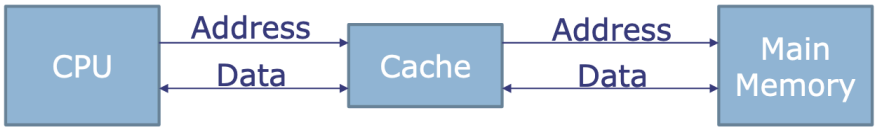
6.004 Recitation Problems
L14 – Memory Hierarchy

Keep the most often-used data in a small, fast SRAM (often local to CPU chip). The reason this strategy works: LOCALITY.

- *Temporal locality: If a location has been accessed recently, it is likely to be accessed (reused) soon*
- *Spatial locality: If a location has been accessed recently, it is likely that nearby locations will be accessed soon*

AMAT(Average Memory Access Time) = HitTime + MissRatio * MissPenalty



Problem 1. ★

The timing for a particular cache is as follows: checking the cache takes 1 cycle. If there's a hit the data is returned to the CPU at the end of the first cycle. If there's a miss, it takes 10 *additional* cycles to retrieve the word from main memory, store it in the cache, and return it to the CPU. If we want an average memory access time of 1.4 cycles, what is the minimum possible value for the cache's hit ratio?

Minimum possible value of hit ratio: **0.96**

$AMAT = HitTime + MissRatio \times MissPenalty$
$1.4 = 1 + (1 - HitRatio) \times 10 \quad \Rightarrow \quad HitRatio = 0.96$

An important note is that the MissPenalty is the number of *additional* clock cycles required to retrieve the data on a cache miss. In this example, the problem specifies that "10 additional clock cycles" are needed. Thus, the total number of clock cycles required to retrieve data on a cache miss is 11, but the penalty is 10.

Problem 2.

The RISC-V Engineering Team is working on the design of a cache. This cache takes *2 clock cycles* to determine if a memory access is a hit or a miss and, if it's a hit, return data to the processor. If the access is a miss, the cache takes *20 additional clock cycles* to fill the cache line and return the requested word to the processor. If the hit rate is 90%, what is the processor's average memory access time in clock cycles?

Average memory access time assuming 90% hit rate (clock cycles): **4**

HitTime = 2 clock cycles
MissRatio = 1 – HitRatio = 1 – 90%

MissPenalty = 20
$AMAT = 2 + (1 - 90\%) \times 20 = 4$

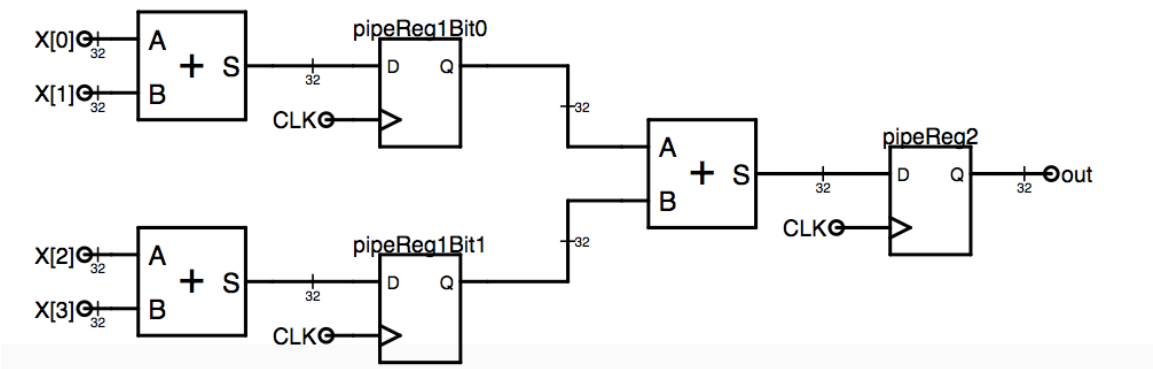
The remainder of this worksheet contains some more pipelining practice in Minispec and a repeat of the questions from R11 about Sequential Logic.

Problem 3. ★

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

```
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.

