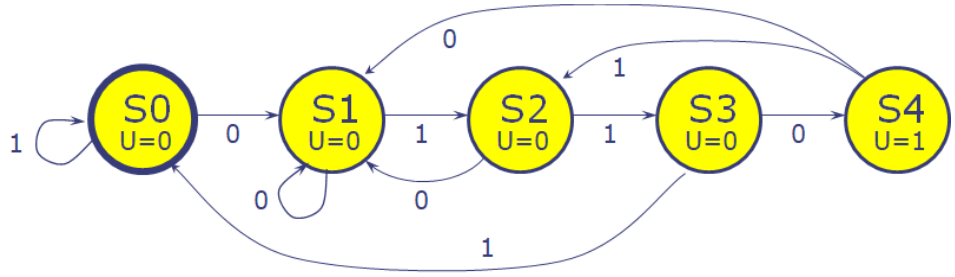
6.004 Worksheet Questions
L11 – Sequential Circuits in Minispec

Note: A subset of problems are marked with a red star (★). We especially encourage you to try these out before recitation.

Problem 1. ★

Implement the combination lock finite-state machine (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-S4.

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

module Lock;
    Reg#(State) state(S0);

    input Bit#(1) in;

    rule tick;
        state <= case (state)
            S0: (in == 0)? S1 : S0;
            S1: (in == 0)? S1 : S2;
            S2: (in == 0)? S1 : S3;
            S3: (in == 0)? S4 : S0;
            S4: (in == 0)? S1 : S2;
        endcase;
    endrule

    method Bool unlock = (state == S4);
endmodule
```

We describe our state machine transitions by using a case statement. With 1 input, we see:

Current State	Next state if in = 0	Next state if in = 1
S0	S1	S0
S1	S1	S2
S2	S1	S3
S3	S4	S0
S4	S1	S2

Our case statement converts this table into code.

(B) How many flip-flops does this lock FSM require to encode all possible states?

5 possible states → 3 bits

5 states mean we have states numbered: 0, 1, 2, 3, 4
We need 3 bits to encode 4 into binary

(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 <= {lastFourBits[2:0], in};
    endrule

    method Bool unlock = (lastFourBits == 4'b0110);
endmodule
```

We update the registers with the most recent 3 bits (lastFourBits[2:0]), and then concatenating with the input in.

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.

```
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
```

Variant 1: Run upper 4 cycles behind

```
module Lock8(Bit#(8) combo);
    Lock4 upper(combo[7:4]);
    Lock4 lower(combo[3:0]);
    Reg#(Bit#(4)) lastFourBits(signExtend(~combo[7]));

    input Bit#(1) in;

    rule tick;
        lower.in = in;
        upper.in = lastFourBits[3];
        lastFourBits <= {lastFourBits[2:0], in};
    endrule

    method Bool unlock = upper.unlock && lower.unlock;
endmodule
```

For the first variant, we want upper to be taking in bits 4 cycles behind lower. Thus, we always feed in the current input bit to lower. However, we also append the input to the register lastFourBits. This register represents the stored input to upper. Thus, if lower is receiving input in on cycle N, upper will only receive that input in when it is the most significant bit of lastFourBits. This will be 4 cycles from now. We only want to return when both are “unlocked”, i.e., the combo is satisfied.

Variant 2: Run upper and lower in lockstep, remember upper's decisions

```
module Lock8(Bit#(8) combo);
    Lock4 upper(combo[7:4]);
    Lock4 lower(combo[3:0]);
    Reg#(Bit#(4)) lastUpperUnlocks(0);

    input Bit#(1) in;

    rule tick;
        lower.in = in;
        upper.in = in;
        lastUpperUnlocks <= {lastUpperUnlocks[2:0], upper.unlock? 1 :
0};
    endrule

    method Bool unlock = (lastUpperUnlocks[3] == 1) && lower.unlock;
endmodule
```

In variant 2, we are feeding the input to both upper and lower at the same time. However, we want to keep track of whether or not upper had unlocked 4 cycles earlier. If it had and lower has unlocked this cycle, it means we can return true. We keep track of this using the register lastUpperUnlocks. Because we are shifting it to the left each iteration, if lastUpperUnlocks[i]==1, that means that upper unlocked i+1 iterations ago (it's zero indexed). Thus, lastUpperUnlocks[3]==1 means that upper unlocked 4 cycles ago.

