一款自定义的总线架构 ICB。该部分将对 ICB 做具体介绍。

1.1 ICB 总线简介

ICB 的设计初衷是为了结合 AXI 和 AHB 的优点，兼顾高速性和易用性，它具有以下基本特点：



- 1，读写操作共用地地址通道，共用结果返回通道，控制简单
- 2，分离的地址和数据阶段（AXI 类似）
- 3，支持任意主从数目（一主一从，一主多从，多主一从，多主多从，AXI 类似）
- 4，每个读写操作都会在地地址通道上产生地址（AHB 类似）
- 5，支持非对齐访问，使用 mask 来控制部分写操作（AXI 类似）
- 6，支持多个滞外交易。（AXI 类似）
- 7，结果顺序返回（AHB 类似）

对于蜂鸟来说，ICB 总线几乎被用于所有场合，包括内部模块之间的接口，SRAM 模块接口，低速设备总线以及系统存储总线等。

有关 AXI/AHB 的详细介绍，有兴趣可以自行上网查一下。

1.2 ICB 协议信号

ICB 总线包括两个通道：命令通道用于主设备向从设备发起读写请求；返回通道用于从设备向主设备返回读写结果。

表 12-1

ICB 总线信号

通 道	方向	宽度	信 号 名	介 绍
命令通道	Output	1	icb_cmd_valid	主设备向从设备发送读写请求信号
	Input	1	icb_cmd_ready	从设备向主设备返回读写接受信号
	Output	DW	icb_cmd_addr	读写地址
	Output	1	icb_cmd_read	读或是写操作的指示
	Output	DW	icb_cmd_wdata	写操作的数据
	Output	DW/8	icb_cmd_wmask	写操作的字节掩码
反馈通道	Input	1	icb_rsp_valid	从设备向主设备发送读写反馈请求信号
	Output	1	icb_rsp_ready	主设备向从设备返回读写反馈接受信号
	Input	DW	icb_rsp_rdata	读反馈的数据
	Input	1	icb_rsp_err	读或者写反馈的错误标志

注 1：上图中 DW 是数据位宽；

注 2：关于 wmask 的意思，以数据位宽 32 位为例（4 个字节，假定数据为 32'h11223344），其对应的 mask 位宽为 4bit，换句话说，每一 bit 用来指示写数据的其中一个字节是否会写入从机。我们假定 0 表示会写入（写数据不被 mask），1 表示不会写入（写数据被 mask）。如果 mask=0000，那么发送给从设备的写数据就是 11223344；如果 mask=1100，那么发送给从设备的数据就是 00003344；依次类推。

注 3：其中的 ready 和 valid 信号是基本的握手信号，简单来说，ready 表示从设备已经做好准备接受主设备的请求；valid 表示主设备已经可以发送一个请求给从设备。当这两者都为高时，表示一次握手成功。反馈通道的 rsp_valid 和 rsp_ready 也是类似的道理。

注 4，在所给代码中，有些地方会发现除了上面的信号外还有 icb_cmd_burst，beat 等信号，相关的连线已经连好，不用管它。

1.3 ICB 时序

下面介绍几种典型时序：

- 写操作同一周期返回结果

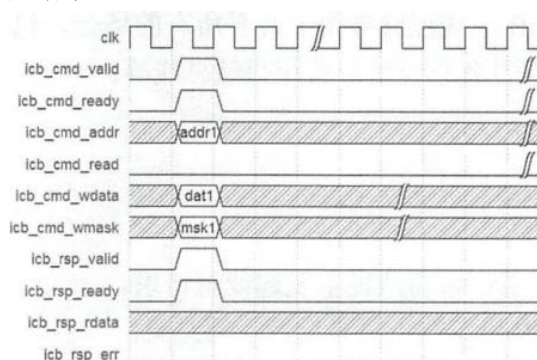


图 12-2 写操作同一周期返回结果

上面的图表示的是一次 ICB 写操作，当 cmd_valid 拉高时，当前周期主设备会发送这一次请求的具体信息，包括地址，读/写（0 表示写），写数据，写 mask。当命令通道握手成功时（cmd_valid 和 cmd_ready 同时为高）表示从设备接受到了这次请求。至于返回通道，上图是当前周期就返回反馈信息，并且成功握手（rsp_valid 和 rsp_ready 都为高），且没有 error（icb_rsp_err 为低）。

● 读操作下一周期返回结果

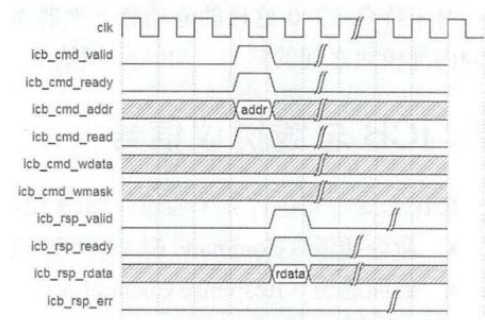


图 12-3 读操作下一周期返回结果

上面的图表示的是一次 ICB 读操作，当 **cmd_valid** 拉高时，当前周期主设备会发送这一次请求的具体信息，包括地址，读/写（1 表示读），注意对于读操作没有 **wdata** 和 **wmask**。当命令通道成功握手时，表示从设备接受到了这次请求。至于返回通道，和第一张图不同，上图是下一个周期返回结果（包括读数据和 **err** 信号）并且马上握手成功。

注意，上图只是两个特定的例子，并不是说写操作就是当前周期返回，读操作就是下一周期返回。事实上，这个取决于从设备处理读写请求的速度，如果当前周期就完成，那么当前周期返回结果，4 个周期完成那就四个周期返回结果。下面是具体的时序：

