Problem 1. ★

The following Minispec function implements a combinational circuit that adds four 32-bit numbers:

```minispec
typedef Bit#(32) Word;

function Word add4(Vector#(4, Word) x);
  return x[0] + x[1] + x[2] + x[3];
endfunction
```

(A) Draw the maximum-throughput 2-stage pipeline for this circuit.

(B) Implement this 2-stage pipeline as a Minispec module by implementing the rule below. Assume the producer and consumer give and take one input and output every cycle, so no valid bits or stall logic are needed.

```minispec
module PipelinedAdd4;
  RegU#(Vector#(2, Word)) pipeReg1;
  RegU#(Word) pipeReg2;
  input Vector#(2, Word) in;
  method Word out = pipeReg2;

  rule tick;
    // First stage
    Vector#(2, Word) v;
    v[0] = in[0] + in[1];
endrule
endmodule
```
pipeReg1 <= v;
// Second stage
pipeReg2 <= pipeReg1[0] + pipeReg1[1];
endrule
endmodule

(C) Complete the skeleton code below to implement a 2-stage pipeline with valid bits (but no stall logic).

module PipelinedAdd4;
  Reg#(Maybe#(Vector#(2, Word))) pipeReg1(Invalid);
  Reg#(Maybe#(Word)) pipeReg2(Invalid);

  input Maybe#(Vector#(4, Word)) in default = Invalid;
  method Maybe#(Word) out = pipeReg2;

rule tick;
  // First stage
  if (isValid(in)) begin
    Vector#(4, Word) x = fromMaybe(?, in);
    Vector#(2, Word) v;
    v[0] = x[0] + x[1];
    pipeReg1 <= Valid(v);
  end else pipeReg1 <= Invalid;

  // Second stage
  if (isValid(pipeReg1)) begin
    Vector#(2, Word) x = fromMaybe(?, pipeReg1);
    pipeReg2 <= Valid(x[0] + x[1]);
  end else pipeReg2 <= Invalid;
endrule
endmodule
(D) Complete the skeleton code below to implement a 2-stage pipeline with valid bits and stall logic. Your pipeline should make progress if one of the stages has an invalid value.

```verilog
module PipelinedAdd4;
    Reg#(Maybe#(Vector#(2, Word))) pipeReg1(Invalid);
    Reg#(Maybe#(Word)) pipeReg2(Invalid);

    input Maybe#(Vector#(4, Word)) in default = Invalid;
    method Maybe#(Word) out = pipeReg2;

    input Bool stallIn default = False;

    // User module will stall producer if
    // stall input is set and pipeline is full
    method Bool isFull
        = isValid(pipeReg2) && isValid(pipeReg1);

    rule tick;
        // Per-stage stall signals
        Bool stall2 = stallIn && isValid(pipeReg2);
        Bool stall1 = stall2 && isValid(pipeReg1);

        // First stage
        if (!stall1) begin
            if (isValid(in)) begin
                Vector#(4, Word) x = fromMaybe(?, in);
                Vector#(2, Word) v;
                v[0] = x[0] + x[1];
                pipeReg1 <= Valid(v);
            end else pipeReg1 <= Invalid;
        end

        // Second stage
        if (!stall2) begin
            if (isValid(pipeReg1)) begin
                Vector#(2, Word) x = fromMaybe(?, pipeReg1);
                pipeReg2 <= Valid(x[0] + x[1]);
            end else pipeReg2 <= Invalid;
        end
    endrule
endmodule
```
Problem 2.

In lecture, we have seen how to increase throughput with pipelining. But we cannot easily pipeline multi-cycle sequential circuits. To increase throughput in this case, we can instead use several multi-cycle circuits in parallel.

Consider the Factorial module from the L11 worksheet (reproduced below for completeness, although you do not need to understand its internals, only its interface):

```verilog
module Factorial;
    Reg#(Bit#(16)) x(0);
    Reg#(Bit#(16)) f(0);

    input Maybe#(Bit#(16)) in default = Invalid;

    rule factorialStep;
        if (isValid(in)) begin
            x <= fromMaybe(?, in);
            f <= 1;
        end else if (x > 1) begin
            x <= x - 1;
            f <= f * x;
        end
    endrule

    method Maybe#(Bit#(16)) result =
        (x <= 1)? Valid(f) : Invalid;
endmodule
```

We want to implement a module MultiFactorial that uses two copies of the Factorial module to improve throughput. MultiFactorial has a similar interface to Factorial: it has a Maybe input in that, when set to Valid, starts a new factorial computation, and a Maybe output result, which is Valid when there is a new factorial result.

However, MultiFactorial can perform two computations in parallel: the module user can give up to two Valid inputs (over two different cycles), and the module will return their outputs through the result method, in the same order that the inputs were given.

Under the covers, MultiFactorial should implement this behavior by alternating factorial computations between its two Factorial submodules, f0 and f1. Because both modules may finish on the same cycle (and the consumer may not be able to use the result immediately) MultiFactorial includes a dequeue input that signals that the consumer has read a valid result. That is, when dequeue is True, MultiFactorial should switch its output to the output of the other factorial submodule.
(A) Complete the skeleton code below to implement MultiFactorial. You don’t need to handle the case where the module user gives a Valid input while both Factorial submodules are already busy. In other words, you can assume that the enclosing module takes care of never having more than two in-flight factorial computations.

```verilog
module MultiFactorial;
    Factorial f0;
    Factorial f1;
    Reg#(Bit#(1)) head(0); // use output of this module
    Reg#(Bit#(1)) tail(0); // pass next input to this module

    input Maybe#(Bit#(16)) in default = Invalid;
    input Bool dequeue default = False;

    rule tick;
        if (isValid(in)) begin
            if (tail == 0) f0.in = in;
            else f1.in = in;
            tail <= ~tail;
        end
        if (dequeue) head <= ~head;
    endrule

    method Maybe#(Bit#(16)) result =
        (head == 0)? f0.result : f1.result;

endmodule
```

(B) Manually synthesize the MultiFactorial module. Use the Factorial submodules as black boxes (i.e., connect their inputs and outputs but do not draw their internals).
Problem 3.

In lecture, we saw the implementation of a 2-element FIFO (first-in, first-out) queue. Complete the skeleton code below to implement an n-element FIFO, using the same structure as the 2-element FIFO we have seen.

```verilog
module FIFO#(Integer n, type T);
  Vector#(n, Reg#(Maybe#(T))) elems(Invalid);

  method Maybe#(T) first = elems[0];
  method Bool isFull;
    Bool res = True;
    for (Integer i = 0; i < n; i = i + 1)
      res = res && isValid(elems[i]);
    return res;
  endmethod

  input Bool dequeue default = False;
  input Maybe#(T) enqueue default = Invalid;

  rule tick;
    Bool needsEnqueue = isValid(enqueue);
    for (Integer i = 0; i < n; i = i + 1) begin
      // First, find next value of elems[i] given dequeue,
      // but not accounting for enqueue
      Maybe#(T) nextValue = (!dequeue)? elems[i] :
        (i == n-1)? Invalid : elems[i+1];
      // Enqueue to the first register that would be Invalid
      if (needsEnqueue && !isValid(nextValue)) begin
        nextValue = enqueue;
        needsEnqueue = False;
      end
      elems[i] <= nextValue;
    end
    if (needsEnqueue)
      $display("Warning: Attempted enqueue to a full queue, enqueued value ignored");
  endrule
endmodule
```