Problem 1. ★
Implement the combination lock FSM from Lecture 10 as a Minispec module. The lock FSM should unlock only when the last four input bits have been 0110. The diagram below shows the FSM’s state-transition diagram.

(A) Implement this state-transition diagram by filling in the code skeleton below. Use the State enum to ensure state values can only be S0-S5.

```verbatim
typedef enum { S0, S1, S2, S3, S4 } State;

module Lock;
    Reg#(State) state(S0);
    input Bit#(1) in;

    rule tick;
        state <= case (state)
            S0: ______________________;
            S1: ______________________;
            S2: ______________________;
            S3: ______________________;
            S4: ______________________;
        endcase;
    endrule

    method Bool unlock = ______________________;
endmodule
```

Note: A subset of problems are marked with a red star (★). We especially encourage you to try these out before recitation.
(B) How many flip-flops does this lock FSM require to encode all possible states?

(C) Consider an alternative implementation of the Lock module that stores the last four input bits. Fill in the skeleton code below to complete this implementation.

module Lock;
    Reg#(Bit#(4)) lastFourBits(4'b1111);
    input Bit#(1) in;
    rule tick;
        lastFourBits <= _________________________________;
    endrule
    method Bool unlock = _________________________________;
endmodule
Problem 2. ★  

Below is an implementation of a 4-bit lock, Lock4, that matches against an arbitrary pattern, given as a module argument. Use Lock4 to implement Lock8, a lock module that unlocks with an 8-bit combination.

```verilog
module Lock4(Bit#(4) combo);
    // Storing the most significant bit, inverted will ensure
    // that we will not unlock in less than four cycles
    Reg#(Bit#(4)) lastFourBits(signExtend(~combo[3]));

    input Bit#(1) in;
    rule tick;
        lastFourBits <= {lastFourBits[2:0], in};
    endrule
    method Bool unlock = (lastFourBits == combo);
endmodule

module Lock8(Bit#(8) combo);

    Lock4 upper(__________);
    Lock4 lower(__________);
    __________________________________________; // Hint: You need some extra state
    __________________________________________; // to make both locks operate in sync
    input Bit#(1) in;
    rule tick;
        __________________________________________;
        __________________________________________;
        __________________________________________;
        __________________________________________;
    endrule
    method Bool unlock = __________________________;
endmodule
```

6.004 Spring 2021 Worksheet - 3 of 13 - L11 – Sequential Circuits in Minispec
Problem 3.

(A) Composing two 4-bit Lock modules to make an 8-bit Lock module is kludgy. Instead, we can make a parametric module, Lock\#(n), that unlocks on an n-bit combination sequence (given as a module argument). Implement Lock\#(n) by filling out the code skeleton below.

```minispec
module Lock#(Integer n)(_______________________);
    Reg#(__)lastBits(_______________________);
    input Bit#(1) in;
    rule tick;
        ____________________________________________;
    endrule
    method Bool unlock = ________________________________;
endmodule
```

(B) Test your Lock\#(n) module by completing the testbench module below, called LockTest. Ideally, your testbench should test all possible 8-bit input sequences; at a minimum, it should check a few incorrect sequences as well as the correct sequence. Your testbench should print PASS if all tests are correct, and FAIL otherwise. You can add additional registers or submodules, though they aren’t needed.

```minispec
module LockTest;
    Bit#(8) combo = 8'b01100111;
    Lock#(8) lock(combo);
    Reg#(Bit#(16)) cycle(0);
    rule test;
        cycle <= cycle + 1;
    endrule
endmodule
```
Problem 4. ★

Implement the Fibonacci FSM from Problem 3 of the previous worksheet by filling in the code skeleton below.

```
// Use 32-bit values
typedef Bit#(32) Word;

module Fibonacci;
    Reg#(Word) x(0);
    Reg#(Word) y(0);
    Reg#(Word) i(0);

    input Maybe#(Word) in default = Invalid;

    rule tick;

        endrule
    endmodule
```

```
Problem 5.

Implement a sequential circuit to compute the factorial of a 16-bit number.

(A) Design the circuit as a sequential Minispec module by filling in the skeleton code below. The circuit should start a new factorial computation when a Valid input is given. Register $x$ should be initialized to the input argument, and register $f$ should eventually hold the output. When the computation is finished, the result method should return a Valid result; while the computation is ongoing, result should return Invalid.

You can use the multiplication operator (*). * performs unsigned multiplication of Bit#(n) inputs. Assume inputs and results are unsigned. Though we have not yet seen how to multiply two numbers, lab 5 includes the design of a multiplier from scratch.

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

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

    rule factorialStep;
        result = __________________________;
    endrule

    method Maybe#(Bit#(16)) result =
        ________________________________;
endmodule
```
(B) Manually synthesize your Factorial module into a sequential circuit with registers and combinational logic blocks (similar to how Lecture 11 does this with GCD). No need to draw the implementation of all basic signals (e.g., you can give formulas, like for the sel signal in Lecture 11).
Problem 6. Sequential Circuits in Minispec (Fall 2019 Quiz 2, Problem 3, 18 points)

