# 6.004 Worksheet Questions
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.

Common pipeline extensions: Valid bits and stall logic

So far, we have seen pipelines that take one input and produce one output every cycle. But when we integrate pipelines with other circuits, we often need to add a couple of extensions. Consider the diagram below, where there is some producer circuit that feeds inputs to the pipeline, and some consumer circuit that consumes the pipeline's outputs:
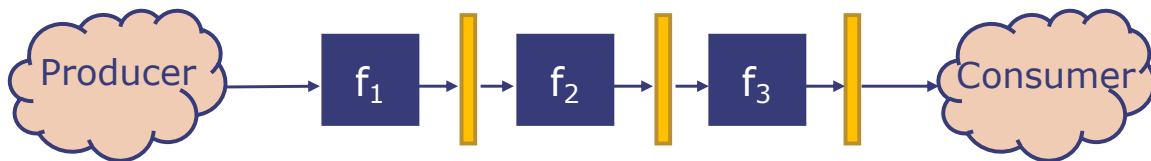
1. **Valid bits:** The producer circuit may not give a value to the pipeline every cycle. To address this, we can tag each stage with a valid bit.
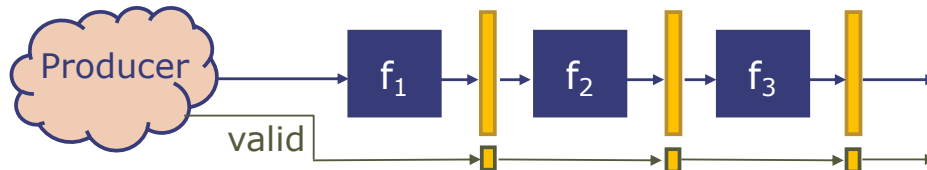

In Minispec, this can be achieved simply by using Maybe types for the input, pipeline registers, and output.

Valid bits propagate through the pipeline, so that an invalid input results in an invalid value at the output of the pipeline, as shown in the pipeline diagram below:

| Cycle	1	2	3	4	5	6	7	8
Stage 1	V1	V2	**Inv**	V3	**Inv**	**Inv**	V4	V5
Stage 2		V1	V2	**Inv**	V3	**Inv**	**Inv**	V4
Stage 3			V1	V2	**Inv**	V3	**Inv**	**Inv**

2. **Stall logic:** If the consumer circuit cannot accept a new value every cycle, we need to be able to *stall*, i.e., to freeze the pipeline (and the producer!). We can achieve this by having a stall signal and using pipeline register with an enable input:
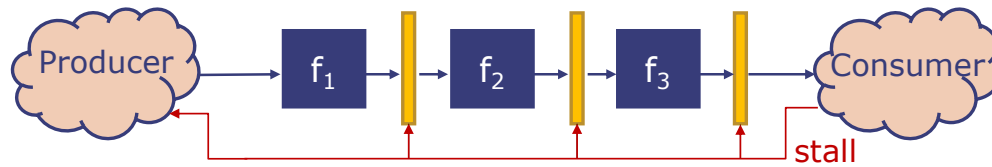
Here, the stall signal is an input controlled by the consumer. If the consumer can't take a value, it sets stall to True. This causes all registers to hold their current values, freezing the pipeline. The pipeline also passes the stall signal to the producer, as it cannot take in a new input:

Cycle	1	2	3	4	5	6	7	8
Stage 1	V1	V2	V3	V4	V4	V5	V5	V5
Stage 2		V1	V2	V3	V3	V4	V4	V4
Stage 3			V1	V2	V2	V3	V3	V3
Stall	False	False	False	True	False	True	True	False