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.

```
module Lock#(Integer n)(Bit#(n) combo);

    Reg#(Bit#(n)) lastBits(signExtend(~combo[n-1]));

    input Bit#(1) in;

    rule tick;

        lastBits <= {lastBits[n-2:0], in};

    endrule

    method Bool unlock = (lastBits == combo);
endmodule
```

We can simply keep track of the last n elements and unlock if the last n elements match the combo specified.

(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.

```
module LockTest;
    Bit#(8) combo = 8'b01100111;
    Lock#(8) lock(combo);
    Reg#(Bit#(16)) cycle(0);

    rule test;
        // Feed the lock all the possible input patterns
        // An easy way to do this is by giving it all possible
        // 8-bit sequences one after the other; this will have
        // many duplicate patterns, but covers the whole space
        Bit#(8) curPattern = cycle[10:3];
        Bit#(3) curIdx = cycle[2:0];
        lock.in = curPattern[7 - curIdx];
```

```
        // Check whether output matches what we expect.
        // We derive the last few bits from cycle,
        // but you could keep them in a register
        Bit#(16) inputWindow = {curPattern-1, curPattern};
        Bit#(4) winIdx = {0, curIdx};
        Bit#(8) lastBits = inputWindow[15-winIdx:8-winIdx];

        if (lastBits == combo && !lock.unlock) begin
            $display("FAIL: lock didn't unlock with correct
combo");
            $finish;
        end
        if (lastBits != combo && lock.unlock) begin
            $display("FAIL: unlocked with wrong combo %b",
lastBits);
            $finish;
        end

        // We pass once we've tested all the patterns
        if (cycle == 1<<12) begin
            $display("PASS after testing %d patterns", cycle-1);
            $finish;
        end

        cycle <= cycle + 1;
    endrule
endmodule


%%sim LockTest

PASS after testing  4095 patterns
```

We would like to iterate from 0 to 2**8 and check that the module behaves properly for each value of combo in this range. Thus, we must keep track of our current value of combo. This is done in cycle. In addition to our current value of combo, we must keep track of which bit of this combo we are currently feeding in, as it takes 8 cycles to feed in one combo. The index of the current value of combo is specified in the 3 least significant bits of cycle. The value of combo is specified in bits 10:3 of cycle. The register cycle is incremented by 1 every iteration, so the index

is increased by 1 until it is equal to 7. Then index is set to 0, and the value of combo is increased by 1, representing a test for a new combination.

For each cycle, we would like to check if the past 8 bits we have fed in were equal to combo or not. This can tell us whether or not we want to unlock. The current combo is represented by curPattern. The previous combo is represented by curPattern – 1. We also know that index of the current combo is curIdx, or winIdx when expressed in 4 bits. Thus, we can index {curPattern-1, curPattern} as specified above to get the last 8 bits fed in.

Problem 4 ★

In this problem, we construct a sequential circuit to compute the N^{th} Fibonacci number denoted by F_N. The following recurrence relation defines the Fibonacci sequence.

$$F_0 = 0, F_1 = 1, F_N = F_{N-1} + F_{N-2} \ \forall \quad \texttt{N >= 2}$$

There are two registers **x** and **y** that store the Fibonacci values for two consecutive integers. In addition, a counter register **i** is initialized to N-1 and decremented each cycle. The computation stops when register **i** goes down to 0 and the result (F_N) is available in register **x**.

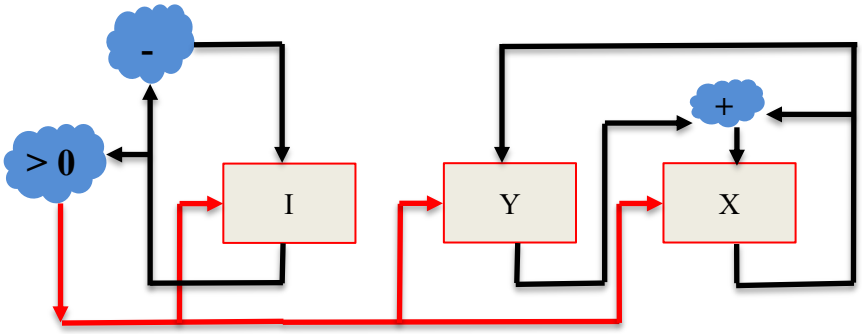
(A) What are the initial values for registers **x** and **y**?
The initial values are y = 0, x = 1 respectively.