```
module PipelinedAdd4;
    RegU#(Vector#(2, Word)) pipeReg1;
    RegU#(Word) pipeReg2;
    input Vector#(4, Word) in;
    method Word out = pipeReg2;

    rule tick;
        // First stage
        Vector#(2, Word) v;
        v[0] = in[0] + in[1];
        v[1] = in[2] + in[3];
        pipeReg1 <= v;
```

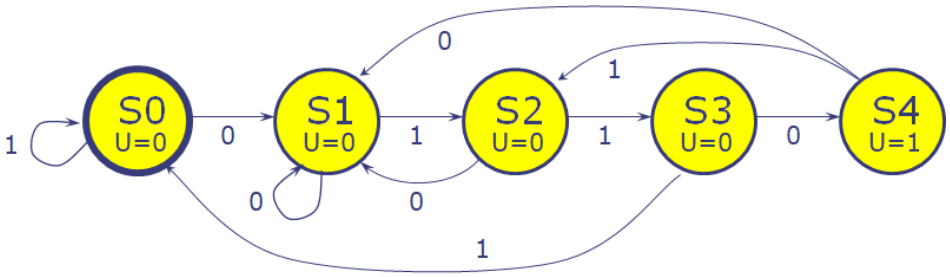
```
            // Second stage
            pipeReg2 <= pipeReg1[0] + pipeReg1[1];
        endrule
    endmodule
```

(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];
                v[1] = x[2] + x[3];
                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
```

Problem 4. ★

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 all the states in 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 5. ★

Implement the Fibonacci FSM from the Sequential Circuits worksheet by filling in the code skeleton below.

From sequential circuits worksheet:
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.

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

*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**.*

```
// 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 6.

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

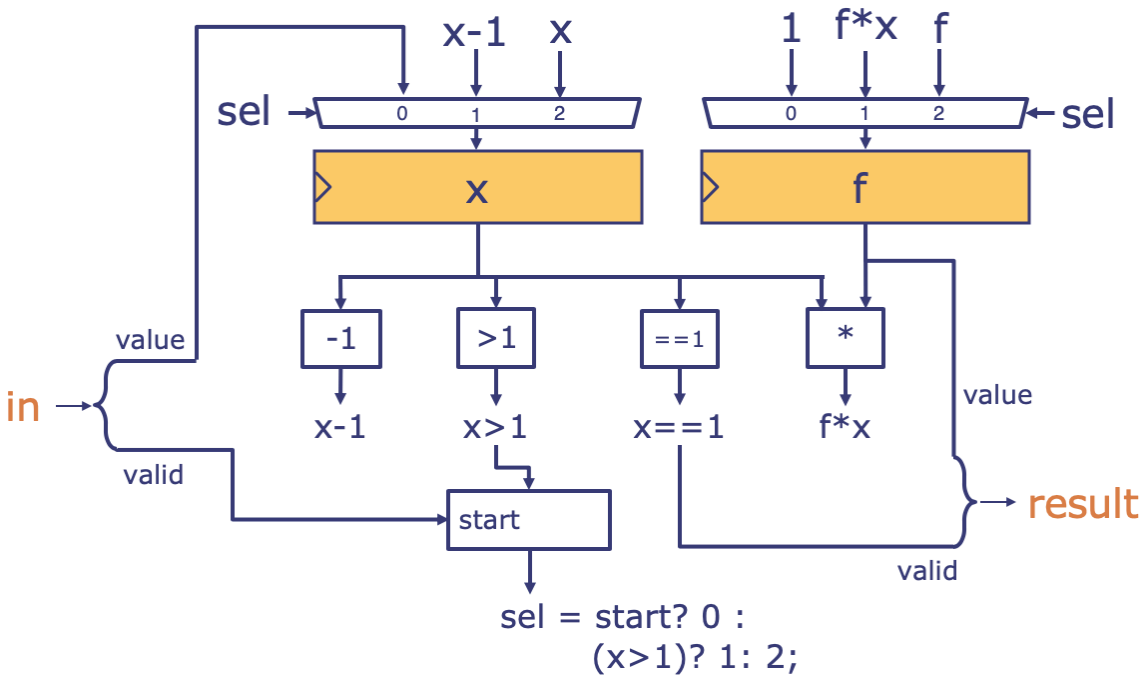
(A) Design the circuit as a sequential Minispec module by filling in the skeleton code below. The circuit should start a new factorial computation when a Valid input is given. Register **x** should be initialized to the input argument, and register **f** should eventually hold the output. When the computation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

Similar to the previous problem, 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
- x>1 tells us if we are in the middle of a calculation or if we are just holding values in the registers
- We calculate f*x through multiplying f and x.

Sel:

- The validity of the in signal tells us if we are able to start. We see that the signal carrying isValid(in) is fed into the start signal. If start is true, then 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