Again, we’d like to encourage you to use the lab time today to finish problems in Labs 2 and 3. Work on the exercises that are most interesting to you. In this handout, we provide a few simple exercises to give you more variety to choose from.

Remember that you can refer to the “How-To” handout from Monday for details on running the Bluespec compiler and simulating the output. More complete documentation on the Bluespec compiler is available in the “Bluespec Compiler User Manual” on the IAPBlue website. The Bluespec language and standard libraries are documented in the “Bluespec Language Manual,” also on the website.

**Exercise 1**

5-stage pipeline with bypasses

Let’s get the 5-stage pipeline working with bypasses.

**Part a:**

Given the source code for **S FIFO** (*stall* FIFO) from Lab 3, can you write **BFIFO** (*bypass* FIFO) which has the following interface:

```haskell
interface BFIFO t =
  eqn  :: t -> Action
  first :: t
  deq  :: Action
  clear :: Action
  find  :: (t -> Bool) -> Maybe t
```

The difference is the `find` method. In the stall FIFO, we simply reported whether an element in the FIFO matched a certain condition. In the bypass FIFO, we return the matched element (using the `Maybe` type) so that we can check whether it has a result that can be routed to another stage.

**Part b:**

Using the BFIFO, change the FiveStageCPUStall code to include bypasses to the decode stage.

**Part c:**

Write some simple programs which would stall without bypassing and run them through both designs, observing that the version with bypasses does execute the program in less cycles.

**Exercise 2**

Modular CPU pipeline

The 5-stage pipeline that we gave you in Lab 3 is written as one monolithic module. All state and rules for all stages are defined in the toplevel module. Let’s explore the possibilities for connecting modules in Bluespec by rewriting the 5-stage pipeline in a more modular form.

You can start by defining modules such as:
type FetchFIFO = FIFO (Ia, Bit 32)
type DecodeFIFO = SFIFO (Ia, InstTemplate)
type ExecFIFO = SFIFO (Ia, InstTemplate)
type MemFIFO = SFIFO (Ia, InstTemplate)

mkFetchStage :: InstrMem -> Reg Ia -> FetchFIFO -> Module Empty
mkFetchStage imem pc bf = ...

mkDecodeStage :: FetchFIFO -> DecodeFIFO -> ExecFIFO -> MemFIFO -> Module Empty
mkDecodeStage bf bd be bm = ...

mkExecuteStage :: DecodeFIFO -> ExecFIFO -> Module Empty

mkMemStage :: DataMem -> ExecFIFO -> MemFifo -> Module Empty

mkWBStage :: RegFile -> MemFifo -> Module Empty

These definitions assume that the buffers are instantiated outside of each stage and passed in as arguments. You could also write modules which instantiate their own buffers and export the enq or deq side to the outside world. A third option would be to have the modules export interfaces in the Connectable class and wire them together with connect.

Once you have written the submodules, put them together in a brief toplevel module.

---

Exercise 3

The FIFO interface that we have shown you in class has the following form:

```haskell
interface FIFO t =
    enq :: t -> Action
    first :: t
    deq :: Action
    clear :: Action
```

Reading the head of the FIFO (first) and dequeuing the head of the FIFO (deq) are separate methods. This leaves open the possibility that a design can read the head of a FIFO and forget to perform the dequeue action. If we want to add some safety to our design and enforce the rule that reading the head of a list must always be combined with dequeuing it, then we can use the type `ActionValue`.

We could write the FIFO interface as follows:

```haskell
interface FIFO t =
    enq :: t -> Action
    deq :: ActionValue t
    clear :: Action
```

An `ActionValue` can only be accessed inside an `Action` block. For example:

```haskell
when (condition)
    => action
        next_elem <- fifo1.deq
        fifo2.enq next_elem
```
IAPBlue  Lab Exercises 4

When used in a module, the “<−” symbol means that the module on the right is being placed into the current design and its interface is being given the name on the left. Similarly, in an action block, the arrow means that the action part of the ActionValue is being added to the action block and the value is being given the name on the left (here next_elem).

To create an ActionValue type, write an Action block and include “return value” as the last line. For example, the deq method could now be written as:

```
  deq = action
      perform the dequeue action
  return the first element
```

Part a:

Change the SFIFO or BFIFO in your 5-stage pipeline design to use ActionValue and alter the pipeline source to use the new FIFO. You should be able to compile and run the CPU testbench to confirm that the altered code still works.

Part b:

The ActionValue type can be used in any situation where an Action is associated with a return value. For example, in the IP lookup example, we had a completion buffer which granted tokens upon request. The request required an acknowledgement, which is an Action:

```
interface CBuffer n a =
  getToken :: CBToken n
  getTokenAck :: Action
  done :: CBToken n → a → Action
  get :: a
  ack :: Action
```

Rewrite the completion buffer in CBuffer.bs (in the IP lookup files provided with Lab 2) to use ActionValue for getToken and get, instead of requiring acknowledgement.

---

Exercise 4  Reorder Buffer

Looking for something more challenging? Write a module which implements the reorder buffer interface RoB given in lecture.

A reorder buffer is a circular buffer, in which instructions are added in order and retired in order. The SFIFO that we provided in Lab 3 is implemented as a circular buffer. You might want to study the source code for SFIFO and try to use the same mechanism for the reorder buffer.