● 写操作下一周期返回结果

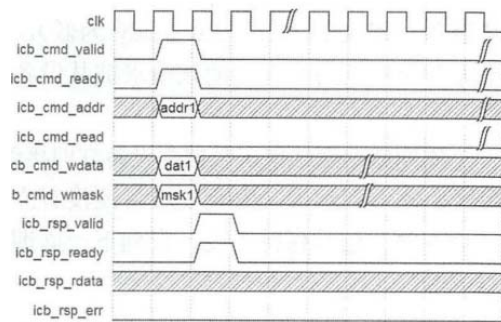


图 12-4 写操作下一周期返回结果

● 读操作 4 个周期后返回结果

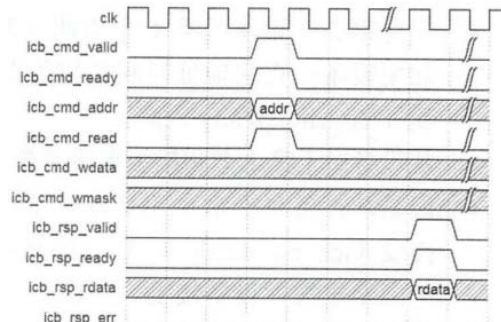


图 12-5 读操作 4 个周期返回结果

● 写操作 4 个周期返回结果且出错

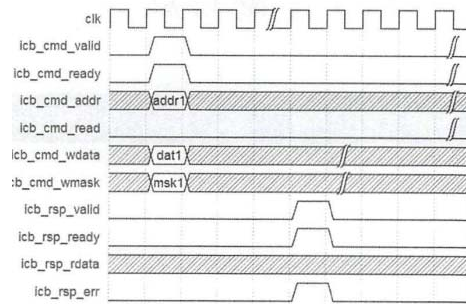


图 12-6 写操作 4 个周期返回结果

当然，也可以连续的发送读写操作请求：

- 连续 4 个读操作均为 4 周期后返回结果且最后一个错误：

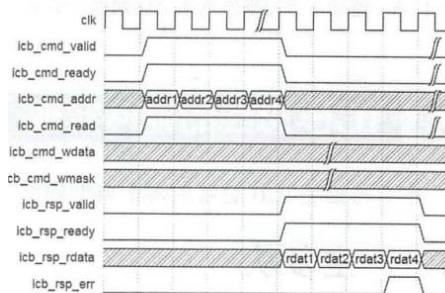


图 12-7 连续 4 个读操作均 4 个周期返回结果

- 连续 4 个写操作均为 4 个周期返回结果，最后一个 err

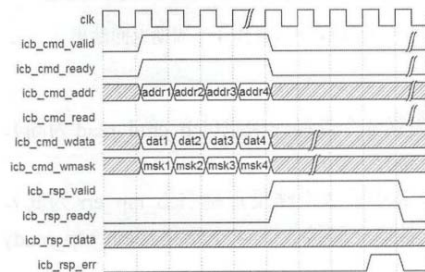


图 12-8 连续 4 个写操作均 4 个周期返回结果

- 读写混合请求：

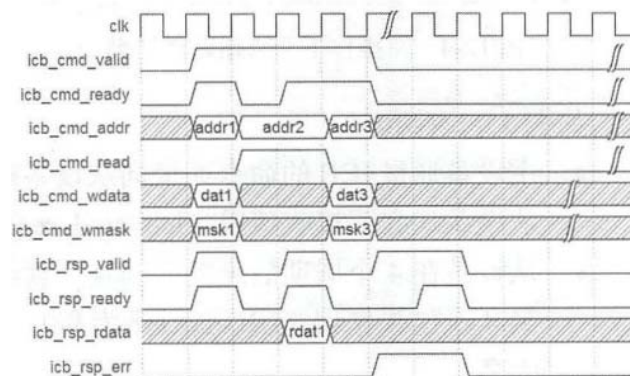


图 12-9 读写操作混合发生

在前面的时序图中 **valid** 和 **ready** 信号都是同时变化的，这表示一旦主机发送请求（**cmd_valid** 拉高），从机马上就响应。同样的，一旦从机发送反馈请求（**rsp_valid** 拉高），主机马上就响应。但并不是所有的都会是这样。比如在上面的图中，第二个请求（**addr2**,读操作）对应的 **cmd_valid** 拉高后，从机的 **ready** 信号并没有到来。表明从机还没有准备好接受请求，此时总线上的命令请求会一直保持直到 **ready** 信号来，握手成功后才意味着真正被从机所接受。类似的，对于第三个写请求（**addr3,data3,msk3**），从机反馈信号来后（**rsp_valid** 拉高），同一周期主机还没有准备好接受响应（**rsp_ready** 为低），此时会一直等到 **rsp_ready** 为高时才表示握手成功，响应被主机所接受。

上面介绍的实际上针对一个 **master** 一个 **slave** 的情况，但很多系统中往往没有这么简单。

考虑多个 **master** 和一个 **slave** 的情况，显然我们需要一个仲裁模块，通过我们自己定义的仲裁机制，在多个 **master** 同时发起请求的时候让优先级高的通过，其它等待。下面是对应的硬件实现：当命令通道的 **in1**, **in2**, **in3** 经过仲裁之后得到输出 **out** 的命令通道，并将仲裁信息压入到 **FIFO** 中，该信息会指示反馈通道应该送往 **in1,in2,in3** 的反馈通道中的哪一个。

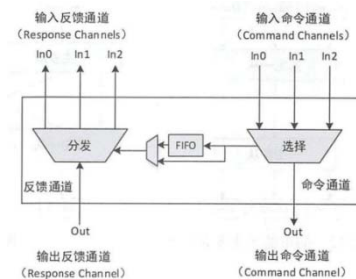


图 12-11 3 个 ICB 汇合成 1 个 ICB

考虑一个 **master** 和多个 **slave** 的情况；显然我们需要一个分发模块将 **master** 的命令请求发送给对应的 **slave**，而分发的依据为 **slave** 的地址区间，即 **master** 的命令请求地址落在哪一个 **slave** 的地址空间中，就将请求发送给哪个 **slave**。下图是其硬件实现思路：判断输入命令通道地址将命令请求发送给对应的 **slave**，并存储分发信息，该信息会指示 3 个输出反馈通道中的哪一个送给输入反馈通道。

