Handling multiple input sources

There are many situations where we would like a module in our circuit to be able to receive its input from multiple different data sources. In these situations, we use the following mux-like structure to parse the input sources into a single output.

\[
x = (v_1 \land x_1) \lor (v_2 \land x_2)
\]
\[
v = v_1 \lor v_2
\]

As long as the Bluespec compiler ensures that only 1 \( v \) value is true at any time, then this module produces the corresponding \( x \) value. We can then use the output of \( v \) to detect that output \( x \) is valid.

Registers: primitive state modules

Registers are an essential module used to store state across clock cycles. They adhere to the following interface:

```plaintext
interface Reg#(type t);
    method Action _write(t x);
    method t _read;
endinterface
```

Registers also include some special syntax in Bluespec, namely

- \( x \leq v \)  With Register \( x \) is equivalent to \( x.write(v) \)
- \( x \)  With Register \( x \) is equivalent to \( x.read \)
Method Interfaces

Every method compiled by Bluespec has can have 3 key interfaces:

- **Enable** is an input that when set to true activates the module
- **Ready** is an output that states whether the module can be activated
- **Data** can be either input or output data interfaces

Not every method has to have all of them, but they dictate when and how a method will interact externally. In Bluespec, the following statements are true in general:

- Each method has an output ready wire
- Each method may have 0 or more input data wires
- Each Action method and ActionValue method has an input enable wire
- Each value method and ActionValue method has an output data wire
- An Action method has no output data wire

*Note:* For Registers in particular, the ready wires for both _write and _read are always true!

Guards and Ready signals

In Bluespec, the ready signal for a method is constructed from all explicit and implicit (inherited from other methods called within the method) guards of that method. Consider the following interface and circuit from lecture:

```plaintext
module mkFifo (Fifo#(1, t));
  Reg#(t)  d <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(t x) if (! v);
    v <= True; d <= x;
  endmethod
  method Action deq if (v);
    v <= False;
  endmethod
  method t first if (v);
    return d;
  endmethod
endmodule
```

![Circuit Diagram](image)

It is apparent from the green ready wires that the ready signals for enq, deq, and first correspond directly with the guards, with them being !v, v, and v, respectively.
Practice Synthesis

Given the following interfaces for a function streaming module, implement the rules `invokeFunction`, which takes an element from `inQ` and passes it to module `f`’s `begin` method, and `getResult`, which gets a value from `f`’s `result` method and enqueues it into `outQ`. Then on the next page, draw a synthesized circuit for the streaming module.

```verilog
interface f#(type t);
    method Action begin(t x);
    method ActionValue#(t) result;
endinterface

interface Fifo#(type t);
    method Action enq(t x);
    method ActionValue#(t) pop;
endinterface

rule invokeFunction;
    let x = inQ.pop;
    f.begin(x);
endrule

rule getResult;
    let x <- f.result;
    outQ.enq(x);
endrule
```
Synthesized Circuit: