实验二 总线实验

首先，更新代码仓库：

```
cd Arch-2022Spring

# 首先，commit 之前的代码。如果你已经 commit，就跳过下面两行
git add .
git commit -m "lab1"

git remote -v
# 如果地址是git://github.com/FDUCSLG/Arch-2022Spring-FDU.git，则执行这一步：
git remote set-url origin https://github.com/FDUCSLG/Arch-2022Spring-FDU.git

# 这一步如果卡很久，就 Ctrl-C 后重试
git fetch --all

git merge origin/master
# Merge 之后可以需要处理 Merge Conflict，在VS Code里可以很方便地处理
git submodule update
```

1 支持更多的指令

添加以下指令：

BNE BLT BGE BLTU BGEU SLTI SLTIU SLLI SRLI SRAI SLL SLT SLTU SRL SRA ADDIW SLLIW SRLIW SRAIW ADDW SUBW SLLW SRLW SRAW

一些 tips：

- logic 是无符号的。`a < b` 是将它们看作无符号数比较，`$signed(a) < $signed(b)` 是将它们看作有符号数比较。
- 逻辑右移 `a >> b[5:0]`，算术右移 `$signed(a) >>> b[5:0]`。

2 支持握手总线

详见`总线（上）`的文档。

3 支持其他粒度的内存读写

添加以下指令： LB LH LW LBU LHU LWU SB SH SW

DBus 的数据宽度是64bit（8Byte）。如果要进行Byte级别的读写，按以下规则：

3.1 字节读

dreq:

- valid = 1
- addr = addr，即无需做对齐处理
- strobe = '0
- msize: 根据字节数，设置为 MSIZE1, MSIZE2, ... 或 MSIZE8

dresp:

- data = mem[addr[63:3]]，即返回的数据是总线对齐的。

写入的数据需要做一定的处理，将读取指定的若干字节置于低位。比如，`lbu r1, 0x13(r0)`，dbus 返回的数据是 data = mem[0x10]，写入 r1 的值是 {56'b0, data[31:24]}。

```
module readdata(
    input u64 _rd,
    output u64 rd,
    input u3 addr,
    input msize_t msize,
    input u1 mem_unsigned
);
    u1 sign_bit;
    always_comb begin
        rd = 'x;
        sign_bit = 'x;
        unique case(msize)
            MSIZE1: begin // LB, LBU
                unique case(addr)
                    3'b000: begin
                        sign_bit = mem_unsigned ? 1'b0 : _rd[7];
                        rd = {{56{sign_bit}}, _rd[7-:8]};
                    end
                    3'b001: begin
                        sign_bit = mem_unsigned ? 1'b0 : _rd[15];
                        rd = {{56{sign_bit}}, _rd[15-:8]};
                    end
                endcase
                // ...
        endcase
   end
```

3.2 字节写

dreq:

- valid = 1
- addr = addr，即无需做对齐处理
- strobe: 根据地址的最后几位进行设置，需要写哪几位。
- msize: 根据字节数，设置为 MSIZE1, MSIZE2, ... 或 MSIZE8
- data: 根据地址的最后几位，进行移位。

写入的数据和 strobe 需要做一定的处理。比如，`sb r1, 0x13(r0)`，strobe 置为 `8'b00001000`，data 置为 `{32'bx, r1[7:0], 24'bx}`。

```
module writedata
    import common::*;
    import decode_pkg::*; (
    input u3 addr,
    input u64 _wd,
    input msize_t msize,
    output u64 wd,
    output strobe_t strobe
);
    always_comb begin
        strobe = '0;
```

```
        wd = '0;
        unique case(msize)
            MSIZE1: begin
                unique case(addr)
                    3'b000: begin
                        wd[7-:8] = _wd[7:0];
                        strobe = 8'h01;
                    end
                    3'b001: begin
                        wd[15-:8] = _wd[7:0];
                        strobe = 8'h02;
                    end
                endcase
                // ...
        endcase
    end
```

