BSV Training

Eg05: Concurrent Registers (CRegs) and Greater Concurrency

Using two examples (a 2-port counter and a FIFO implementation), we see that CRegs permit greater concurrency than traditional registers.

www.bluespec.com

Prerequisites and related material

Before you study the examples here in Eg05_Greater_Concurrency

you should understand the concept of “concurrency of rules” in BSV as described in:

- Lecture: Lec_Rule_Semantics
- Example: Eg03_Bubble_Sort

The following lectures:

- Lecture: Lec_CRegs
- Lecture: Lec_RWires

describe CRegs in greater detail (along with the related topic of RWires)

Eg05a: A counter with 2 concurrent ports

Each port is an ``increment method'', i.e., allows the counter to incremented.

By ``concurrent ports'' we mean that both methods can be invoked in the same clock.

The interface for the counter

File src_BSV/Counter2_IFC.bsv

```
interface Counter2_IFC;
   method ActionValue #(Int #(32)) count1 (Int #(32) delta);
   method ActionValue #(Int #(32)) count2 (Int #(32) delta);
endinterface
```

- The interface has two methods count1 and count2, with identical types.
- A module implementing this interface should have an internal register representing the current count.
- When either method is invoked, the argument is used to increment the internal counter, and it returns the old value of the counter.

bluespec

1st attempt, using ordinary registers

File src_BSV/Counter2_Reg.bsv

```
(* synthesize *)
module mkCounter2 (Counter2_IFC);

   Reg #(Int #(32)) rg <- mkReg (0);

   method ActionValue #(Int #(32)) count1 (Int #(32) delta);
      rg <= rg + delta;
      return rg;
   endmethod

   method ActionValue #(Int #(32)) count2 (Int #(32) delta);
      rg <= rg + delta;
      return rg;
   endmethod

endmodule: mkCounter2
```

But:

- count1 and count2 both read and write the register rg
- Because of the "_read < _write" ordering constraint on register methods, both methods could not be invoked in the same clock

==> they could never be concurrent

A testbench for the counter

File src_BSV/Testbench.bsv

```
module mkTestbench (Empty);

   Counter2_IFC ctr <- mkCounter2;

   Reg #(int) step <- mkReg (0);

   rule rl_step;
      step <= step + 1;
      if (step == 10) $finish;
   endrule

   rule rl_1 (step <= 6);
      let delta_1 = step + 10;
      let old_v_1 <- ctr.count1 (delta_1);
      $display ("%0d:         rl_1: delta_1 %0d    v_1 %0d", step, delta_1, old_v_1);
   endrule

   rule rl_2 (step >= 4);
      let delta_2 = 5 - step;
      let old_v_2 <- ctr.count2 (delta_2);
      $display ("%0d:         rl_2: delta_2 %0d    v_2 %0d", step, delta_2, old_v_2);
   endrule

endmodule: mkTestbench
```

- The testbench runs for 10 steps (see rl_step).
- rl_1 attempts to invoke the count1 method on steps 0-6
- rl_2 attempts to invoke the count2 method on steps 4-10
- Question: what happens on steps 4-6, when both attempt to fire?

Build and run, using Counter2_Reg.bsv

- In the “src_BSV” directory, create a symbolic link from “Counter2.bsv” to Counter2_Reg.bsv:

  % ln –s –f   Counter2_Reg.bsv Counter2.bsv

- In the Build directory, build and run using the ‘make’ commands, either with Bluesim or with Verilog sim, as described earlier.

- During compilation, note that bsc produces this warning message concerning the conflict between count1 and count2:

```
Warning: "src_BSV/Testbench.bsv", line 18, column 8: (G0010)
  Rule "rl_1" was treated as more urgent than "rl_2". Conflicts:
    "rl_1" cannot fire before "rl_2": calls to ctr.count1 vs. ctr.count2
    "rl_2" cannot fire before "rl_1": calls to ctr.count2 vs. ctr.count1
```

Build and run, using Counter2_Reg.bsv

Simulation output:

```
  0: rl_1: delta_1 10  old_v_1  0
  1: rl_1: delta_1 11  old_v_1 10
  2: rl_1: delta_1 12  old_v_1 21
  3: rl_1: delta_1 13  old_v_1 33
  4: rl_1: delta_1 14  old_v_1 46
  5: rl_1: delta_1 15  old_v_1 60
  6: rl_1: delta_1 16  old_v_1 75
  7:                                  rl_2: delta_2 -2    old_v_2 91
  8:                                  rl_2: delta_2 -3    old_v_2 89
  9:                                  rl_2: delta_2 -4    old_v_2 86
 10:                                  rl_2: delta_2 -5    old_v_2 82
```

Each line shows the step, the rule that fired, the argument to the method call (delta), and the returned previous value of the counter (v).