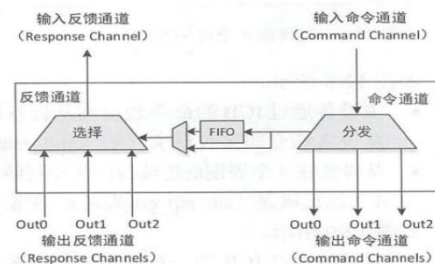
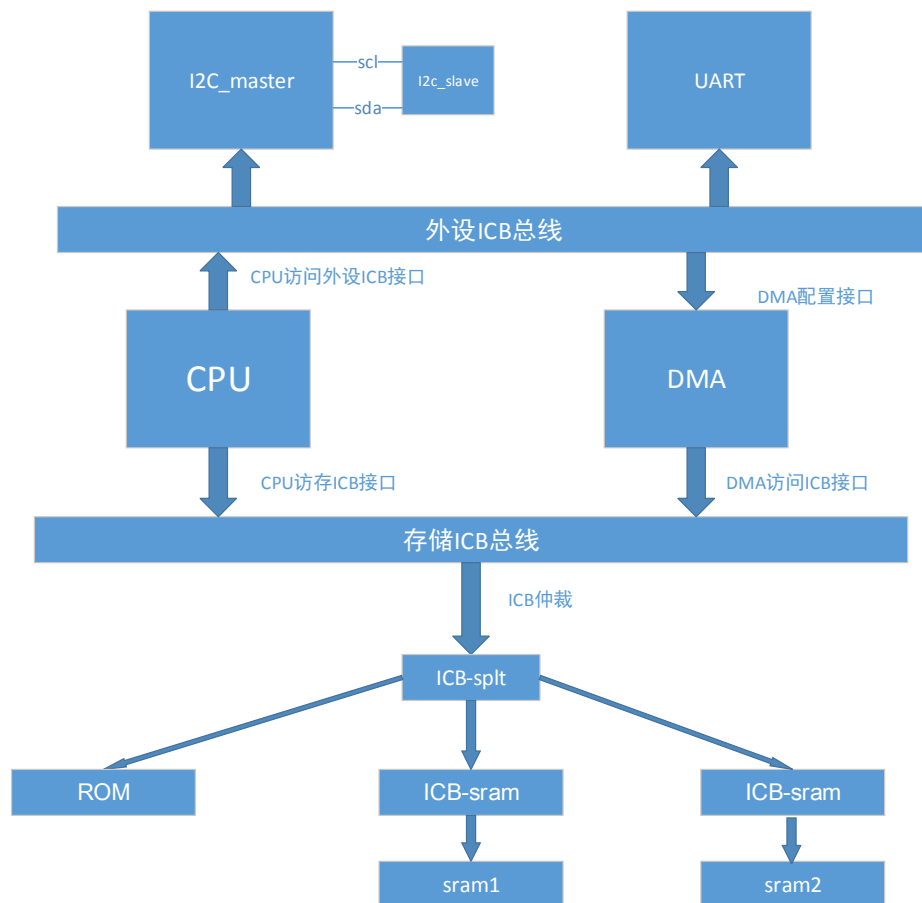


图 12-10 1 个 ICB 分分成 3 个 ICB

当多个 **master** 对应多个 **slave** 时，显然上面两者都需要。实际系统中，仲裁和分发都有专门的转接桥实现。

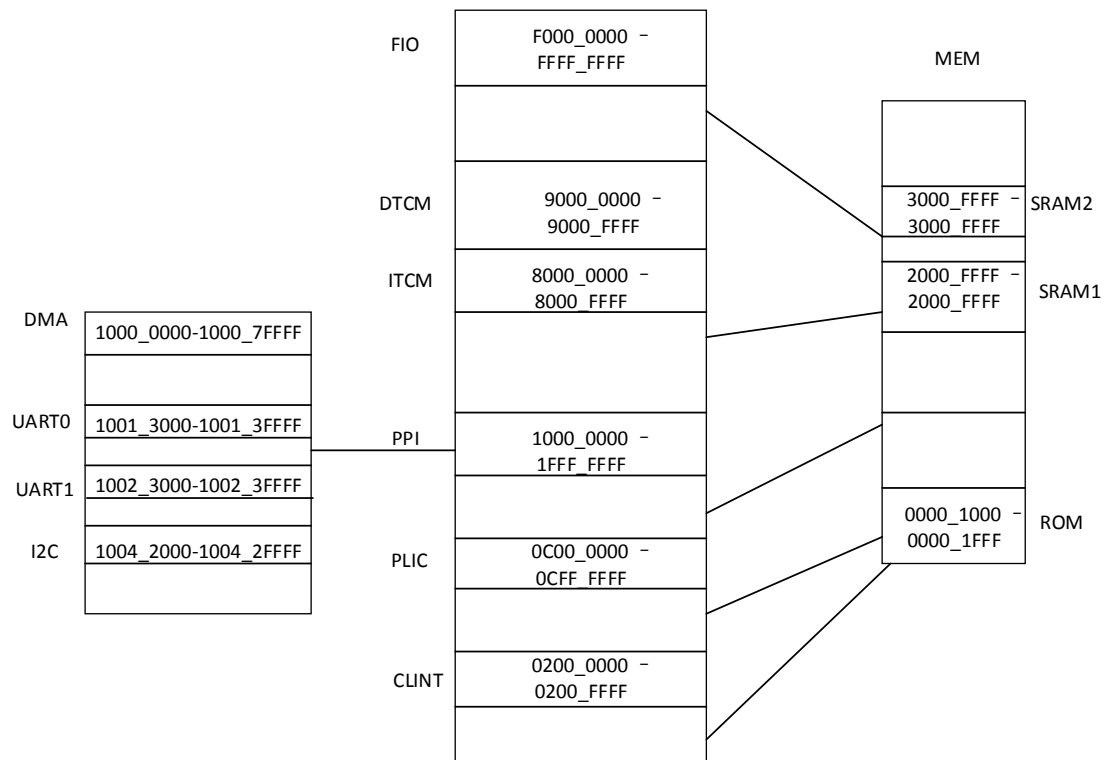
2 实验 SOC 介绍

2.1 基本框架



上图是本次实验用到的具体框架，其中 DMA 的部分实验一不关心，请暂时先忽略。CPU 有两个 ICB 接口，一个是外设 ICB 接口，一个是存储 ICB 接口。在上电时（初始化），CPU 会通过存储 ICB 接口访问 ROM，ROM 里面有引导程序，使 PC 值跳到 CPU 里面的指令寄存器；外设 ICB 方面，本实验中保留了 I2C，UART，DMA 三个外设。从 CPU 的请求会根据地址区间不同通过 ICB-splt 桥分发给对应的外设模块（这部分桥没有在上图中画出来）。其中，I2C 和 UART 的代码已经给出。

2.2 地址映射划分



这个图的意思是，当地址请求落在相应的地址区间时，会发送到相应的 ICB 接口；其中，ITCM 是 CPU 内部的指令存储器，DTCM 是 CPU 内部的数据存储器，这两部分 ICB 接口意味着可能会有外部模块发起对 ITCM,DTCM 的直接访问（当然本实验不会出现这种情况）。

这个图的中间部分，大家只需要关心两个接口，一个是 MEM 接口；MEM 接口的地址空间为中间那个图中所有空白部分对应的地址，在这些地址空间基础上进一步划分了 ROM,SRAM1,SRAM2 的地址区域（即右边那个图）。另一个接口是 PPI 接口，该接口是 CPU 的私有外设接口，在该地址空间上进一步划分出 DMA,UART0,UART1,I2C 等外设的地址区域（左边部分）；

注，该平台是删减而成，所以 MEM 空间和 PPI 空间都有一些地址空间没有用到（空白部分），所以大家不要向这些没有用到的地址区域发访问请求，否则可能会有 X 态出现。