(B) Derive the next state computation equations for the three registers.
$i^{t+1} = i^t - 1$
$y^{t+1} = x^t$
$x^{t+1} = x^t + y^t$

(C) Derive the logic for the enable signal that determines when the registers are updated using the next state logic. Note that all three registers are controlled by a single enable signal.
$i^t > 0$

This ensures that computation stops when counter i becomes 0.

(D) Implement the sequential circuit using the next state and enable logic derived above



Implement the Fibonacci FSM 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;


        if (isValid(in)) begin
            x <= 1;
            y <= 0;
            i <= fromMaybe(?, in) - 1;
        end else if (i > 0) begin
            x <= x + y;
            y <= x;
            i <= i - 1;
        end


    endrule

    method Maybe#(Word) result = (i == 0)? Valid(x) : Invalid;
endmodule
```

The next state computation equations are:

$i^{t+1} = i^t - 1$

$y^{t+1} = x^t$

$x^{t+1} = x^t + y^t$

Note that we update x, y, and i at each clock cycle. We check for a valid input in to load into i, and the result is found at x when i == 0.

Problem 5.

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

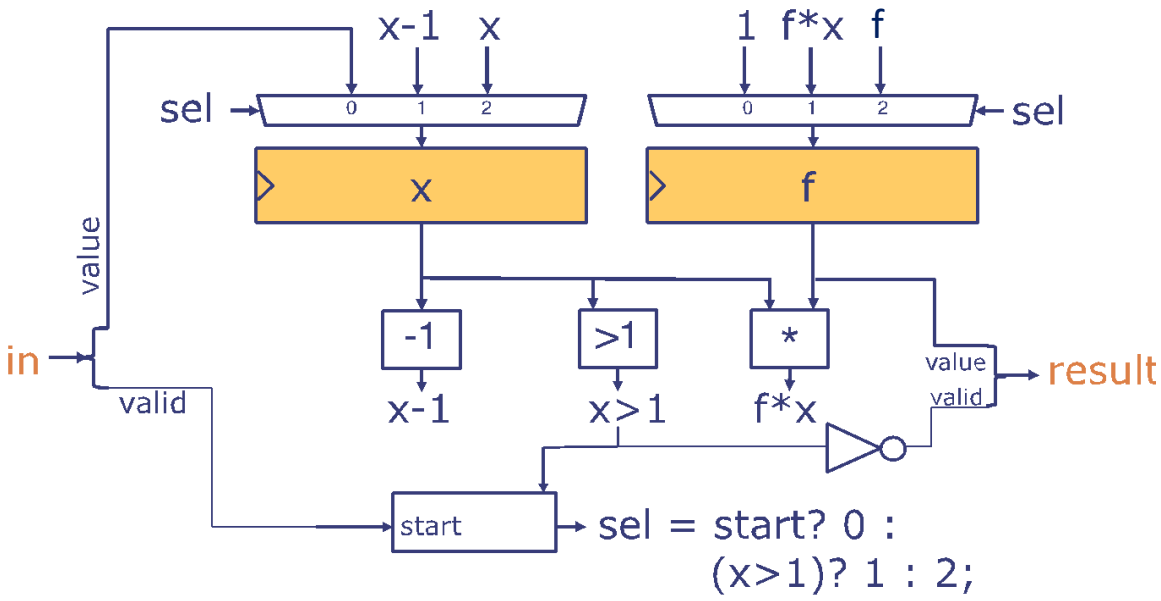
(A) Design the circuit as a sequential Minispec module by filling in the skeleton code below. The circuit should start a new factorial computation when a Valid input is given. Register **x** should be initialized to the input argument, and register **f** should eventually hold the output. When the computation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