Observe that in cycles 4-6, only rl_1 fires.

2nd attempt: Concurrent Registers

File src_BSV/Counter2_CReg.bsv

```
import Counter2_IFC :: *;

(* synthesize *)
module mkCounter2 (Counter2_IFC);

   Reg #(Int #(32)) crg [2] <- mkCReg (2, 0);

   method ActionValue #(Int #(32)) count1 (Int #(32) delta);
      crg[0] <= crg[0] + delta;
      return crg[0];
   endmethod

   method ActionValue #(Int #(32)) count2 (Int #(32) delta);
      crg[1] <= crg[1] + delta;
      return crg[1];
   endmethod

endmodule: mkCounter2
```

- The internal register has been replaced by a CReg
- count1 uses port [0] of the CReg
- count2 uses port [1] of the CReg
- The can be invoked concurrently. If invoked concurrently:
  - the counter will be incremented by both delta arguments
  - the ``old value” returned by count2 will be the value incremented by count1

Build and run, using Counter2_CReg.bsv

- In the “src_BSV” directory, create a symbolic link from “Counter2.bsv” to the Counter2_CReg.bsv:

```
% ln –s –f   Counter2_CReg.bsv  Counter2.bsv
```

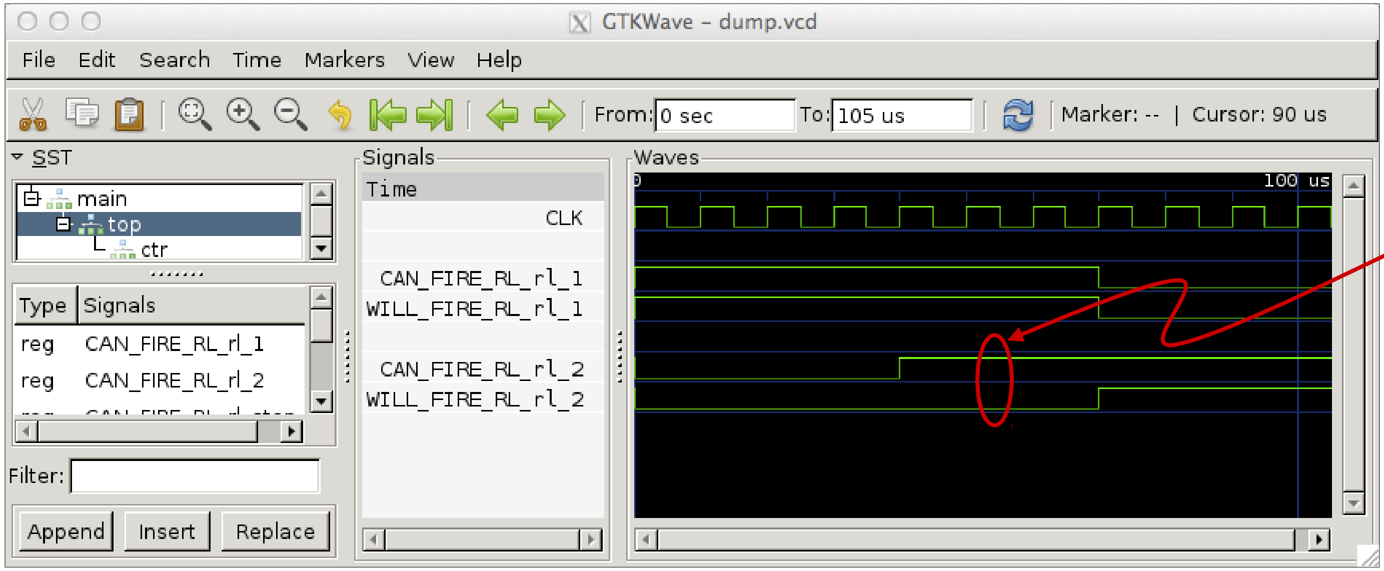
- In the Build directory, build and run using the ‘make’ commands, either with Bluesim or with Verilog sim, as described earlier.

Simulation output:

```
 0: rl_1: delta_1 10  old_v_1  0
 1: rl_1: delta_1 11  old_v_1 10
 2: rl_1: delta_1 12  old_v_1 21
 3: rl_1: delta_1 13  old_v_1 33
 4: rl_1: delta_1 14  old_v_1 46
 4:                                 rl_2: delta_2  1    old_v_2 60
 5: rl_1: delta_1 15  old_v_1 61
 5:                                 rl_2: delta_2  0    old_v_2 76
 6: rl_1: delta_1 16  old_v_1 76
 6:                                 rl_2: delta_2 -1    old_v_2 92
 7:                                 rl_2: delta_2 -2    old_v_2 91
 8:                                 rl_2: delta_2 -3    old_v_2 89
 9:                                 rl_2: delta_2 -4    old_v_2 86
10:                                 rl_2: delta_2 -5    old_v_2 82
```

Both rules fire in cycles 4-6

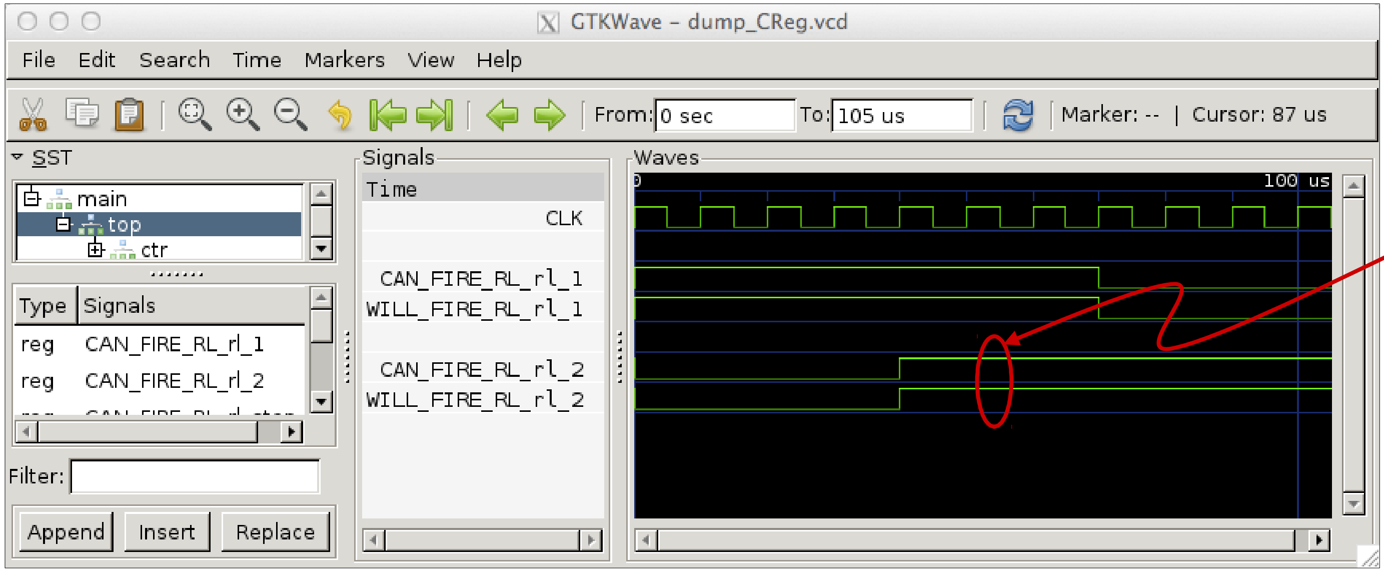
Waveforms



Picture file: Waves_Reg.tiff

Reg version:

CAN_FIRE_rl_2 is true, but WILL_FIRE_rl_2 is false as long as WILL_FIRE_rl_1 is true, because of the conflict on methods count1 and count2



Picture file: Waves_CReg.tiff

CReg version:

CAN_FIRE_rl_2 is true, and WILL_FIRE_rl_2 is also true even though WILL_FIRE_rl_1 is true, because of there the methods count1 and count2 do not conflict.