You join a startup building hardware to mine Dogecoins. In this cryptocurrency, mining coins requires repeatedly evaluating a function with two arguments, \( sc(x, y) \). \( x \) is given to you, and mining requires trying different values of \( y \) until you find a \( y \) for which \( sc(x, y) \) is below a threshold value. Finding such a \( y \) value yields several Dogecoins as a reward, which you can then exchange for cold hard cash.

Because the \( sc \) function is expensive, it is implemented as a multi-cycle sequential module, called SC. SC is given to you. Its implementation is irrelevant, and its interface, shown below, is the usual interface for multi-cycle modules: SC has a single input, in, and a single method, getResult(). To start a new computation, the module user sets in to a Valid Args struct containing arguments \( x \) and \( y \). Some cycles later, SC produces the result as a Valid output of its getResult() method. While SC is processing an input, the getResult() method returns Invalid and in should stay Invalid.

```minispec
module SC;
    input Maybe#(Args) in default = Invalid;
    method Maybe#(Bit#(32)) getResult();
        // unknown implementation
    endmethod
    // unknown rules
endmodule
```

You are asked to design the ArgFeeder module, which accepts an input \( x \), and feeds a sequence of inputs \( (x, 0), (x, 1), (x, 2), \ldots, (x, y-1), (x, y) \) to the SC module. ArgFeeder keeps feeding values to SC until SC's result is less than threshold (a parameter to your module). At that point, ArgFeeder should return the \( y \) such that \( (x, y) \) meets this condition through its getResult() method. The diagram below sketches the implementation of ArgFeeder. Like SC, ArgFeeder follows the usual interface for a multi-cycle module.

```
Implement the ArgFeeder module by completing the implementation of the getResult() method and the tick rule. The rule considers three cases:
(i) a new input is provided to ArgFeeder,
(ii) SC returns a Valid result, and it is less than the threshold value, and
(iii) SC returns a Valid result, but it is not less than the threshold value.

You may use any Minispec operator, including arithmetic (+, -, *, /). You will not need additional registers to complete this problem. Do not add additional rules, methods, or functions.

6.004 Spring 2021 Worksheet - 8 of 13 - L11 – Sequential Circuits in Minispec
module ArgFeeder#(Integer threshold);
  SC sc;

  Reg#(Maybe#(Bit#(32))) out(Invalid);
  RegU#(Bit#(32)) x;
  RegU#(Bit#(32)) y;

  input Maybe#(Bit#(32)) in_x default = Invalid;

  method Maybe#(Bit#(32)) getResult();
    // implement the getResult() method
    return ;
  endmethod

  rule tick;
    if (isValid(in_x)) begin
      // case (i): received a new input; start a new sequence of (x, y) pairs
      sc.in = Valid(Args{x: , y: });
      out <= ;
      x <= ;
      y <= ;
    end else if (isValid(sc.getResult())) begin
      if (fromMaybe(?, sc.getResult()) < threshold) begin
        // case (ii): result satisfies threshold
        out <= ;
      end else begin
        // case (iii): result does not yet satisfy threshold
        //   send next (x, y) pair to SC
        sc.in = Valid(Args{x: , y: });
        y <= ;
      end
    end
  endrule
endmodule
Problem 7. Sequential Minispec (Spring 2020 Quiz 2, Problem 4, 16 points) ★

The incomplete Minispec module, FindLongestBitRun, below counts the length of the longest string of 1’s in a 32-bit word. The algorithm works by repeatedly performing a bitwise AND of the word with a version of itself that has been left-shifted by one. This repeats until the word is 0. The number of iterations required is the longest string of 1’s in the word. This works because each iteration converts the last 1 in any string of 1’s into a 0. The word will not equal zero until its longest string of 1’s has all been converted into 0’s.

The circuit should start a new computation when a Valid input is given and bitString is 0. The bitString register should be initialized to the input argument, and register n should hold the output. When the computation is finished, the result method should return a Valid result; while the computation is ongoing, result should return Invalid.

typedef Bit#(32) Word;

module FindLongestBitRun;
  Reg#(Bool) initialized(False);
  Reg#(Bit#(6)) n(0);
  Reg#(Word) bitString(0);

  input Maybe#(Word) in default = Invalid;

  method Maybe#(Bit#(6)) result;
