Massachusetts Institute of Technology – Laboratory for Computer Science

IAPBlue A Mini Course on Bluespec

http://www.csg.lcs.mit.edu/IAPBlue

Lab Exercises 3 15 Jan 2003

Since there was a lot of material in Lab 2, we're also giving you today to finish those exercises. When you finish with Lab 2 or are just ready to move on to something new, we've provided a few exercises here on the topics in today's lectures. This lab gives you practice with abstract types and bit packing, as well as chance to explore changes to a 5-stage CPU pipeline.

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 documention 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 Divide by three

In the Lab 2 exercises, you wrote several implementations of divide-by-three circuits. In those exercises, we asked you to represent the remainder as the type `(Bit 2)`. This meant that some of your implementations had `case`-statements that didn't cover the entire range of values or else covered values (such as 3) that we don't ever expect to encounter.

We could have defined the following type:

```
data Mod3 = Rem0 | Rem1 | Rem2
    deriving (Bits, Eq)
```

We ask the compiler to make an instance of this type in the `Bits` class because we want to be able to store it in a register, which means that there must be a bit representation for it. We ask for an instance in the class `Eq` because we need to do comparisons or pattern matches on this type – for example, we will want to check when the remainder is zero

Rewrite one of your divide-by-three modules to use the `Mod3` data type.

Exercise 2 Bits class

In Lab 2, you were asked to implement an array and a FIFO using registers. In general, the `Array` interface looks like this:

```
interface Array i a =
    upd :: i -> a -> Action
    sub :: i -> a
```

However, in order to avoid talking about type classes in Lab 2, we asked you implement an array whose index and elements are of type `(Bit n)`. Now that we know more about type classes, it should not confuse you to see `mkArray` with the following type:

```
mkArray :: (Bits i is, Bits a as) => Module (Array i a)
```

Change your implementation of `mkArray` from Lab 2 to have this more general type. It should only involve a few small changes.

Alternatively, you can change your implementation of `mkFIFO` to have this more general type:

```
mkFIFO :: (Bits t ts) => Module (FIFO t)
interface FIFO t =
    enq :: t -> Action
    ...
```

Exercise 3 CPU with 5-stage pipeline

We have provided you with the source code for a 5-stage pipeline with stalls but no bypasses, using the simple instruction set from the first lectures. The code is in `FiveStageCPUStall.bs`. To compile the CPU module, you might need to give the flag `-no-opt-sched-bdd` to turn off expensive BDD processing. We have provided a `Makefile` to compile the CPU and the included testbench for you.

Part a:

Add three more instructions to the `Instr` data type:

1. `LoadC`, which loads a constant value into a register
2. `LoadPC`, which loads the value of the PC into a register
3. `LShift`, which left shifts a register value by another register value

You will have to change the `Instr` type definition, of course. You will also need to add rules to the decode stage and execute stage to handle these new instructions. And don't forget to also consider the new rules in the stall condition in the decode stage!

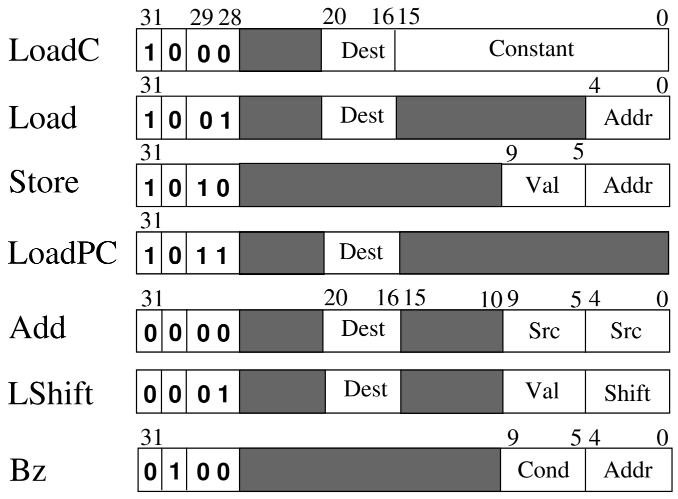
Once you have added these instructions, you will need a test bench to run the CPU and test whether you wrote the instructions correctly. We have provided a testbench called `mkCPUTest` in the file `CPUTest.bs`. The `Makefile` we have provided will compile the binary `CPUTest` when you type `make`. The testbench loads a very simple program into instruction memory, starts the CPU running, and makes use of the `PrintF` library to print out the value of data memory location 10 on every cycle. If you have implemented the new instructions more or less correctly, you should see the value 8 appear after a dozen or so cycles.

You may change the testbench to load your own programs into memory. If you do write any interesting programs for this simple instruction set, feel free to share them with the class!

Part b:

Notice that the `Instr` data type has a derived instance of the `Bits` class. This means that the compiler picks a straightforward packing scheme. What if the encoding of our instruction set into Bits doesn't match the straightforward encoding? Then it would be necessary to define the `pack` and `unpack` functions by hand.

Remove the `Bits` class from the `deriving`-clause of the `Instr` data type (but leave the `Eq` class!) and write your own instance of `Bits` given the following instruction set encoding:



To write an instance of `Bits`, you will need to add the following lines and fill in the definition for `pack` and `unpack`:

```
instance Bits Instr 32 where
    pack ...
    unpack ...
```

Part c:

Add a more advanced mechanism for branch prediction to the pipeline.

In the current implementation, the fetch instruction always assumes that a branch will fail. It keeps fetching "`pc + 1`" after a branch instruction. If, in a later stage, it is determined that the branch should have jumped to a different address, then a signal is sent back to clear all the incorrectly pre-fetched instructions and to set the PC to the address of the jump.

One mechanism for more accurate branch prediction would be to keep a record of how often a branch resulted in a jump or not, and use that information to predict the path which has been most often taken. You can implement this with a hash table associating the branch instruction's memory location with the number of times the branch was taken and not taken. Then, when that branch instruction is encountered again, the fetch stage can take the path which has been most common. The information that is enqueued into the pipeline now has to include information on which branch was taken so that a later stage can determine whether an incorrect prediction was made. If the predition was incorrect, the pre-fetched instructions have to be cleared and the PC set to other location. (So you might want to record this other location in the information that you send down the pipeline.)

Making this change will thus involve changing the types of the FIFOs in the pipeline to include additional information.

Part d:

If you have time, add a `Multiply` instruction to the instruction set. But don't implement it as an additional operation in the execute stage. That would be too easy! Implement it by having the hardware enqueue the appropriate `Shift` and `Add` instructions into the pipeline.