Suggested exercises

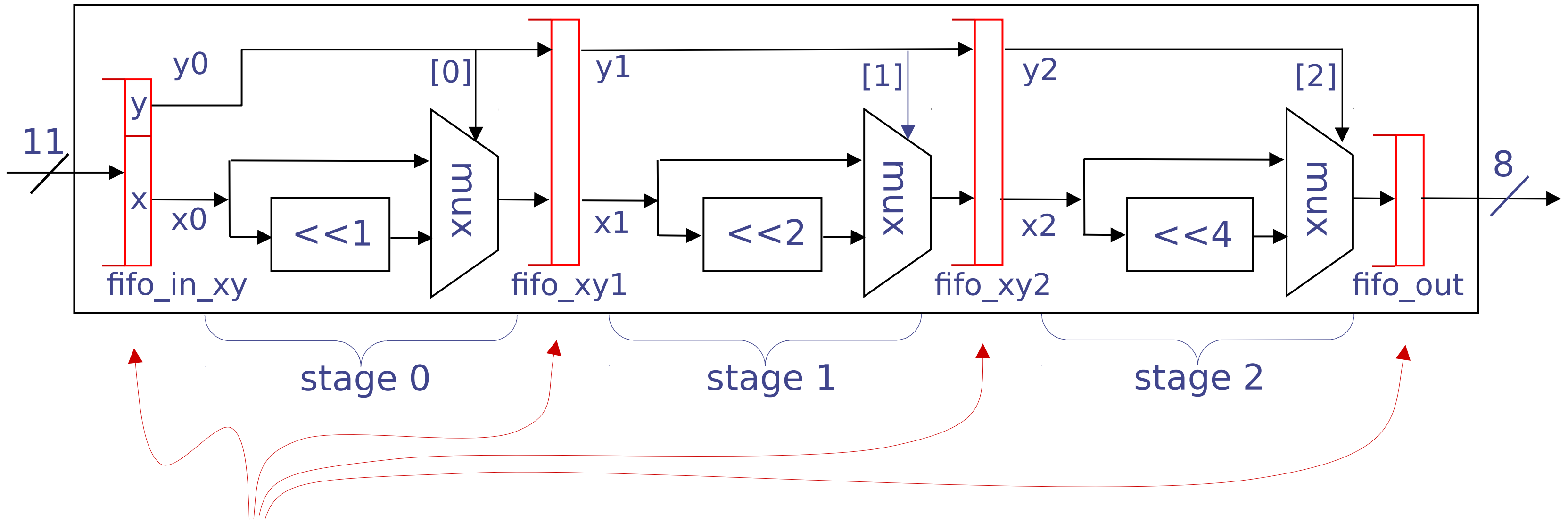
- What is the scheduling order of the two methods count1 and count2?
- Change the CReg port usage to get the opposite schedule for count1 and count2.
- Add a third concurrent count port to the counter.
- Add another method to the counter that is just a value method that returns its net post-increment value, taking into account that any or all of the count methods may be invoked in any clock.

Example 2: Concurrent FIFOs

By a ``Concurrent FIFO'' we mean a FIFO where the ``enq'' method can be invoked concurrently (in the same clock) as the ``first/deq'' methods .