2.3 实验文件介绍

本次实验提供了一个 demo_e200 的文件夹，该文件是在蜂鸟的 SOC 基础上删减而成，所以大家在看代码时如果有看到很多悬空的输出引脚或者一些 define 但是却没有用到的宏请不要在意。感兴趣的同学可以在 linux 终端输入下面的命令下载原本的蜂鸟项目：

```
git clone https://github.com/SI-RISCV/e200_opensource.git
```

demo_e200 中的文件如下：

```
hbird-e-sdk-master isa_test LICENSE riscv-tools rtl tb tool vsim
```

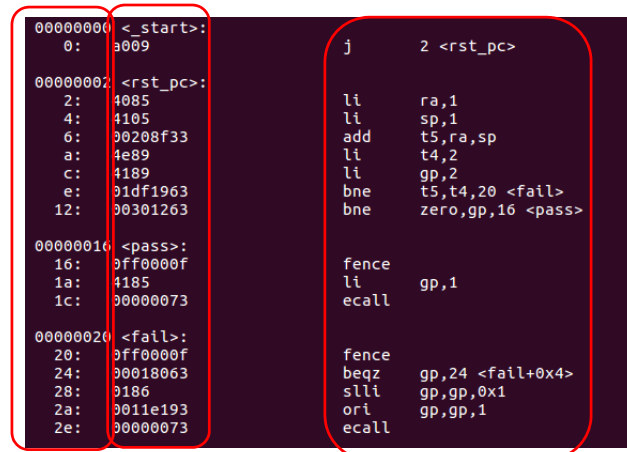
- RTL
该文件夹下是所有的代码，它包含几个子文件夹：
Core 文件夹下放的是 CPU 代码；
fab 文件夹下放的是 ICB-splt 桥的代码
general 文件夹下放的是一些公共模块，包括 sram 的仿真模型，其它 ICB 相关模块（比如 icb_buffer, icb_arbt, icb 转 apb 的桥等）等等
mems 文件夹下放的是 rom 相关的代码
perips 文件夹下放的是异步复位相关的代码以及所有外设模块代码, **本实验用到的 I2C, 和实验 2 用到的 DMA 硬件代码都放在这里面。**
Subsys 文件夹下是将 3.1 中介绍的系统框架搭起来的相关代码，其中：
1, e203_subsys_top.v 是整个 sub-system 的顶层，实现 CPU, subsys_perip, subsys_mem 的互联；
2, e203_subsys_mems.v 是存储器相关的控制，它实现 3.1 框图中存储总线 ICB 部分以及各存储器的互联
3, define.v 定义了 SRAM1, SRAM2 的起始地址
4, e203_subsys_perip.v 是外设模块的顶层，它实现 3.1 框图中外设总线 ICB 部分以及外设模块的互联
soc 文件夹中是整个 soc 平台的顶层。
上述文件中，subsys 和 soc 的代码暂时没有发现问题，如果大家在做实验的过程中发现错误请和助教联系。其余文件属于蜂鸟平台原本的代码，请大家不要动。
- Riscv-tools
该文件夹下是原本蜂鸟的测试样例，本实验中用不到，大家不要动。
- tool
该文件夹下是仿真过程中所需要的工具链，大家不要动。
- tb
该文件下提供测试平台，详细部分在后文介绍。
- vsim
该文件夹下为仿真目录，详细部分在后文介绍。
- hbird-e-sdk-master
该文件夹是和软件仿真相关的文件, **本实验用到的 I2C 软件程序放在这里**，详细部分在后文介绍。
- isa_test
该文件夹下是一个简单的加法指令测试的汇编文件

2.4 实验步骤

2.4.1 SOC 工作原理

- 测试程序介绍
在 isa_test 目录下准备了一个名为 add.S 的汇编文件，大家可以通过在该目录下输入 **make dump** 命令生成 add.o, add.dump, add.verilog 文件。其中 add.o 是可执行文件，

add.dump 是反汇编文件，可以用 vim 打开查看具体的指令及其编码，如下图所示（从左到右三个框中分别为 PC，指令，对应的汇编代码）：



（上图的汇编指令的大概意思是分别把立即数 1 存放在 ra,sp 寄存器中，并调用 add 指令将两者相加，结果存放在 t5 寄存器中，同时把理论的结果 2 存放在 t5 寄存器中；再将 t4 和 t5 比较，如果不相等则跳转到 fail，fail 程序中将 gp（x3 寄存器）置 0。如果 gp 不等于 0 则跳转到 pass，pass 中将 gp 置 1）

● 验证平台介绍

在 tb 目录下准备了一个简单的测试代码，总体来说并不复杂，下面只介绍处理器相关的部分：

```

`define CPU_TOP u_e203_soc_top.u_e203_subsys_top.u_e203_cpu_top
`define ITCM `CPU_TOP.u_e200_srams.u_e200_itcm_ram.u_e200_itcm_gnrl_ram.u_sirv_sim_ram
`define EXU `CPU_TOP.u_e200_cpu.u_e200_core.u_e200_exu
wire [`E200_XLEN-1:0] x3 = `EXU.u_e200_exu_regfile.rf_r[3];
wire [`E200_PC_SIZE-1:0] pc = `EXU.u_e200_exu_commit.alu_cmt_i_pc;
wire [`E200_PC_SIZE-1:0] pc_vld = `EXU.u_e200_exu_commit.alu_cmt_i_valid;
wire [`E200_XLEN-1:0] instr = `EXU.u_e200_exu_commit.alu_cmt_i_instr;

```

上面的代码是将处理器内部的一些信号线引出来，x3 对应整数寄存器 3(gp)的值，pc 和 pc_vld 可简单理解为处理器当前执行的指令 PC 以及对应的 valid 信号。

ITCM 是处理器核内部用来存放指令的存储器。

在仿真的过程中，会将前文生成的.verilog 文件通过 readmemh 函数读进来，然后存放在 itcm 中，初始化后，rom 里的上电程序执行完成后会跳转到 ITCM 的地址（8000_0000），再开始执行 ITCM 里面的指令。这样，处理器里执行的指令就是你自己编写的代码所对应的指令了。

```

reg [7:0] itcm_mem [0:(`E200_ITCM_RAM_DP*8)-1];
initial begin
    $readmemh({testcase, ".verilog"}, itcm_mem);

    for (i=0;i<(`E200_ITCM_RAM_DP);i=i+1) begin
        `ITCM.mem_r[i][00+7:00] = itcm_mem[i*8+0];
        `ITCM.mem_r[i][08+7:08] = itcm_mem[i*8+1];
        `ITCM.mem_r[i][16+7:16] = itcm_mem[i*8+2];
        `ITCM.mem_r[i][24+7:24] = itcm_mem[i*8+3];
        `ITCM.mem_r[i][32+7:32] = itcm_mem[i*8+4];
        `ITCM.mem_r[i][40+7:40] = itcm_mem[i*8+5];
        `ITCM.mem_r[i][48+7:48] = itcm_mem[i*8+6];
        `ITCM.mem_r[i][56+7:56] = itcm_mem[i*8+7];
    end
end

