# Lab 2

Question 1.1: A pipelined version cannot use the same interface as the combinational circuit. Explain why not. We’ve provided a new interface for you in RightShifterTypes.bsv.

答：因为组合电路不是时序的，流水线是时序电路。组合电路只需要当前的状态，流水线记录每一次流水操作的结果，因此接口不一样。

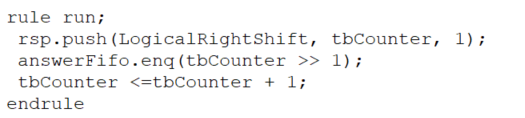
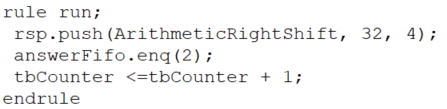
Question 1.2: In the test file, the methods start and result of the shifter are being called from two different rules. Explain why there had to be two rules instead of one. (Hint: try to call both methods from the same rule).

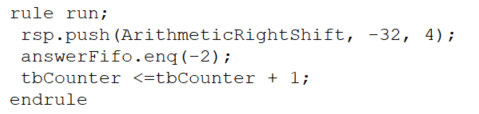
答：一个规则是用来执行部件，另一个则是检查部件，因为设计的是流水线，规则分开可以按照流水线一样并发执行指令。

Question 1.3: 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.

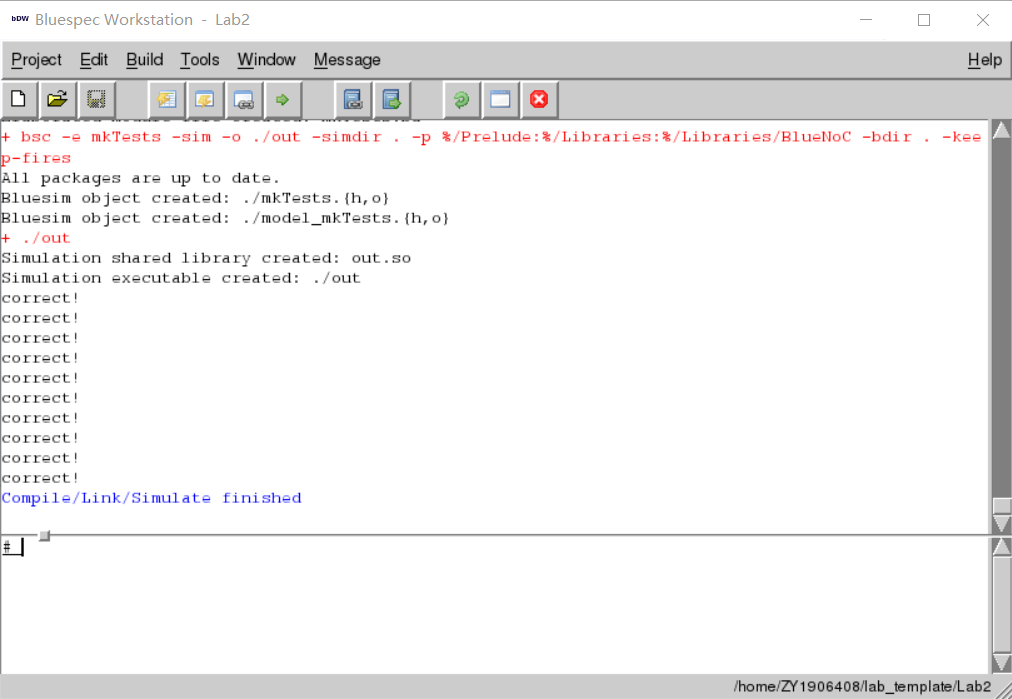
答：测试结果中需要13个周期完成测试中的10条指令，吞吐量为10/13。实现的方法为将右移操作中的右移1,2,4,8,16位分别拆分开了，比如10001代表右移16+1位，按照二进制拆开可以方便流水线的并发执行，判断对应位置是否为1就右移1,2,4,8,16位，拆分为5个部件即可获得该吞吐量。

测试内容：3种测试



测试结果均为：



## Submitting Your Solution

Provide all of the files that you changed, along with the answers to the Discussion Question in the same format as Lab 1 (but, of course, using Lab2 as the prefix to the directory.)