    return (initialized && bitString == 0) ? _[Part A1]_ : ___[Part A1]___;
  endmethod

  rule tick;
    if (isValid(in) && bitString == 0) begin
      n <= 0;
      bitString <= _____[Part A2]_____;
      initialized <= True;
    end
    else if (initialized && (bitString != 0)) begin
      n <= n + 1;
      bitString <= _________[Part A3]________;
    end
  endrule
endmodule

(A) (8 points) There are blanks in the code above labeled [Part A#]. #. Fill in the missing code, by copying each of the lines below and filling in the blanks corresponding to parts A1, A2, and A3.

You may use any Minispec operators, built-in functions, and literals. You will not need additional registers to complete this problem. Do not add other rules, methods, or functions.

(Label: 4A_1) A1: return (initialized && bitString == 0) ? ____:
____;

(Label: 4A_2) A2: bitString <= ___________________;
(Label: 4A_3) A3: bitString <= ______________________________;

(B) (8 points) At cycle 0, the input is set to Valid(32'b0111). Copy and fill in the table below to indicate the values at the output of the result() method, the value in register n, and the value in the bitString register. Write "Invalid" if a value is invalid, "?" if a value is unknown, and just a number to indicate a valid value (i.e. you do not need to write "Valid(5)"; just write "5"). "0b" indicates that the number after it is a binary value.

(Label: 4B) Copy and fill in the table below

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0b0111</td>
<td>Invalid</td>
<td>0b1111</td>
<td>Invalid</td>
<td>0b0001</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>result() output</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value in register n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value in bitString</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 8. Finite State Machines and Sequential Circuits in Minispec (Fall 2020 Quiz 2, Problem 4, 19 points)

Suppose we want to create a system that decides if the concatenation of its previous 2 single-bit inputs is a power of 2 (where the MSB is the input from 2 cycles ago and the LSB is from 1 cycle ago). If the previous 2 bits (prior to the current input) are a power-of-two the system outputs a 1, otherwise it outputs 0. Before any input is sent, assume the initial previous 2 bits are 2'b00.

A partial FSM diagram of this circuit is shown below:

Before receiving any inputs the FSM is in state A.

(A) (7 points) For this FSM to provide the correct answer, to what existing states must D transition to (A, B, C, or D), and what output does D give (0 or 1)?

(label 4A_1) Current State = D, Input = 0, Next State = ___________

(label 4A_2) Current State = D, Input = 1, Next State = ___________

(label 4A_3) Current State = D, Output = ___________

(B) (2 points) Using the partial FSM, fill out the truth table below.

(label 4B)

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>Part_A</td>
<td>Part_A</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>Part_A</td>
<td></td>
</tr>
</tbody>
</table>

Before receiving any inputs the FSM is in state A.
(C) (10 points) We now want to implement a different version of this is-power-of-2 sequential circuit in minispec. In this version, if the previous 5 bits are a power of 2 the module’s getOutput method will return \( \log_2(\text{previous 5 bits}) \), otherwise it will return -1.

To determine is-power-of-2, we will use the following identity for integers \( X \geq 0 \) and \( N > 0 \):

\[
X = 0 \text{ or } X = 2^N \rightarrow X \& (X - 1) = 0
\]

The module has 2 registers:
- **prevBits**: contains the 5 bits to check for is-power-of-2 in the current cycle.
- **newOneIndex**: contains the index of the most recent “1” bit in **prevBits**. Hint: how does this relate to \( \log_2(\text{prevBits}) \)? (Note: this value is stored as a 4-bit 2’s complement value to support initializing it to -1).

Fill in the minispec implementation of the described module:

(label 4C)

module PowTwo;
    Reg#(Bit#(5)) prevBits(0);
    Reg#(Bit#(4)) newOneIndex(-1);

    input Bit#(1) in;

    rule tick;
        prevBits <= ________________________________________;

        if (in == 1) newOneIndex <= ________________________;

        else if (_______________) newOneIndex <=____________;

        else if (_______________) newOneIndex <=__________;
    endrule

    method Bit#(4) getOutput();
        // fill in missing code for this method
    endmethod
endmodule