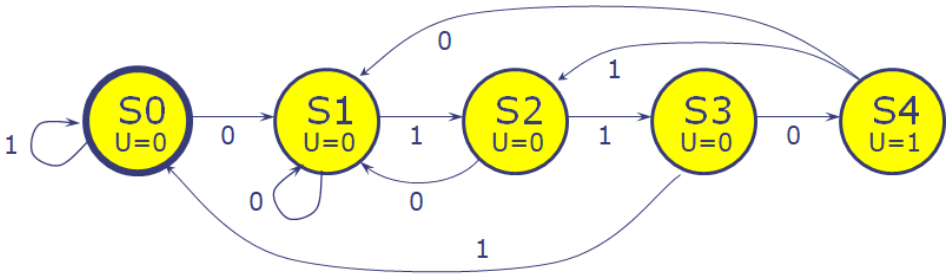
6.004 Tutorial Problems
L11 – Sequential Circuits in Minispec

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.

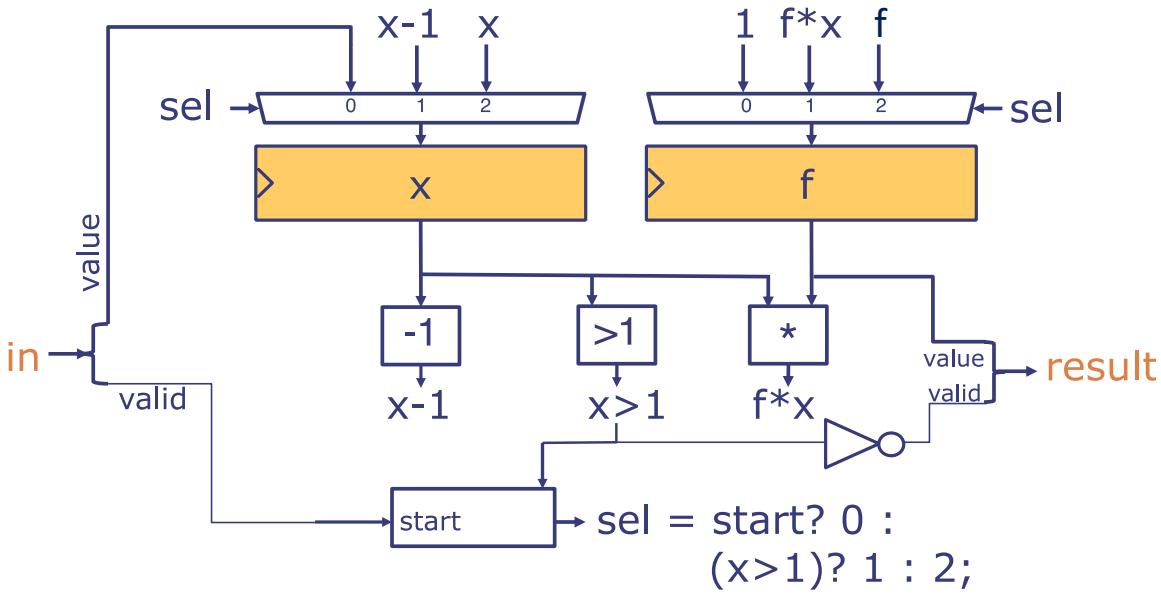
(A) Design the circuit as a sequential Minispec module by filling in the skeleton code below. The circuit should start a new factorial computation when a Valid input is given. Register **x** should be initialized to the input argument, and register **f** should eventually hold the output. When the computation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