```

这个测试平台的思路在于等到测试程序运行完毕，查看 x3 的值，如果是 1 表示没

问题，否则有问题。显然，这种测试思路是只针对这类固定的程序。对于后面关于 DMA 的测试，大家可以根据自己的思路搭建自己的测试平台。

2.4.2 运行 demo e200

首先我们要保证测试的 case 编译完毕；在 demo_e200 目录下键入

```
cd isa test
```

然后 `isa test` 目录下键入

make dump

该命令会编译生成对应的.o, .verilog, .dump 文件;

编译完毕后，进入 vsim 目录：

```
cd ../vsim
```

然后依次键入下面两条命令：

make clean

```
make install
```

其中 `clean` 命令会将 `install` 文件夹删除, `install` 命令会将仿真的 `rtl` 代码(包括测试平台)复制到新建文件夹 `install` 下, 如果没有对 `rtl` 代码和 `tb` 进行修改, 可以不用敲这两行。

接下来，键入：

```
make run test
```

该命令会运行整个平台，并在 `vsim/run/(CASE_NAME)` 文件夹下生成对应的波形文件。默认测试的 `case` 为 `isa test` 里面的程序。

对于默认的程序，运行后应该会出现如下结果，表示成功。

```

~~~~~The final x3 Reg value: 1 ~~~~~
~~~~~TEST_PASS ~~~~~
~~~~~
~~~~~##### ## ##### ~~~~~
~~~~~# # # # # ~~~~~
~~~~~# # # # ##### ~~~~~
~~~~~##### ##### # ~~~~~
~~~~~# # # # # # ~~~~~
~~~~~# # # ##### ~~~~~
~~~~~

```

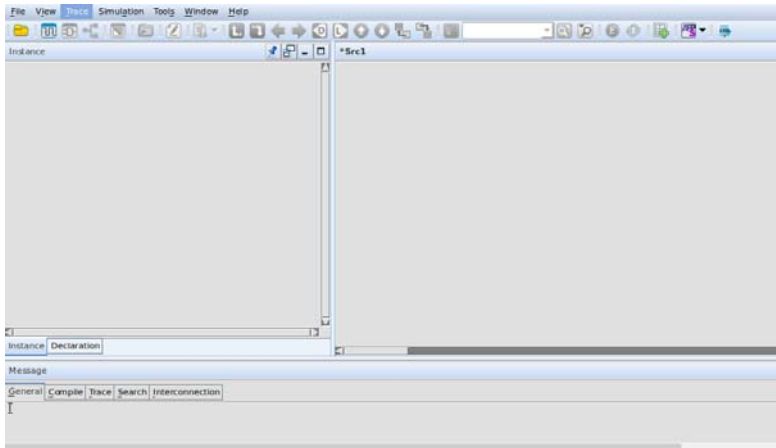
2.4.3 波形查看说明

首先在 demo e200/rtl 目录下输入:

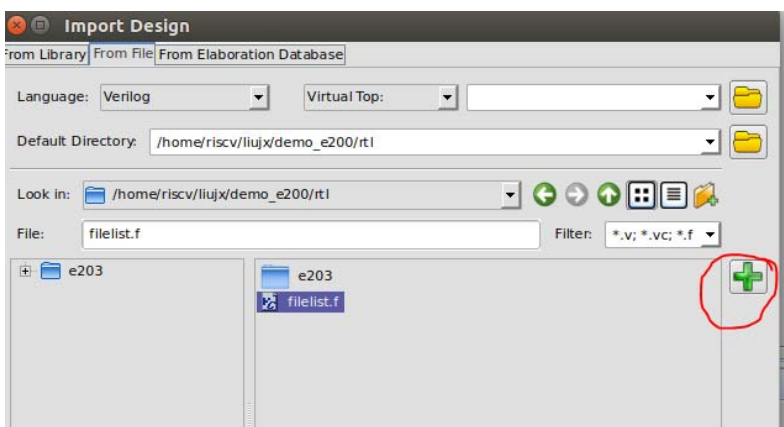
```
find -name "*.y" > filelist.f
```

该命令会生成一个 `filelist.f` 文件

在任意目录下输入 **verdi** 打开波形查看软件:



然后点击 File->import design, 在查找路径中找到对应的 filelist 文件:



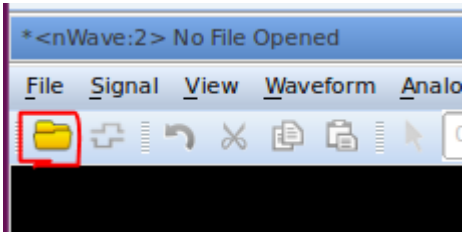
然后点击图中的加号后确定, 这样可以在 verdi 中查看代码。

注意, 导进来之后会报很多错误, 因为有些文件中包含了一些 SV 语言的语法在这里是通不过的, 大家不用管。因为这个软件不负责编译, 只是方便大家追信号。

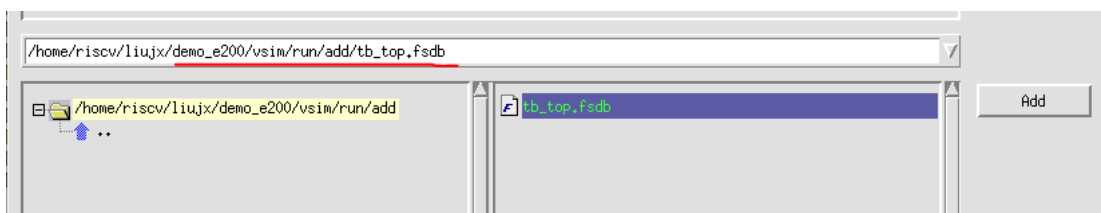
然后点击下面的图标会跳出一个空白的波形窗口, 大家可以根据自己的习惯调整窗口的大小。