```
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
```

Similarly to Problem 2, we initialize our values with valid input in. Then, x stores the number of iterations in the factorial computation, and f stores the product.

(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 sel in Lecture 11).



We use the structure of the GCD circuit as a base. The shaded blocks of x and f are registers (indicated by the clock notch on the side).

For x:

- We start at value in (sel = 0)
- Once valid in, each subsequent clock pulse will select x-1 to multiply.
- Once done, we hold the value of x at x

For f:

- We start at value 1 (sel = 1)
- Once loaded, each subsequent clock pulse will select f*x into f
- Once done, we hold the value of f at f

Using registers:

- We calculate x-1 through subtracting the value out of x
- We check for x <= 1 as ~(x > 1)
- We calculate f*x through multiplying f and x.

Sel:

- A start signal makes sel = 0
- If the factorial is still being computed (x > 1), sel = 1
- When we are done and need to hold value (x <= 1), sel = 2

Problem 6. From Past Quizzes (Fall 2019)

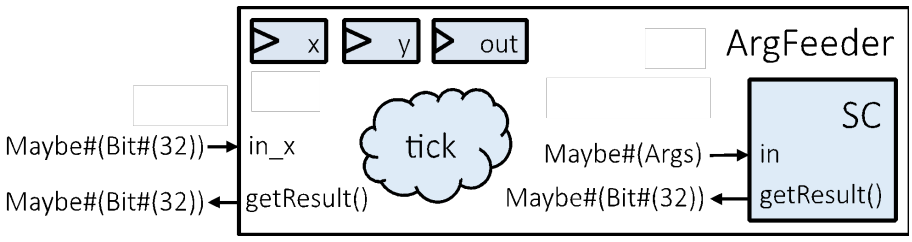
You join a startup building hardware to mine Dogecoins. In this cryptocurrency, mining coins requires repeatedly evaluating a function with two arguments, `sc(x, y)`. `x` is given to you, and mining requires trying different values of `y` until you find a `y` for which `sc(x, y)` is below a threshold 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`.

```
module SC;
    input Maybe#(Args) in default = Invalid;

    method Maybe#(Bit#(32)) getResult();
      // unknown implementation
    endmethod

    // unknown rules
endmodule
```

```
// input struct to SC
typedef struct {
    Bit#(32) x;
    Bit#(32) y;
} Args;
```

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)`, ..., `(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.

```
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 [out                                                    ];
    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: [fromMaybe(?, in_x)] , y: [0]                   });
 out <=      [Invalid                                                  ];
 x <=    [fromMaybe(?, in_x)                                           ];
 y <=    [1                                                            ];

        end else if (isValid(sc.getResult())) begin
            if (fromMaybe(?, sc.getResult()) < threshold) begin
                // case (ii): result satisfies threshold
 out <=      [Valid(y-1)                                               ];

            end else begin
                // case (iii): result does not yet satisfy target
                //   send next (x, y) pair to SC
 sc.in = Valid(Args{x: [x]          , y: [y]                   });
 y <=    [y + 1                                                        ];

