6.004 Tutorial Problems
L13 – Design Tradeoffs in Sequential Circuits

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

Note: These problems mainly seek to cover the concepts in lecture by implementing them in Minispec. This will be useful for labs, but we won't ask you to write this much code in the quiz.

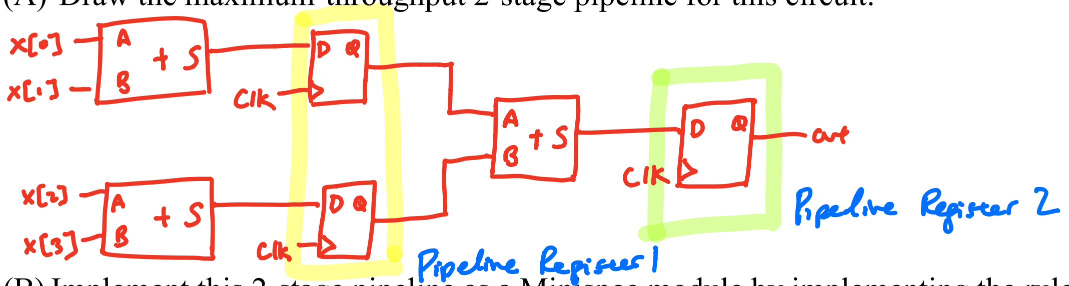
Problem 1. ★

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#(2, Word) in;
    method Word out = pipeReg2;

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

     //stage 2
     if (isValid(pipeReg1)) begin
        Vector #(2, Word) x = fromMaybe(?, pipeReg1);
        pipeReg2 <= Valid(x[0] + x[1]);
     end else pipeReg2 <= Invalid;


    endrule
endmodule
```

(D) Complete the skeleton code below to implement a 2-stage pipeline with valid bits and stall logic. Your pipeline should make progress if one of the stages has an ~~invalid~~ valid value.

never have a case where both stuck being valid <u>unless</u> you continuously share inputs into in.

Valid in every cycle ⇒ JAM

```
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;

    input Bool stallIn default = False;
                  ONE SHOT OR CONTINOUS

    // User module will stall producer if
    // stall input is set and pipeline is full
    method Bool isFull

        = isValid(pipeReg2) && isValid(pipeReg1);

    rule tick;
        Bool stall2 = stallIn && isValid(pipeReg2);
        Bool stall1 = stall2 && isValid(pipeReg1);
        //First stage
        if (!stall1) begin
            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;
        end
        //Second Stage
        if (!stall2) begin
            if (isValid(pipeReg1)) begin
                Vector#(2, word) x = fromMaybe(?, pipeReg1);
                pipeReg2 <= Valid(x[0] + x[1]);
            end else pipeReg2 <= Invalid;
        end

    endrule
endmodule
```

True → stalls after we computation

False → never stalls.

This is only going to run one time. [until both pipeReg1 & pipeReg2 are valid)

⇒ after that, the 2 pipeRegs need to reset

$\underbrace{!(\text{stallIn} \ \&\& \ \text{isValid(PipeReg2)})}$

Runs !stall2

equivalent to: $\overline{\text{stallIn}}$ OR $\overline{\text{isValid(PipeReg2)}}$

if pipeReg1 is invalid, pipeReg2 ALWAYS invalid

Problem 2. ★

In lecture, we have seen how to increase throughput with pipelining. But we cannot easily pipeline multi-cycle sequential circuits. To increase throughput in this case, we can instead use several multi-cycle circuits in parallel.

Consider the `Factorial` module from the L11 worksheet (reproduced below for completeness, although you do not need to understand its internals, only its interface):

```
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
```

We want to implement a module `MultiFactorial` that uses *two* copies of the `Factorial` module to improve throughput. `MultiFactorial` has a similar interface to `Factorial`: it has a `Maybe` input `enqueue` that, when set to `Valid`, starts a new factorial computation, and a `Maybe` output `result`, which is `Valid` when there is a new factorial result.

However, `MultiFactorial` can perform up to two computations in parallel: the module user can give up to two `Valid` inputs (over different cycles), and the module will return their outputs through the `result` method, in the same order that the inputs were given.

Under the covers, `MultiFactorial` should implement this behavior by alternating computations between its two `Factorial` submodules, `f[0]` and `f[1]`.

Since there are multiple computations in flight, the interface of `MultiFactorial` is similar to that of a FIFO queue. Specifically:

example of using a queue

– the problem is implemented annoyingly / is wrong

- The user of MultiFactorial enqueues a new input by setting the enqueue input to a Valid value. MultiFactorial also includes an isFull method to signal whether it's ready to accept a new input. If isFull is True, enqueue should not be set to a Valid value, and MultiFactorial need not process the value at the enqueue input.
- The user of MultiFactorial reads a ready output through the result method, and consumes it by setting the dequeue input to True. When dequeue is set to True, MultiFactorial should advance its output to the next result. MultiFactorial should produce results in the same order that the inputs were given. result should return Invalid if the next result to be consumed is not ready yet, or if there are no ongoing computations.

(A) Complete the skeleton code below to implement MultiFactorial.

↙ put computations into

head, tail
reading from

head & tail
⇒ 2 positions in the f vector,
but they switch

everytime you read head & tail switch