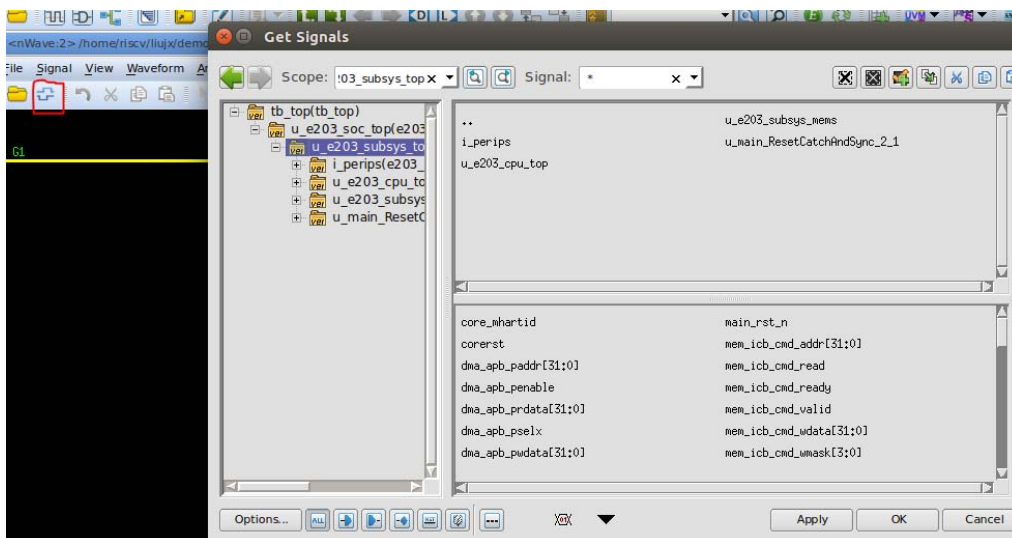
点击波形窗口中的文件夹图标:



在下面的路径下找到.fsdb 波形文件 (这里的是跑的 add 测试代码生成的波形), 点击 Add 后再点击 OK 导入波形文件。

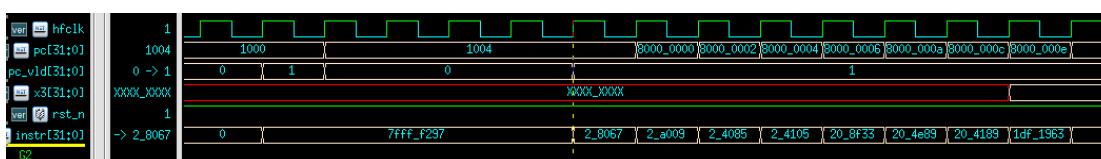


点击下图中的信号图标即选择需要的信号：

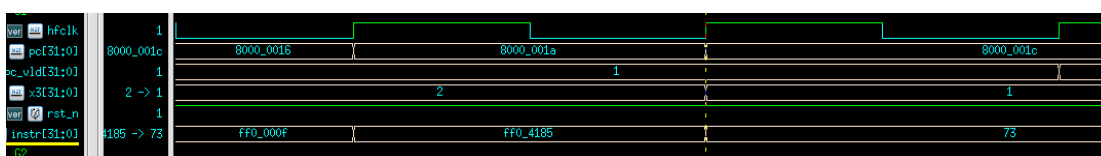


如果需要重新加载波形按 **shift+L** 或者点击 **File-->Reload** 即可。

下面这张图显示了 **add.S** 测试程序对应的波形（图中所有信号来自 **tb** 顶层）：



可以看到复位后 PC 值从 1000 开始（ROM 地址）然后执行 ROM 里面的两条指令后跳转到 8000_0000（ITCM 地址），后面的 PC 及对应的指令顺序和 **add.dump** 文件中显示的执行顺序一致。



在执行最后一条指令后（**ecall** 指令，参见 3.4.1 中的测试程序介绍），**gp (x3)** 寄存器已经设置成 1；

2.4.4 软件编写

在 **demo_e200/hbird-e-sdk-master/software/demo_i2c** 路径下有已经写好的 **demo_i2c.c** 的软件程序，

在 **demo_e200/hbird-e-sdk-master** 目录下输入

```
make dasm PROGRAM=demo_i2c
```

做交叉编译，并生成相应的 **.dump** 文件和 **.verilog** 文件。然后回到 **vsim** 目录下，先输入

vim Makefile:

```
4 #TESTNAME      := add
5 #TESTCASE      := ${RUN_DIR}/../../isa_test/${TESTNAME}
6
7 TESTNAME       := demo_i2c
8 TESTCASE       := ${RUN_DIR}/../../hbird-e-sdk-master/software/demo_i2c/demo_i2c
```

然后修改 TESTNAME 和 TESTCASE 为 demo_i2c;

然后再输入 **make run_test** 就可以开始仿真了

注：如果打开 demo_i2c.dump 文件可以发现汇编代码是非常长的，并不仅仅包含 demo_i2c.c 写的部分；事实上，在编译的过程中还包含了 bsp 文件夹里面的很多文件，有兴趣的同学可以自己看看，本实验不做要求。）

（注：在跑 i2c 程序时肯定会出现 fail 的打印信息，前面已经提到过，这个验证平台是专门为了跑 add.S 的程序验证的，跑 I2C 时注意把这段打印信息注释掉）

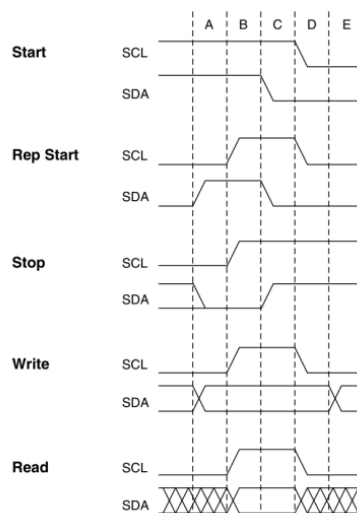
```
DEBUG i2c_slave; stop condition detected at 1000
~~~~~The final x3 Reg value: 2415922696 ~~~~~
~~~~~ TEST_FAIL ~~~~~
~~~~~##### ~~~~~
~~~~~# # # # ~~~~~
~~~~~# # # # ~~~~~
~~~~~##### ~~~~~
~~~~~# # # # ~~~~~
~~~~~# # # # ~~~~~
~~~~~# # # # ~~~~~
~~~~~# # # # ~~~~~
~~~~~##### ~~~~~
$finish called from file "/home/riscv/liujx/demo_e200/vsim/run/./install/tb/tb_top.v", line 119.
$finish at simulation time 1710000
~~~~~##### ~~~~~
```

（注：I2c 用到的宏定义在 demo_e200/hbird-e-sdk-master/bsp/hbird-e200/include/headers/devices/i2c.h 文件中，该文件在 demo_e200/hbird-e-sdk-master/bsp/hbird-e200/env/platform.h 文件中调用，同时 platform.h 文件中还定义了其它一些用到的函数，最后在 demo_i2c.c 中调用 platform.h。）