4 测试方法

4.1 Verilator 仿真

检测是否通过：使用 `make test-lab2 DELAY=3 TEST=all`，查看是否有 `HIT GOOD TRAP`。

4.1.1 测试项目

调试时，`TEST` 可改为以下几项（如`make test-lab2 DELAY=3 TEST=paint`）：

test	拨码开关	预期指令数	预期输出
paint	0x1	~4.4e6	输出Done! Result:，后面的内容是一张图片的base64编码，应该可以直接将其（包含开头的data:）复制到浏览器的地址栏打开。如果你的浏览器不支持直接打开base64编码的图片，请尝试在百度上找一个base64转图片的工具，如https://tool.chinaz.com/tools/imgtobase。输出的图片应该是一个你很熟悉的事物，并且没有明显的变形和噪点。图片的左上角有一行数字，表示cpu绘制这张图所画的毫秒数*。
coremark	0x2	~4.0e6	输出Finised in xxxms.表示cpu运行coremark测试程序的时间（Finised是原程序中就有的typo）。在答案正确的情况下，还会输出CoreMark Iterations/Sec，表示cpu在一秒内执行coremark循环的次数，分数越高说明cpu性能越好。作为对比，i7-7700的分数约为2e4 Iter/Sec**。
dhrystone	0x3	~4.8e6	输出Dhrystone PASS（如果是FAIL，那么说明答案错误，所给得分没有意义），并给出cpu的得分。作为对比，i7-7700k的分数约为1e5**。
stream	0x4	~8.5e6	输出对cpu访存速度的测试结果，且后面应有avg error less than 1.000000e-13 on all three arrays（表示测试过程中未出现错误）。这个测试没有官方的参考分数，我们只提供i7的copy速度供参考：L1 cache ~ 500GB/s; L2&3 cache ~ 100GB/s; Memory ~ 25GB/s。stream是线性执行的，基本上可以认为copy速度与L1 cache接近。
conwaygame	0x5	~9.9e6	输出元胞自动机（conway生命游戏）的演化结果。演化结果与随机数种子有关，没有标准答案，但在绝大多数情况下，应是看上去比较有规律的图形。
all	0x0	~3.2e7	依次执行上面五个测试。

*：表格及注释中提及的所有时间均以cpu时钟为准，仿真时会与真实时间不一致。受串口的输出速率较慢影响，上板时间多于仿真时间是正常现象。

**：i7-7700的时钟频率为3.6GHz，i7-7700k的时钟频率为4.2GHz，lab中使用的时钟频率为50MHz。可以用分数除以时钟频率来评估仅考虑流水线架构时的性能差异。

4.1.2 波形图

添加波形图的参数依然是 `VOPT="--dump-wave"`。但周期数很多，波形图会很大。你可以先跑一遍不带波形图的，然后看一下在哪个周期出错，然后用形如 `VOPT="--dump-wave -b <start> -e <end>"` 的参数来跑。

例： `make test-lab2 DELAY=3 TEST=all VOPT="--dump-wave -b 1000 -e 2000`

4.1.3 DELAY 参数

延迟周期的范围是 [2, 1 + DELAY]。在下个实验里，DELAY 将为默认值 31。

4.2 Vivado 仿真与上板

在 Vivado 中执行 `vsrc/add_sources.tcl` 添加源文件。

将实验板上右侧四个开关往下拨，连接串口软件，然后 Program Device。

在 Vivado 中执行 `run simulation` 即可开始仿真。在 `vivado/src/with_delay/simtop.sv` 中修改 `.sw(0)`，可执行单个测试。

5 作业提交

DDL：**4月24日23:59**

提交内容：以学号命名的`tar`压缩包，如`18307130024.tar`，包括 `vsrc` 和 `report.pdf`。

生成方法：

```
mkdir 18307130024
cp -r vsrc 18307130024
cp report.pdf 18307130024
tar cf 18307130024.tar 18307130024
```

你需要通过 Verilator 仿真和 Vivado 上板。

实验报告需要包括：

- 为了支持随机延迟，流水线的改动
- 测试通过的截图

实验报告里不应有大段代码的复制。

通过测试+实验报告有上述内容，本次实验就可以满分。

实验报告里可以有：

- 对本门课程实验（文档、代码风格、视频录制等）的建议
- 对后续内容的期待