Lab2 实验报告

PART1 PIPELINED RIGHT SHIFTER

完成 RightShifterPipelined 只需要修改 RightShifter.bsv 一个文件。

首先需要确定 Pipeline 的设计，本作业将流水线划分为 5 个阶段，分别负责将操作数移动 1，2，4，8，16 个比特位，每个阶段根据 shamt 相应的比特位来确定是否对操作数进行移动。由此可以确定流水线中的寄存器数量和位数，每个流水级需要记录操作数的移动结果，shamt，高位扩展信息，再加上输入寄存器，一共 6 组寄存器，每组寄存器 38bit。寄存器的创建如图 1，其中 shift_in 表示输入寄存器，shift16_status 为输出寄存器。

```
FIFO#(Bit#(38)) shift_in <- mkFIFO();
FIFO#(Bit#(38)) shift1_status <- mkFIFO();
FIFO#(Bit#(38)) shift2_status <- mkFIFO();
FIFO#(Bit#(38)) shift4_status <- mkFIFO();
FIFO#(Bit#(38)) shift8_status <- mkFIFO();
FIFO#(Bit#(38)) shift16_status <- mkFIFO();
```

图 1 寄存器定义

然后需要对 RightShifterPipelined 接口的两个方法进行实现，两个方法实际上就是输入数据时对输入寄存器进行初始化，在流水线给出计算结果后将运算结果返回，实现如图 2。

最后是对时序逻辑的实现，每个流水级中的逻辑实际上是大同小异的，都是对 shamt 中相应的比特位进行判断完成相应的操作。如果相应比特位是 1，则将从上一级取得的运算结果移动相应的位数，并正确设置高位的扩展，再将结果保存到本流水级的结果寄存器中；否则将原操作数直接保存到本流水级的结果寄存器中。图 3 是时序逻辑的实现，图中为了减少代码重复使用的是 C 语言风格的宏扩展。BSC 本身支持 C 语言风格的宏扩展，编译时加上-cpp 选项即可（但是 BSC 似乎不支持##等运算符）。

```
method Action push(ShiftMode mode, Bit#(32) operand, Bit#(5) shamt);
    Bit#(1) sign = case(mode)
        LogicalRightShift: 1'b0;
        ArithmeticRightShift: operand[31];
    endcase;
    shift_in.enq({operand, shamt, sign});
    // $display("rsp push ", mode, " ", operand, " ", shamt);
endmethod

method ActionValue#(Bit#(32)) pull();
    Bit#(32) res = shift16_status.first()[37:6];
    shift16_status.deq();
    return res;
endmethod
```

图 2 RightShifterPipelined 接口方法实现

```
typedef Bit#(32) Operand;
typedef Bit#(5) Shamt;
typedef Bit#(1) Sign;

#define STEP_RULE_DECL(fifo_in, fifo_out, shamt_offset, shift_offset) \
    Operand operand_shift = fifo_in.first()[37:6]; \
    Shamt shamt_shift = fifo_in.first()[5:1]; \
    Sign sign_shift = fifo_in.first()[0]; \
    operand_shift = multiplexer32( \
        shamt_shift[shamt_offset], operand_shift, \
        {signExtend(sign_shift), operand_shift[31:shift_offset]}); \
    fifo_out.enq({operand_shift, shamt_shift, sign_shift}); \
    fifo_in.deq();

rule shift1 (True);
    STEP_RULE_DECL(shift_in, shift1_status, 0, 1) endrule
rule shift2 (True);
    STEP_RULE_DECL(shift1_status, shift2_status, 1, 2) endrule
rule shift4 (True);
    STEP_RULE_DECL(shift2_status, shift4_status, 2, 4) endrule
rule shift8 (True);
    STEP_RULE_DECL(shift4_status, shift8_status, 3, 8) endrule
rule shift16 (True);
    STEP_RULE_DECL(shift8_status, shift16_status, 4, 16) endrule
```

图 3 时序逻辑实现

另外还需要吐槽的是，在实验过程中发现 Lab2 中 multiplexer1 的定义和 Lab1 中不一样，这就很细节了☹。

```
function Bit#(1) multiplexer1(Bit#(1) sel, Bit#(1) a, Bit#(1) b);
    // return orGate(andGate(a, sel), andGate(b, notGate(sel)));
    return orGate(andGate(a, notGate(sel)), andGate(b, sel));
endfunction
```

图 4 一个坑

至此，RightShifterPipelined 的设计全部完成。

DISCUSSION QUESTIONS

1. A pipelined version cannot use the same interface as the combinational circuit. Explain why not.

   组合电路可以认为是无状态的，输入数据便能够期望立刻得出结果；而时序电路输入数据之后，需要等待一定的时钟周期，数据完成计算后才能从中取出结果，因此输入数据和读取数据分别需要一个方法。

2. In the test file, the methods start and result of the shifter are being called from two different rules. Explain why there had to be two rules instead of one.

   RUN 规则描述了测试数据和测试结果的输入方式，TEST 规则对从时序电路中读取到的结果和正确结果进行比较。由于在流水线中，输入数据和得出结果不在同一个周期，所以如果直接写在同一个规则中，可能会导致模拟器死锁，无法正确模拟。

3. What is the throughput of your shifter (throughput is the number of full shifts per cycle)? Depending on the conditions, you may get different results. For each throughput level, explain under what conditions you are able to achieve that throughput and what you needed to do in order to get that throughput.

