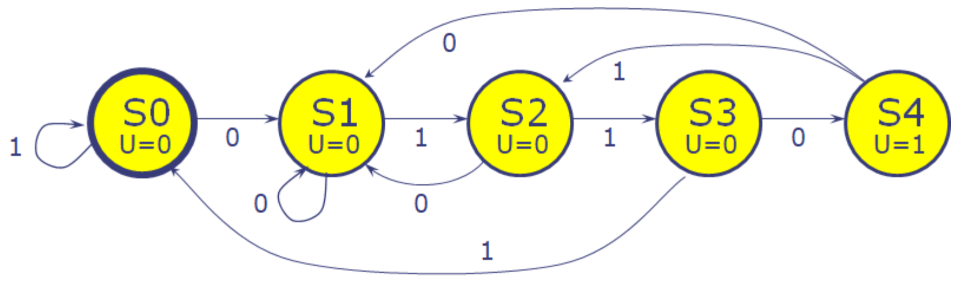
# 6.004 Tutorial Problems
# L11 – Sequential Circuits in Minispec

**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 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.

```
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.** ★

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;

        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 (from the previous worksheet) 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 3.**

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

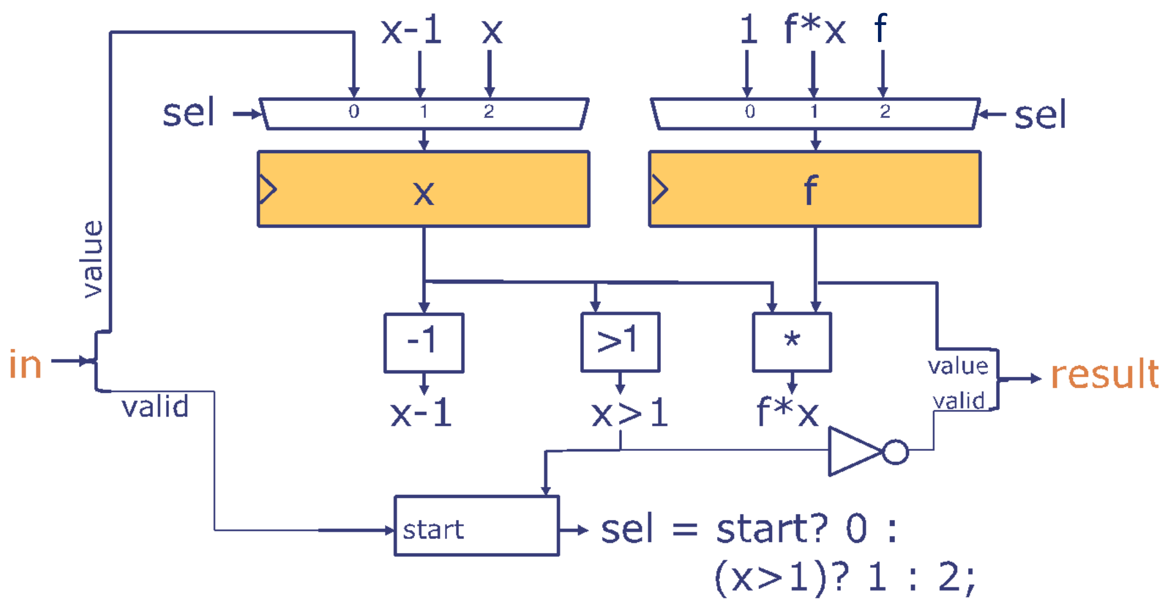
(A) Design the circuit as a sequential Minispec module by filling in the skeleton code below. The circuit should start a new factorial computation when a Valid input is given. Register **x** should be initialized to the input argument, and register **f** should eventually hold the output. When the computation 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 4. Sequential Circuits in Minispec (Fall 2019 Quiz 2, Problem 3, 18 points)**

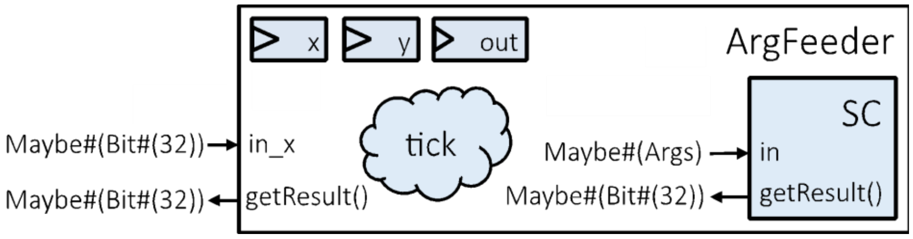
You join a startup building hardware to mine Dogecoins. In this cryptocurrency, mining coins requires repeatedly evaluating a function with two arguments, `sc(x, y)`. `x` is given to you, and mining requires trying different values of `y` until you find a `y` for which `sc(x, y)` is below a threshold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

```
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
```

**Problem 5. 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) ? Valid(n) : Invalid;`

**(Label: 4A_2) A2:** `bitString <= ____fromMaybe(?, in)____;`

**(Label: 4A_3) A3:** `bitString <= __bitString & (bitString << 1)__;`

(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**

| **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** |