```
module MultiFactorial;
    Vector#(2, Factorial) f;

    Reg#(Bit#(1)) head(0);     // use output of this module
    Reg#(Bit#(2)) inFlight(0); // number of computations
                               // in flight (0, 1, or 2)

    input Maybe#(Bit#(16)) enqueue default = Invalid;
    method Bool isFull = (inFlight == 2);

    input Bool dequeue default = False;
    method Maybe#(Bit#(16)) result =
        (inFlight > 0) ? f[head].result : Invalid;

    rule tick;
        let nextInFlight = inFlight;
        if (dequeue && inFlight > 0) begin
            head <= head + 1;
            nextInFlight = nextInFlight - 1;
        end
        if (isValid(enqueue) && nextInFlight < 2) begin
            Reg#(Bit#(1)) tail = head + inFlight[0];
            f[tail].in = enqueue;
            nextInFlight = nextInFlight + 1;
        end
        inFlight <= nextInFlight;
    endrule
endmodule
```

wire for inFlight

fib(5)

if inflight = 0

head: 0   1

f: ⟨ F , F ⟩
tail   head

inflight: # F being calculated at once

incrementing head
head = ~head

deque
return other one next

enqueue
using value of head → which one we wanna return from

& nextInFlight = inflight at beginning

deque ↓   enque ↑

return in order given

⇒ tail = head + inFlight[0];

taking 0th bit (either 0 or 1), adding to head (0 or 1)

$$\text{head} \wedge \text{inFlight}[0] = \underbrace{\text{head} + \text{inFlight}[0]}_{\text{cast to 1-bit}}$$

(B) Manually synthesize the `MultiFactorial` module. Use the `Factorial` submodules as black boxes (i.e., connect their inputs and outputs but do not draw their internals).

head	inFlight	tail
0	00	0
1	00	1
0	01	1
1	01	0
0	10	X
1	10	X

} don't care, don't put anything

Problem 3.

In lecture, we saw the implementation of a 2-element FIFO (first-in, first-out) queue. Complete the skeleton code below to implement an n-element FIFO, using the same structure as the 2-element FIFO we have seen.

```
module FIFO#(Integer n, type T);
    Vector#(n, Reg#(Maybe#(T))) elems(Invalid);

    method Maybe#(T) first = elems[0];
    method Bool isFull;
        Bool res = True;
        for (Integer i = 0; i < n; i = i + 1)
            res = _________________________;
        return res;
    endmethod

    input Bool dequeue default = False;
    input Maybe#(T) enqueue default = Invalid;

    rule tick;
        Bool needsEnqueue = isValid(enqueue);
        for (Integer i = 0; i < n; i = i + 1) begin
            // First, find next value of elems[i] given dequeue,
            // but not accounting for enqueue
            Maybe#(T) nextValue = ____________________




            // Enqueue to the first register that would be Invalid
            if (_________________________________) begin
                nextValue = enqueue;
                needsEnqueue = False;
            end
            elems[i] <= nextValue;
        end

        if (needsEnqueue)
            $display("Warning: Attempted enqueue to a full queue,
                      enqueued value ignored");
    endrule
endmodule
```

Problem 4.

Partial Products, Inc., has hired you as its vice president of marketing. Your immediate task is to determine the sale prices of three newly announced multiplier modules. The top-of-the-line Cooker is a pipelined multiplier. The Sizzler is a combinational multiplier. The Grunter is a slower sequential multiplier. Their performance figures are as follows (T is some constant time interval):

	Throughput	**Latency**
Cooker	1/T	5T
Sizzler	1/4T	4T
Grunter	1/32T	32T

Customers follow a single principle: Buy the cheapest combination of hardware that meets their performance requirements. These requirements may be specified as a maximum allowable latency, a minimum acceptable throughput, or some combination of these. Customers are willing to try any parallel or pipelined configuration of multipliers in an attempt to achieve the requisite performance.

You may neglect the cost (both financial and as a decrease in performance) of any routing, registers, or other hardware needed to construct a configuration. Concentrate only on the inherent capabilities of the arrangement of multipliers itself.

It has been decided that the Cooker will sell for $1000. The following questions deal with determining the selling prices of Sizzlers and Grunters.

(A) How much can you charge for Sizzlers and still sell any? That is, is there some price for Sizzlers above which any performance demands that could be met by a Sizzler could also be met by some combination of Cookers costing less? If there is no such maximum price, indicate a performance requirement that could be met by a Sizzler but not by any combination of Cookers. If there is a maximum selling price, give the price and explain your reasoning.

(B) How little can you charge for Sizzlers and still sell any Cookers? In other words, is there a price for the Sizzler below which every customer would prefer to buy Sizzlers rather than a Cooker? Explain your reasoning.

(C) Is there a maximum price for the Grunter above which every customer would prefer to buy Cookers instead? Give the price if it exists, and explain your reasoning.

(D) Is there a minimum price for the Grunter below which every customer would prefer to buy Grunters rather than a Cooker? Give the price if it exists, and explain your reasoning.

(E) Suppose that, as a customer, you have an application in which 64 pairs of numbers appear all at once, and their 64 products must be generated in as short a time as practicable. You have $1000 to spend. At what price would you consider using Sizzlers? At what price would you consider using Grunters?