Problem 1

Recall the following interface and the corresponding \texttt{mkFifo (Fifo#(1, t))} module implementation of a FIFO queue given in lecture:

\begin{verbatim}
interface Fifo#(numeric type size, type t);
  method Action enq(t x);
  method Action deq;
  method t first;
endinterface

Create an module implementation using the following interface, using \texttt{Bit#(8)} as type \texttt{t}, where \texttt{pop} both dequeues and returns the follow of the first element in the Fifo:

\begin{verbatim}
interface Fifo#(numeric type size, type t);
  method Action enq(t x);
  method ActionValue pop;
endinterface

module mkPopFifo8 (Fifo#(1,Bit#(8)));
  Reg#(Bit#(8)) d <- mkRegU;
  Reg#(Bool) v <- mkReg(False);

  method Action enq(Bit#(8) x) if (!v);
    v <= True;
    d <= x;
  endmethod

  method ActionValue#(Bit#(8)) pop if (v);
    v <= False;
    return d;
  endmethod
endmodule
\end{verbatim}
**Explore:** Try changing some of the assignment operators (“< -”, “< =”, “=””) into the other ones. What compiler errors do you receive?

“< -” to “< =” Will give “assignment forbidden” if occurs in general module context (where many initializations with Registers using “< -” occur, or may complain that you are assigning a Register before instantiation.

(“< -” to “=”) or (“=” to “< -”) will give a type error, as the compiler expects to have the value instantiated but is instead being set equal to the return type of a module, or vice versa.

“< =” to (“=” or “< -”) will complain about an improper assigning of the variable. These other operators try to assign the lhs to the rhs, so if we have a Reg#(t) being “assigned” a ‘t’ the compiler may think that this variable is instead a different local variable with the same name of type t, that is being assigned without being declared.

---

**Problem 2**

Create a “streaming” rule that uses your “pop” Fifo implementation. This rule should be contained in a module that creates 3 Fifos, an “input” Fifo, a “storage” Fifo which takes its enqueues from items dequeued from the input Fifo, and an “output” Fifo, which takes its enqueues from items dequeued from the storage Fifo. *(hint: You will need 2 rules to accomplish this)*

```vhdl
module streamFifo();
    Fifo#(1, Bit#(8)) inQ <- mkPopFifo8;
    Fifo#(1, Bit#(8)) midQ <- mkPopFifo8;
    Fifo#(1, Bit#(8)) outQ <- mkFifo8;

    rule toMid;
        let v <- inQ.pop;
        midQ.enq(v);
        endrule

    rule toOut;
        let v <- midQ.pop;
        outQ.enq(v);
        endrule
endmodule
```
**Problem 3**

Create a version of the FIFO queue that can hold 2 elements. Note: When this FIFO only has a single element in it, this element should be at the “front” of the queue, or in other words it should be the element about to be dequeued/the “first” element.

```verilog
module mkPop2ItemFifo8 (PopFifo#(2, Bit#(8)));
    Reg#(Bit#(8)) a <- mkRegU;
    Reg#(Bit#(8)) b <- mkRegU;
    Reg#(Bool) full <- mkReg(False);
    Reg#(Bool) empty <- mkReg(True);

    method Action enq(Bit#(8) x) if (!full);
        Bool newFull = False;
        if (empty) begin
            a <= x;
        end
        else begin
            b <= x;
            newFull = True;
        end
        full <= newFull;
        empty <= False;
    endmethod

    method ActionValue#(Bit#(8)) pop if (!empty);
        Bool newEmpty = False;
        let retVal = a;
        if (full) begin // when removing a, move b to a
            a <= b;
        end
        else begin // otherwise, becomes empty
            newEmpty = True;
        end
        full <= False;
        empty <= newEmpty
        return retVal;
    endmethod
endmodule
```
Expressing Loops in Bluespec

Consider the following while loop in Python:

```python
s = s0
while (s > 5):
    s = s - 1
return s
```

The number of iterations that this loop will perform is dependent on the initial value s0. As a result, the loop cannot be described by unfolding. Instead, we can use a register to hold the value of s from one iteration to the next. Once we've done that, we can use a rule to update the value of s each cycle until the computation terminates.

The following diagram and Bluespec implementation describe this process:

```bluespec
Reg#(Bit#(8)) s <- mkReg(s0);
rule iteration if (s > 5);
    s <= s – 1;
endrule
```

For the described Python code, f(s) = s – 1, and p(s) = (s > 5). ‘sel’ only picks s0 for the initial value of s, and ‘en’ is dependent on ‘notDone’, or the output of p(s).