Introduction and summary

- We use a copy of Eg04a_MicroArchs/src_BSV/Shifter_pipe_elastic.bsv and Testbench.bsv as our starting point:



- For the FIFOs in the design, we replace the BSV library mkFIFOF module with our own mkMyFIFOF module (implementing a FIFO of depth 1)
- Our first attempt (MyFIFOF_reg.bsv) will just use traditional registers, and we'll see that it does not have enough concurrency to pipeline properly
- Our second attempt (MyFIFOF_creg.bsv) will use CRegs instead of traditional registers; these will have enough concurrency to enable proper pipelining
- Takeaway lesson: with registers and CRegs one can implement microarchitectures with any desired degree of concurrency (and therefore performance)

1st attempt, using ordinary registers

File src_BSV/MyFIFOF_Reg.bsv

```
module mkMyFIFOF (FIFOF #(t))
   provisos (Bits #(t, tsz));

   Reg #(t)          rg       <- mkRegU;     // data storage
   Reg #(Bit #(1))   rg_count <- mkReg (0);  // # of items in FIFO (0 or 1)

   method Bool notEmpty = (rg_count == 1);
   method Bool notFull  = (rg_count == 0);

   method Action enq (t x) if (rg_count == 0);  // can enq if not full
      rg <= x;
      rg_count <= 1;
   endmethod

   method t first ()  if (rg_count == 1);  // can see first if not empty
      return rg;
   endmethod

   method Action deq () if (rg_count == 1);  // can deq if not empty
      rg_count <= 0;
   endmethod

   method Action clear;
      rg_count <= 0;
   endmethod
endmodule
```

But: enq and {first, deq} could never be concurrent, with mutually exclusive conditions:
rg_count == 0   and   rg_count == 1

==> enq could never execute in the same clock as {first,deq} (it isn't really a pipeline!)

bluespec

A testbench for the pipeline

File src_BSV/Testbench.bsv

```
module mkTestbench (Empty);
   Shifter_IFC  shifter <- mkShifter;

   Reg #(Bit #(4)) rg_y <- mkReg (0);

   rule rl_gen (rg_y < 8);
      shifter.request.put (tuple2 (8'h01, truncate (rg_y)));  // or rg_y[2:0]
      rg_y <= rg_y + 1;
      $display ("%0d: Input 0x0000_0001 %0d", cur_cycle, rg_y);
   endrule

   rule rl_drain;
      let z <- shifter.response.get ();
      $display ("                               %0d: Output %8b", cur_cycle, z);
      if (z == 8'h80) $finish ();
   endrule
endmodule: mkTestbench
```

- This is the same testbench as before (Eg04b).
- Rule rl_gen continuously attempts to feed the pipeline with the value 8’h01 and increasing shift amounts 0,1,2,…
- Rule rl_drain continously attempts to drain the output and displays it.
- If the shifter behaves like a proper pipeline, we should be able to feed and drain it on every cycle.

Build and run, using MyFIFOF_Reg.bsv

- In the "src_BSV" directory, create a symbolic link from "MyFIFOF.bsv" to MyFIFOF_Reg.bsv:

```
% ln –s –f   MyFIFOF_Reg.bsv   MyFIFOF.bsv
```

- In the Build directory, build and run using the 'make' commands, either with Bluesim or with Verilog sim, as described earlier.

Simulation output:

```
1: Input 0x0000_0001 0
3: Input 0x0000_0001 1
5: Input 0x0000_0001 2
                                5: Output 00000001
7: Input 0x0000_0001 3
                                7: Output 00000010
9: Input 0x0000_0001 4
                                9: Output 00000100
11: Input 0x0000_0001 5
                                11: Output 00001000
13: Input 0x0000_0001 6
                                13: Output 00010000
15: Input 0x0000_0001 7
                                15: Output 00100000
                                17: Output 01000000
                                19: Output 10000000
```

Observe that we can feed inputs into the pipeline and drain outputs from the pipeline only on every other cycle.

bluespec

2nd attempt, using CRegs (MyFIFOF_creg.bsv)

File src_BSV/MyFIFOF_CReg.bsv

```
module mkMyFIFOF (FIFOF #(t))
   provisos (Bits #(t, tsz));

   Reg #(t)          crg [3]       <- mkCRegU (3);      // data storage
   Reg #(Bit #(1))  crg_count [3] <- mkCReg (3, 0);   // # of items in FIFO

   method Bool notEmpty = (crg_count [0] == 1);
   method Bool notFull  = (crg_count [1] == 0);

   method Action enq (t x) if (crg_count [1] == 0);
      crg [1] <= x;
      crg_count [1] <= 1;
   endmethod

   method t first ()  if (crg_count [0] == 1);
      return crg [0];
   endmethod

   method Action deq () if (crg_count [0] == 1);
      crg_count [0] <= 0;
   endmethod

   method Action clear;
      crg_count [2] <= 0;
   endmethod
endmodule
```

Note: {first, deq} access CReg port [0], and enq accesses port [1].