附录

I2C 说明

I2C 总线全称为 Inter-Integrated Circuit(集成电路互联总线)，是 MCU 中常用的接口模块。该协议规定了 2 条串行总线线路：一条串行数据线 SDA 和一条串行时钟线 SCL。SDA 传输数据是大端传输，每次传输 8bit 即 1 字节。每个挂载在 I2C 总线上的从设备使用一个 7bit 的地址进行唯一识别，当总线上传来的地址与自身符合时，返回一个 ACK 信号表示匹配。通过 SCL 与 SDA 电平及电平跳变关系可以表示不同的协议标志位。如图所示：



Start 信号表示传输开始，对应电平关系为：SCL 为 1 时，SDA 从 1 向 0 翻转。

Rep Start 信号表示 Master 从 Slave 读取数据的开始，对应电平关系为：SDA 为 1 时，SCL 从 0 向 1 翻转。

Stop 信号表示传输过程的结束，对应电平关系为：SCL 为 1 时，SDA 发生 0 到 1 的翻转。

Write 过程表示 Master 向 Slave 写数据的过程，对应电平关系为：SCL 为 1 时，SDA 必须保持稳定，使得数据能够成功通过总线写入 Slave。

Read 过程表示 Master 从 Slave 读取数据的过程，对应电平关系为：SCL 为 1 时，SDA 必须保持稳定，使得数据能够成功通过总线传递到 Master。

Ack 信号如下图 2 所示。Master 每发送完 8bit 数据后等待 Slave 的 ACK。即在第 9 个 clock，若从 Slave 发 ACK，SDA 会被拉低。若没有 ACK，SDA 会被置高，这会引起 Master 发生 Restart 或 Stop 流程。如果是读取数据，则第 9 个周期，Master 向 Slave 发送 Ack 或者 Nack 信号；

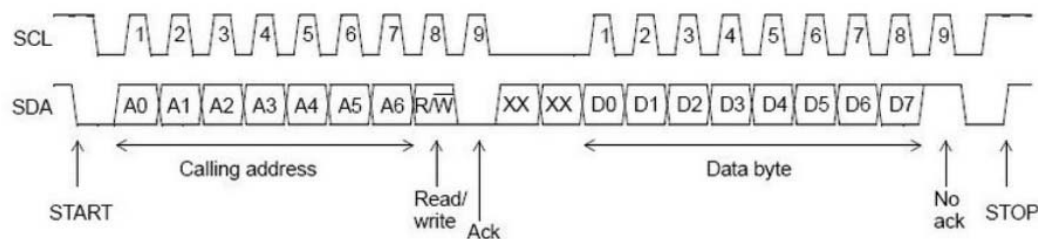


图 2. I2C Ack 信号原理图

本实验提供了写好的 i2c_master 和一个 i2c_slave，CPU 通过设置 i2c_master 里面的寄存器来控制时序。寄存器如下：

寄存器名称	偏移地址	位宽	读写属性	描述
PRERHI	0x00	8	可读可写	时钟预分频
PRERLO	0x01	8	可读可写	时钟预分频
CTR	0x02	8	可读可写	控制寄存器
TXR	0x03	8	只写	发送寄存器
RXR	0x03	8	只读	接受寄存器
CR	0x04	8	只写	命令寄存器
SR	0x04	8	只读	状态寄存器

PRERHI 和 PRERLO 是 prer 的高 8 位和低 8 位，表示分频系数，用来得到 SCL 频率。计算方式为 $\text{prer} = \text{i2c 所处时钟域时钟频率} / (5 * \text{“SCL 时钟频率”} - 1)$

由于我们给的时钟是 250MHZ，按下面的配置。

表 1. SCL 时钟频率与寄存器值关系表

SCL 时钟频率	PRERhi	PRERlo	Prescale
100KHz	0x01	0xF4	0x01F4
400KHz	0x00	0x7D	0x007D

CTR:

域名	比特域	复位默认值	描述
EN	7	0	该值为 1 表示 I2C 功能使能，软件能够配置命令寄存器来发起命令操作，否则不会响应
IEN	6	0	中断使能

TXR

域名	比特域	复位默认值	描述
TXR	7: 1	0	I2C 发送字节的高 7 位
LSB	0	0	I2C 发送字节的最低位，当是在传输从设备地址时，1 表示从从设备读数据，0 表示写数据

RXR

域名	比特域	复位默认值	描述
RXR	7: 0	0	I2C 接收到的 1 字节

CR

域名	比特域	复位默认值	描述
STA	7	0	产生开始/再开始命令
STOP	6	0	产生停止命令
RD	5	0	向从设备发起读命令
WR	4	0	向从设备发起写命令
ACK	3	0	在读数据时，接受完一字节数据后： 如果配置成 1，则发送应答标志 如果配置成 0，则发送非应答标志
IACK	0	0	中断应答，置 1 时清除等待的中断，

SR

域名	比特域	复位默认值	描述
RxACK	7	0	是否收到从设备的应答： 1 表示未收到，0 表示收到
BUSY	6	0	1 表示已检测到开始位，总线忙 0 表示已检测到停止位，总线闲
AL	5	0	1 表示仲裁丢失，仲裁丢失条件为： 未发停止位命令时检测到停止位 驱动 SDA 为高但实际为低

TIP	1	0	1 表示正在传输数据，0 表示传输数据完成
IF	0	0	中断标志

UART 说明

本实验中，UART 的作用在于方便大家调试软件代码。因为原本的 `printf` 函数是没有办法起作用的（虽然写的是软件 C 程序，但时刻要注意，这些 C 代码最终还是要有相应的硬件支持）。所以，将 `printf` 函数改写，使其对应到硬件的 `uart` 端口，这样调用 `printf` 函数的过程实际上通过串口将字符串发送到终端的过程。这部分不做详细解释。

中断说明

本实验为用到的两个外设 `i2c` 和 `DMA` 都设有一个中断输出信号，分别为 `i2c_irq` 和 `dma_irq`。为简化处理，将这两根信号取或之后直接接到处理器核的外部中断接口。在检测到中断信号来临时，指令 PC 会跳入到一个固定的中断异常服务程序入口（由 `mtvec` 维护）：

```
void _init()
{
    #ifndef NO_INIT
    uart_init(115200);

    //printf("Core freq at %d Hz\n", get_cpu_freq());

    write_csr(mtvec, &trap_entry);
    #endif
}
```

在 SOC 初始化的时候，会将 `mtvec` 指向函数 `trap_entry` 的地址（如上图，文件路径为 `demo_e200/hbird-e-sdk-master/bsp/hbird-e200/env/init.c`）