3. **Combining valid bits and stall logic:** If the consumer stalls and the producer does not produce an input every cycle, we can still make progress if a stage has an invalid value:
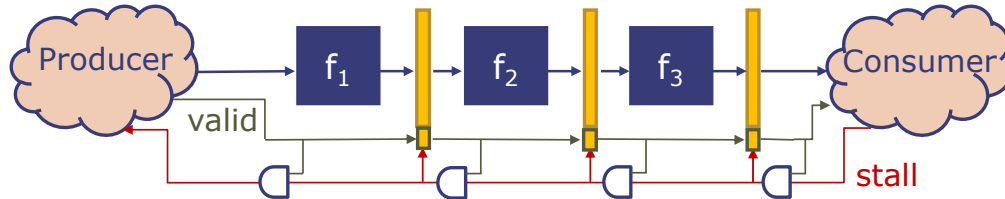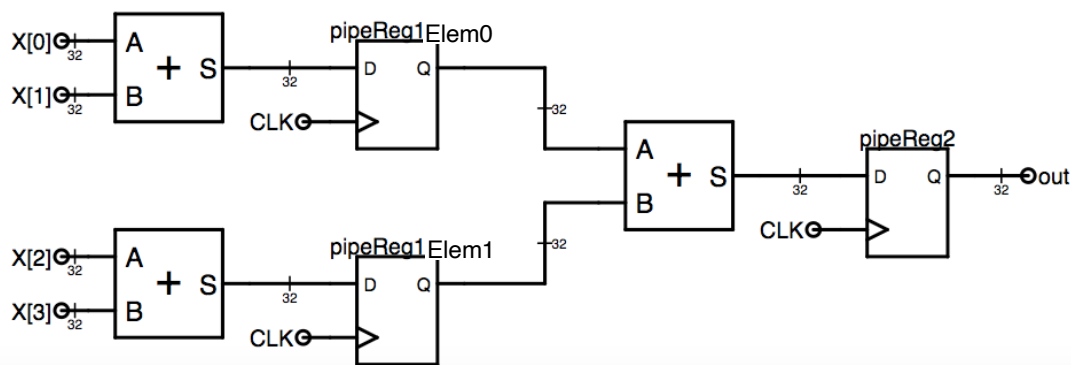
Problem 1 ★

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#(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
```

(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 value.

```
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;

    // User module will stall producer if
    // stall input is set and pipeline is full
    method Bool isFull
        = isValid(pipeReg2) && isValid(pipeReg1);

    rule tick;
        // Per-stage stall signals
        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
```

Problem 2 ★

Complete the skeleton code below to implement a parametric adder.

```
module PipelinedAddN#(2);
    Reg#(Maybe#(Word)) result(Invalid);

    input Maybe#(Vector#(2, Word)) in default = Invalid;
    method Maybe#(Word) out = result;

    rule tick;
        if (isValid(in)) begin
            Vector#(2, Word) x = fromMaybe(?, in);
            result <= Valid(x[0] + x[1]);
        end else begin
            result <= Invalid;
        end
    endrule
endmodule

module PipelinedAddN#(Integer n);
    Reg#(Maybe#(Word)) result(Invalid);
    PipelinedAddN#(n / 2) low;
    PipelinedAddN#(n / 2) high;

    input Maybe#(Vector#(n, Word))  in default = Invalid;
    method Maybe#(Word) out = result;

    rule tick;
        // Stage 1
        if (isValid(in)) begin
            low.in = Valid(take(fromMaybe(?, in)));
            high.in = Valid(takeTail(fromMaybe(?, in)));
        end else begin
            low.in = Invalid;
            high.in = Invalid;
        end