These can run concurrently, in that order, thereby allowing pipelining.

bluespec

Build and run, using MyFIFOF_CReg.bsv

- In the “src_BSV” directory, create a symbolic link from “MyFIFOF.bsv” to MyFIFOF_CReg.bsv:

```
% ln –s –f   MyFIFOF_CReg.bsv   MyFIFOF.bsv
```

- In the Build directory, build and run using the ‘make’ commands, either with Bluesim or with Verilog sim, as described earlier.

Simulation output:

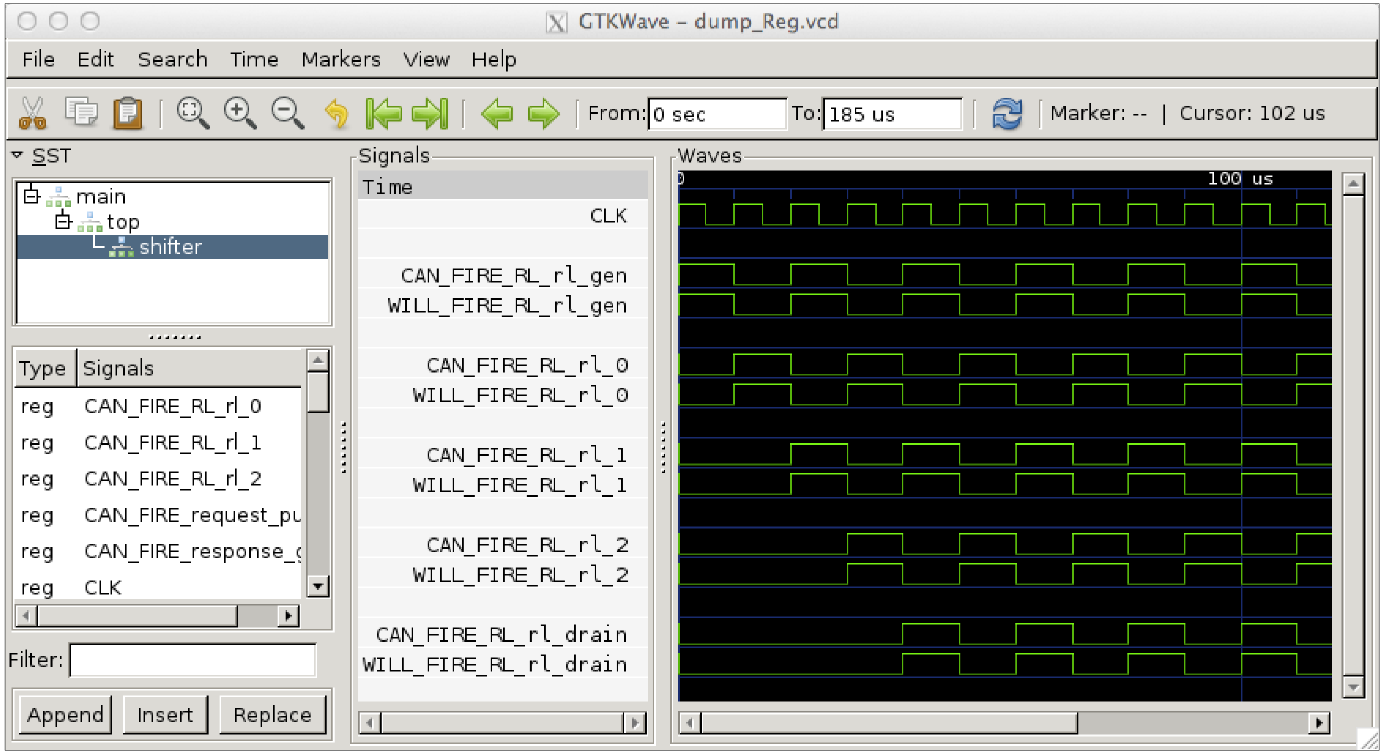
```
1: Input 0x0000_0001 0
2: Input 0x0000_0001 1
3: Input 0x0000_0001 2
4: Input 0x0000_0001 3
                                5: Output 00000001

5: Input 0x0000_0001 4
                                6: Output 00000010

6: Input 0x0000_0001 5
                                7: Output 00000100

7: Input 0x0000_0001 6
                                8: Output 00001000

8: Input 0x0000_0001 7
                                9: Output 00010000
                                10: Output 00100000
                                11: Output 01000000
                                12: Output 10000000
```