`trap_entry` 函数是用汇编写的（文件路径为 `demo_e200/hbird-e-sdk-master/bsp/hbird-e200/env/entry.S`），该函数会调用 `handle_trap` 函数，其它细节不做介绍。

`handle_trap` 函数同样在 `init.c` 中，里面调用了 `handle_m_ext_interrupt` 函数，该函数是一种 `weak` 属性的函数，熟悉 C++ 的可能能够理解，该函数本身是空函数，但是当检测到同名函数时会被覆盖；所以在写中断服务程序时，只需要像 `demo_i2c.c` 文件中一样另外写一个 `handle_m_ext_interrupt` 函数即可。

Linux 可能用到的基本命令：

```

cd /home 进入 '/ home' 目录'
cd .. 返回上一级目录
cd ../../ 返回上两级目录
cd 进入个人的主目录
cd - 返回上次所在的目录

ls 查看目录中的文件
ls -l 显示文件和目录的详细资料
ls -a 显示隐藏文件
ls *[0-9]* 显示包含数字的文件名和目录名

pwd 显示工作路径
tree 显示文件和目录由根目录开始的树形结构
mkdir dir1 创建一个叫做 'dir1' 的目录'
    mkdir dir1 dir2 同时创建两个目录
    mkdir -p /tmp/dir1/dir2 创建一个目录树

rm -f file1 删除一个叫做 'file1' 的文件'
    rm dir dir1 删除一个叫做 'dir1' 的目录'
    rm -rf dir1 删除一个叫做 'dir1' 的目录并同时删除其内容
    rm -rf dir1 dir2 同时删除两个目录及它们的内容
mv dir1 new_dir 重命名/移动 一个目录
cp file1 file2 复制一个文件
    cp dir/* . 复制一个目录下的所有文件到当前工作目录
    cp -a /tmp/dir1 . 复制一个目录到当前工作目录 |
    cp -a dir1 dir2 复制一个目录
    cp -r dir1 dir2 复制一个目录及子目录
find / -name file1 从 '/' 开始进入根文件系统搜索文件和目录
grep "define*" -r * 在整个文件夹中查找所有出现define的语句并显示出来
    grep "define*" -r * >a 在整个文件夹中查找所有出现define的语句并将结果放在新建文件a中

```

VIM 可能用到的基本命令

```

vim test.v #打开一个 test.v 文件，如果不存在则新建它
            #打开 test.v 后会处于一般模式，按 i 进入编辑模式，编辑完成后按 esc
退出编辑模式
#在编辑模式下，可以用鼠标选中一串字符，然后按下滚轮可以达到复制粘贴的效果
:w # 一般模式下,保存文件
:q # 一般模式下，退出文件或退出子窗口
:wq # 一般模式下，保存文件并退出
:q! # 一般模式下，不保存文件并退出
j(或者键盘上的下箭头) #一般模式下，向下移动光标
k(或者键盘上的上箭头) #一般模式下，向上移动光标
h(或者键盘上的左箭头) #一般模式下，向左移动光标
l(或者键盘上的右箭头) #一般模式下，向右移动光标
gd #一般模式下，搜索光标所在字符串并高亮（如果设置了高亮的话），相当于 ctrl_F
/your_string #一般模式下，搜索字符串 your_string 并高亮（如果设置了高亮的话），
相当于 ctrl+F
n #一般模式下，在搜索之后，跳入到搜索词出现的下一个地方
N #一般模式下，在搜索之后，跳入到搜索词出现的上一个地方
gg #一般模式下，跳到文件第一行

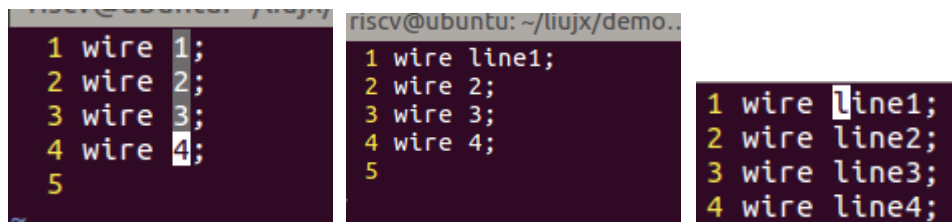
```

`gg 100` #一般模式下，调到文件第 100 行
`gf` #一般模式下，调到 `filelist` 文件中光标所在目录文件
`ctrl+6` #一般模式下，跳回 `filelist` 文件，和 `gf` 成对使用
`ctrl+f` #一般模式下，跳转到下一页
`ctrl+b` #一般模式下，跳转到上一页
`:sp [filename]` #一般模式下，切割窗口。如果无参数，切割当前窗口；如果有参数，打开新窗口
`ctrl + w +j` #一般模式下，移动到下方窗口 等同于: `ctrl + w + ↵`
`ctrl + w +k` #一般模式下，移动到上方窗口 等同于: `ctrl + w +⏮`
`u` #一般模式下，撤回，相当于 `ctrl+z`
`yy` #一般模式下，复制从光标所在行
`100yy` #一般模式下，复制从光标所在行起 100 行
`dd` #一般模式下，删除从光标所在行，并放入剪贴板，相当于 `ctrl+x`
`100dd` #一般模式下，删除从光标所在行起 100 行，并放入剪贴板
`p` #一般模式下，在光标所在行的下一行粘贴，
`P` #一般模式下，在光标所在行的上一行粘贴，
`o` #一般模式下，在光标所在行下面插入一行，同时进入编辑模式
`O` #一般模式下，在光标所在行上面插入一行，同时进入编辑模式
`:%s/aaa/bbb/g` #一般模式下，将所有 `aaa` 字符串替换为 `bbb`

另外，关于 `visual block` 的用法举例：

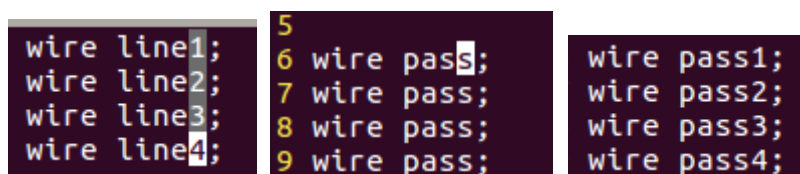
`ctrl+v` #一般模式下，进入 `visual block`

#在 `visual block` 下，通过上下左右箭头或 `jklh` 键可以实现部分区域选中（左边的图）



#然后按下大写的 `I`（一定要大写），再输入任意信息（中间的图），然后按下 `esc` 键，会变成右边图中的样子

#按照上面所说的选中 `1234` 区域，并输入 `y`，将光标移到下方，再输入 `p`，结果如下：



可以看到，`visual block` 是一种非常方便的用来处理这类含有大量重复字符文本的方式