        // Stage 2
        if (isValid(low.out) && isValid(high.out)) begin
            result <= Valid(fromMaybe(?, low.out) + fromMaybe(?,
high.out));
        end else begin
            result <= Invalid;
        end
    endrule
endmodule
```
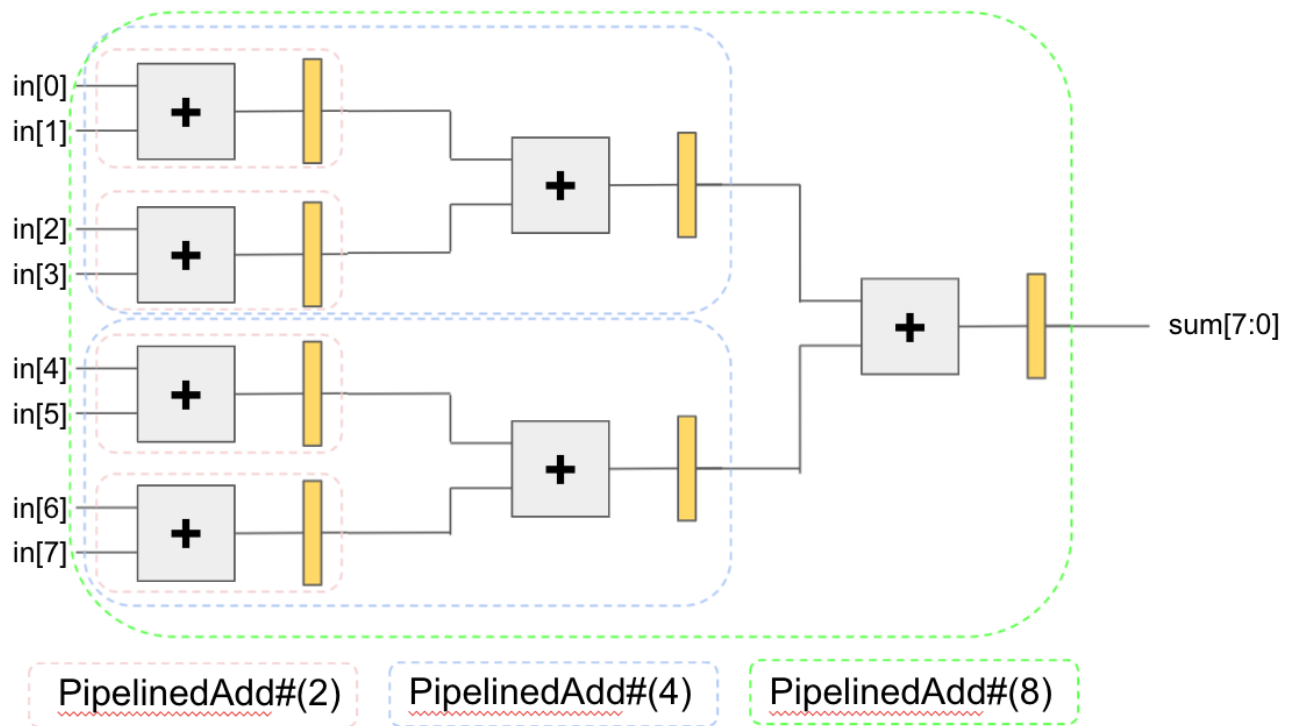
Manually synthesize `PipelinedAddN#(8)`. The components you have available to you are:

N-bit adders: Registers:

PipelinedAdd#(2)   PipelinedAdd#(4)   PipelinedAdd#(8)

Problem 3

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:
- 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.