代码：

import RightShifterTypes::\*;

import Gates::\*;

import FIFO::\*;

// 坑一：Lab1是return (sel == 0)?a:b;这里代码变成return (sel == 1)?a:b这不是坑人？

function Bit#(1) multiplexer1(Bit#(1) sel, Bit#(1) a, Bit#(1) b);

return orGate(andGate(a, notGate(sel)),andGate(b, sel));

endfunction

function Bit#(32) multiplexer32(Bit#(1) sel, Bit#(32) a, Bit#(32) b);

Bit#(32) res\_vec = 0;

for (Integer i = 0; i < 32; i = i+1)

begin

res\_vec[i] = multiplexer1(sel, a[i], b[i]);

end

return res\_vec;

endfunction

function Bit#(n) multiplexerN(Bit#(1) sel, Bit#(n) a, Bit#(n) b);

Bit#(n) res\_vec = 0;

for (Integer i = 0; i < valueof(n); i = i+1)

begin

res\_vec[i] = multiplexer1(sel, a[i], b[i]);

end

return res\_vec;

endfunction

// 上一个Lab1编写的复制函数

function Bit#(n) copy\_frontnum(Bit#(1) sign , Integer num);

Bit#(n) result = 0;

for(Integer i = 0;i < num;i = i+1)

begin

result[i] = sign;

end

return result;

endfunction

module mkRightShifterPipelined (RightShifterPipelined);

FIFO#(Bit#(1)) high\_num <- mkFIFO();

FIFO#(Bit#(5)) shift\_num <- mkFIFO();

FIFO#(Bit#(32)) op\_num <- mkFIFO();

FIFO#(Bit#(38)) step\_for\_shift\_1 <- mkFIFO();

FIFO#(Bit#(38)) step\_for\_shift\_2 <- mkFIFO();

FIFO#(Bit#(38)) step\_for\_shift\_4 <- mkFIFO();

FIFO#(Bit#(38)) step\_for\_shift\_8 <- mkFIFO();

FIFO#(Bit#(38)) step\_for\_shift\_16 <- mkFIFO();

// 处理右移1位的规则

rule step1 (True);

Bit#(32) operand = op\_num.first();

Bit#(5) shamt = shift\_num.first();

Bit#(1) high\_bit = high\_num.first();

let result = multiplexerN(shamt[0] , operand , {high\_bit,operand[31:1]});

// 存储流水线step的结果

step\_for\_shift\_1.enq({high\_bit,shamt,result});

high\_num.deq();

op\_num.deq();

shift\_num.deq();

endrule

// 处理右移2位的规则

rule step2 (True);

Bit#(32) result = step\_for\_shift\_1.first()[31:0];

Bit#(5) shamt = step\_for\_shift\_1.first()[36:32];

Bit#(1) high\_bit = step\_for\_shift\_1.first()[37];

result = multiplexerN(shamt[1] , result , {copy\_frontnum(high\_bit,2),result[31:2]});

step\_for\_shift\_2.enq({high\_bit,shamt,result});

step\_for\_shift\_1.deq();

endrule

// 处理右移4位的规则

rule step3 (True);

Bit#(32) result = step\_for\_shift\_2.first()[31:0];

Bit#(5) shamt = step\_for\_shift\_2.first()[36:32];

Bit#(1) high\_bit = step\_for\_shift\_2.first()[37];

result = multiplexerN(shamt[2] , result , {copy\_frontnum(high\_bit,4),result[31:4]});

step\_for\_shift\_4.enq({high\_bit,shamt,result});

step\_for\_shift\_2.deq();

endrule

// 处理右移8位的规则

rule step4 (True);

Bit#(32) result = step\_for\_shift\_4.first()[31:0];

Bit#(5) shamt = step\_for\_shift\_4.first()[36:32];

Bit#(1) high\_bit = step\_for\_shift\_4.first()[37];

result = multiplexerN(shamt[3] , result , {copy\_frontnum(high\_bit,8),result[31:8]});

step\_for\_shift\_8.enq({high\_bit,shamt,result});

step\_for\_shift\_4.deq();

endrule

// 处理右移16位的规则

rule step5 (True);

Bit#(32) result = step\_for\_shift\_8.first()[31:0];

Bit#(5) shamt = step\_for\_shift\_8.first()[36:32];

Bit#(1) high\_bit = step\_for\_shift\_8.first()[37];

result = multiplexerN(shamt[4] , result , {copy\_frontnum(high\_bit,16),result[31:16]});

step\_for\_shift\_16.enq({high\_bit,shamt,result});

step\_for\_shift\_8.deq();

endrule

method Action push(ShiftMode mode, Bit#(32) operand, Bit#(5) shamt);

/\* Write your code here \*/

// Action push操作输入右移的类别、右移数字、右移位数

Bit#(1) sign\_bit = operand[31];

Bit#(1) flag = 0;

// 按照Lab1的思路进行：逻辑右移补0，算术右移补符号位

if(mode == LogicalRightShift)

begin

flag = 1;

end

sign\_bit = multiplexerN(flag , sign\_bit , 0);

// 把计算出来的数字和参数入队列

high\_num.enq(sign\_bit);

op\_num.enq(operand);

shift\_num.enq(shamt);

endmethod

method ActionValue#(Bit#(32)) pull();

/\* Write your code here \*/

// ActionValue这里返回右移的结果

Bit#(32) result = step\_for\_shift\_16.first()[31:0];

step\_for\_shift\_16.deq();

return result;

endmethod

endmodule