   当输入数据不断地进入流水线，在状态稳定之后，流水线的吞吐量能达到 1IPC。下面对功能测试和压力测试的设计进行描述。

```
RightShifterPipelined rsp <- mkRightShifterPipelined;
FIFO#(Bit#(32)) answerFifo <- mkSizedFIFO(6);
Reg#(Bit#(1)) test_stage <- mkReg(0);
Reg#(Bit#(32)) i <- mkReg(0);
Reg#(Bit#(32)) cycle <- mkReg(0);

rule run;
    Bit#(32) operand[CASE_NUM] = {
        32'hFFFFFFFF, 32'hFFFFFFFF, 32'd255, 32'd255,
        32'hFFFFFFFF, 32'hFFFFFFFF, 32'hFFFFFFF1, 32'd255, 32'd255};
    ShiftMode mode[CASE_NUM] = {
        SRL, SRL, SRL, SRL,
        SRA, SRA, SRA, SRA, SRA};
    Bit#(5) shamt[CASE_NUM] = {
        5'd15, 5'd31, 5'd3, 5'd0,
        5'd15, 5'd31, 5'd3, 5'd3, 5'd0};
    Bit#(32) ans[CASE_NUM] = {
        32'h1FFFF, 32'h1, 32'd31, 32'd255,
        32'hFFFFFFFF, 32'hFFFFFFFF, 32'hFFFFFFFE, 32'd31, 32'd255};

    // Functional test.
    if (test_stage == 1'b0) begin
        if (i == 0) $display(
            "+-----------------------------+\n",
            "| Functional test running ... |\n",
            "+-----------------------------+");
        rsp.push(mode[i], operand[i], shamt[i]);
        answerFifo.enq(ans[i]);
        i <= (i >= CASE_NUM - 1)? 32'b0 : (i + 1);
        test_stage <= (i >= CASE_NUM - 1)? 1'b1 : 1'b0;
    end
    // Throughput test.
    else begin
        if (i == 0) $display(
            "+-----------------------------+\n",
            "| Throughput test running ... |\n",
            "+-----------------------------+");
        // {i[0], i[31:1]} is to avoid the result is always 0(i >> 31).
        rsp.push(SRL, {i[0], i[31:1]}, 5'd31);
        answerFifo.enq({i[0], i[31:1]} >> 5'd31);
        i <= i + 1;
    end
    cycle <= cycle + 1;
endrule
```

图 5 RUN 规则描述

如图 5 所示所有的测试数据分为两组。第一组是功能测试，共 9 组数据，用于测试时序电路的行为是否正确；第二组是压力测试，共 32 组数据，用于测试该时序电路的吞吐量。值得注意的是在压力测试数据集的设计中，有意将正确结果设计成为 0，1 交错的形式，从而能够从输出中更加明显地看出指令运行的痕迹。

接着使用如图 6 所示的 TEST 规则对时序电路输出的数据和应得结果进行比较，并输出相应的信息用于正确性和吞吐量的判断。设置 46 个时钟周期之后停止模拟（CASE_NUM = 9）。

```
rule test;
    let b <- rsp.pull();
    let answer = answerFifo.first();
    answerFifo.deq();

    if (b != answer) begin
        $display("cycle:", cycle,
                 ", result is ", b, " but expected ", answer);
    end
    else begin
        $display("cycle:", cycle, ", correct! res = ", b);
    end

    if (cycle >= CASE_NUM + 32 + 5) $finish(0);
endrule
```

图 6 TEST 规则描述

最后经过编译并运行得到的输出如图 7 所示。首先，从输出结果中我们可以看到所有的结果均符合预期，因此时序电路运行正确。其次，功能测试开始之后，直到吞吐量测试开始之前总共输入了 9 组测试数据，却只输出了 5 个结果，这是因为流水线没有填满的缘故；而在后续的执行中，每个时钟周期都有正确的结果输出，可以近似认为吞吐量为 1IPC。

```
Simulation executable created: ./out
./out
+-----------------------------+
| Functional test running ... |
+-----------------------------+
cycle:         6, correct! res =     131071
cycle:         6, correct! res =          1
cycle:         7, correct! res =         31
cycle:         8, correct! res =        255
cycle:         9, correct! res = 4294967295
+-----------------------------+
| Throughput test running ... |
+-----------------------------+
cycle:        10, correct! res = 4294967295
cycle:        12, correct! res = 4294967294
cycle:        12, correct! res =         31
cycle:        13, correct! res =        255
cycle:        14, correct! res =          0
cycle:        15, correct! res =          1
cycle:        16, correct! res =          0
cycle:        18, correct! res =          1
cycle:        18, correct! res =          0
cycle:        19, correct! res =          1
cycle:        20, correct! res =          0
cycle:        21, correct! res =          1
cycle:        22, correct! res =          0
cycle:        24, correct! res =          1
cycle:        24, correct! res =          0
cycle:        25, correct! res =          1
cycle:        26, correct! res =          0
cycle:        27, correct! res =          1
cycle:        28, correct! res =          0
cycle:        30, correct! res =          1
cycle:        30, correct! res =          0
cycle:        31, correct! res =          1
cycle:        32, correct! res =          0
cycle:        33, correct! res =          1
cycle:        34, correct! res =          0
cycle:        36, correct! res =          1
cycle:        36, correct! res =          0
cycle:        37, correct! res =          1
cycle:        38, correct! res =          0
cycle:        39, correct! res =          1
cycle:        40, correct! res =          0
cycle:        42, correct! res =          1
cycle:        42, correct! res =          0
cycle:        43, correct! res =          1
cycle:        44, correct! res =          0
cycle:        45, correct! res =          1
cycle:        46, correct! res =          0
```

图 7 测试结果