```
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
            let tail = head + inFlight[0];
            f[tail].in = enqueue;
            nextInFlight = nextInFlight + 1;
        end
        inFlight <= nextInFlight;
    endrule
endmodule
```

NOTE: This solution accepts a new input even when isFull==True as long as dequeue is also set to True on the same cycle (so both Factorial submodules can be kept fully utilized). Simpler solutions that do not do this are also correct, e.g., by checking for inFlight < 2 instead of nextInFlight < 2 (and in this case, the rule need not handle dequeues before enqueues). However, they wouldn't work with the skeleton code that was provided.

(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).
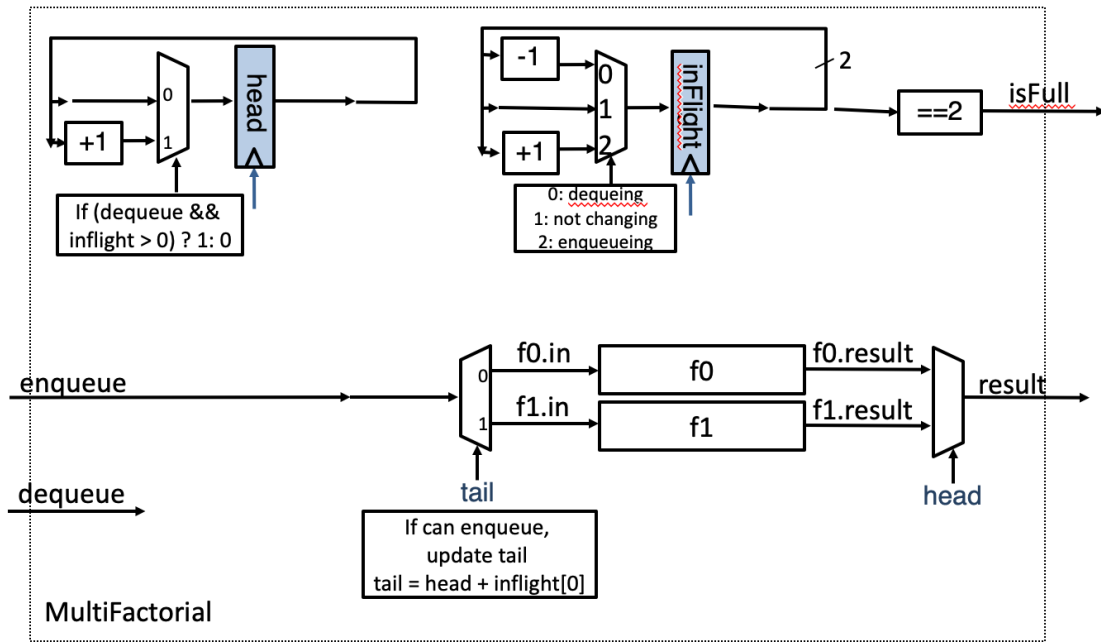
Problem 4

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.

If there is a performance requirement for the latency to be < 5T, then there is no combination of Cookers that will meet this performance requirement. So it is in theory possible to sell some Sizzlers at any price. Using multiple Cookers can further improve the overall multiplier throughput, but their latency cannot be shortened.

(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.

The price of a Sizzler must be >$250 if we want to continue to sell Cookers. If the price of a Sizzler is $250 or lower, 4 Sizzlers could be used in parallel to achieve the same throughput as a Cooker with a better latency in the bargain.

(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.

The price for the Grunter must be <$1000, since for applications that can accept long latencies (>= 32T) or low throughputs, it's worth buying a Grunter if it saves any money at all.

(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.

There is no minimum price for a Grunter that would cause every customer to buy Grunters instead of Cookers. The latency of the Grunter will always be 32T, so when performance requirements demand latencies < 32T, Grunters won't do the job.

(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?

Sizzlers will be considered when they cost $250 or less (otherwise, a single Cooker will yield better throughout).

If Sizzlers cost $100 or less, then Grunters are not acceptable at any price point, because 10 Sizzlers achieve a latency of 28T, less than that of a single Grunter.

If Sizzlers more than $100 but less than $250, a solution consisting only of Grunters is preferable if they cost $1000/64 = $15.62 or less (as 64 Grunters in parallel achieve a 32T latency).

Moreover, for some Sizzler price points, we can achieve slightly higher throughput with a solution that consists of both Sizzlers and Grunters, using Grunters to spend our remaining budget. Among these cases, the highest price we may consider for a Grunter is $124.93. This happens when Sizzlers cost $125.01 (so we can afford seven of them, but not eight). Buying seven Sizzlers would yield a latency of 40T at a cost of $875.07. We cannot afford another Sizzler, but adding a single Grunter for $124.93 will reduce the latency to 36T.