            end
        end
    endrule
endmodule
```

If we receive a new input, we want to store x and initialize y. We also want to pass the new input into sc. We set y to equal 1 as y will only equal 1 on the next clock cycle, where it will go into the third branch.

If our result is valid, we want to update our output register. It is y-1 as it is the value of y in the previous cycle that was fed to sc to produce this valid result.

If our result is not valid, we simply query sc again. We increment y for the next cycle.

Problem 7. From Past Quizzes (Spring 2020) ★

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) 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.

A1: `return (initialized && bitString == 0) ? Valid(n) : Invalid;`

A2: `bitString <= ____fromMaybe(?, in)____;`

A3: `bitString <= __bitString & (bitString << 1)__;`

(B) 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.

Copy and fill in the table below

Cycle	0	1	2	3	4	5	6
Input	`0b0111`	`Invalid`	`0b1111`	`Invalid`	`0b0001`	`Invalid`	`Invalid`
`result()` output	**Invalid**	**Invalid**	**Invalid**	**Invalid**	**3**	**Invalid**	**1**
value in register `n`	**0**	**0**	**1**	**2**	**3**	**0**	**1**
value in `bitString`	**0**	**0b0111**	**0b0110**	**0b0100**	**0b0000**	**0b0001**	**0b0000**

For part A, we would like to keep track of the number of zeros seen in the register n. Therefore, when we have received an input (initialized = True) and are done processing the input (bitstring = 0), we can return the result n. When we receive a new input in, we want to store that in the register bitString. Lastly, every iteration we and bitstring with itself left shifted by 1, as the problem specifies.

For part B, all registers take a cycle to update, which is why bitString only registers a value on cycle 1. N is incremented by 1 each cycle until the value of bitString is zero, which causes the program to go into the first if statement and set n to be 0 on the next cycle.

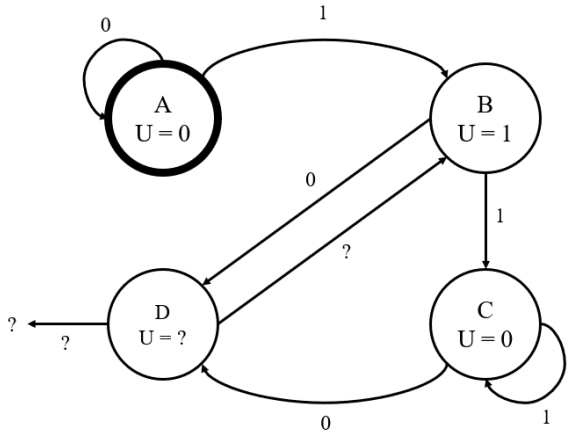
A new input can only be received when bitString is 0, which is on cycle 5. Bitstring gets updated with this input a cycle after.

The value of bitString can be found by looking at the value of bitString on the previous cycle and anding it with itself left shifted by 1.

Problem 8. From Past Quizzes (Fall 2020)

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) 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)?

Current State = D, Input = 0, Next State = ____A_____

Current State = D, Input = 1, Next State = ____B_____

Current State = D, Output = ____1_____

A represents 00, C represents 11, B represents 01, and D represents 10. Therefore, receiving an input of 0 at state D would lead to state 00, and receiving an input of 1 at state D would lead to state 01. If you are currently at D, the previous two bits are a power of 2, leading the output to be 1.

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

State	Input	Next State	Output
A	0	A	0
A	1	**B**	
B	0	D	**1**
B	1	**C**	
C	0	D	0
C	1	C	

D	0	Part_A	Part_A
D	1	Part_A	

A represents 00, C represents 11, B represents 01, and D represents 10. Therefore, receiving an input of 1 for A would lead to 01. A is not a power of 2, so the output is 0. Receiving an input of 1 at B would lead to state 11. 01 is a power of 2, so the output at state B is a 1.

(C) 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 $\log_2$(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:**

$$X = 0 \; or \; X = 2^N \rightarrow X \; \& \; (X - 1) = 0$$

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 $\log_2$(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:

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

    input Bit#(1) in;

    rule tick;

      prevBits <= __{prevBits[3:0], in}____________________;

      if (in == 1) newOneIndex <= ___0_____________________;

      else if (newOneIndex == 4) newOneIndex <=__-1_______;

      else if (newOneIndex != -1)
                              newOneIndex <= newOneIndex + 1;

    endrule
```

```
    method Bit#(4) getOutput();
      // fill in missing code for this method
      if ((prevBits & (prevBits - 1)) == 0) return
newOneIndex;
      else return -1;



   endmethod
endmodule
```

We update prevBits every cycle as we receive a new bit as input every cycle. Thus, the last 5 bits change. newOneIndex represents the index of the most recent one. Thus, when we see that in is equal to 1, we are receiving a new 1 that will become the last bit of our prevBits. Thus, we initialize newOneIndex to be 0. Every cycle, we add 1 to newOneIndex. This is because the 1 gets shifted to the left every time we add a new bit. When newOneIndex is equal to 4, we know that in the next iteration, there will no longer be a 1 in prevBits. Thus, we set newOneIndex equal to -1.

In getOutput(), we only want to return newOneIndex if the condition above is satisfied. Else, we return -1.