Observe that we can feed inputs into the pipeline and drain outputs from the pipeline on *every* cycle.

bluespec

Waveforms: using MyFIFOF_Reg.bsv

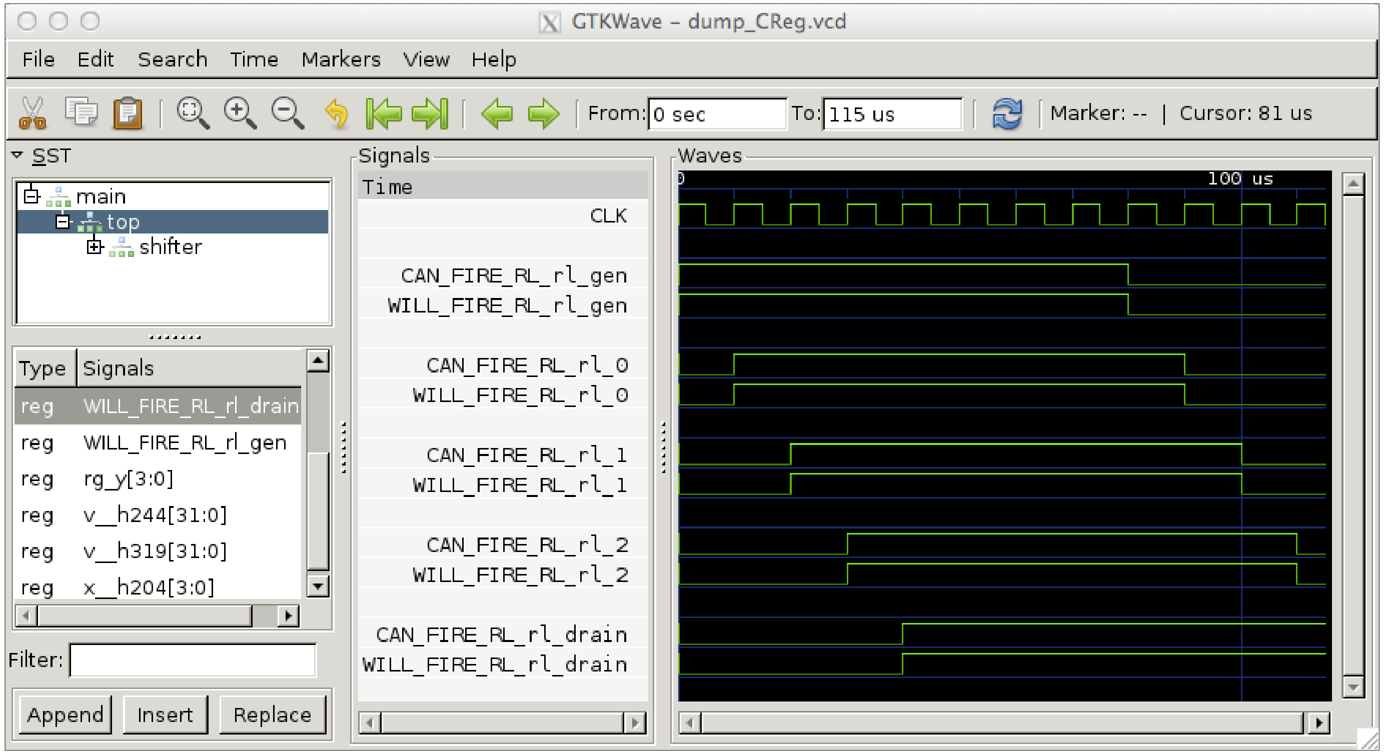
Picture file: Waves_Reg.tiff



rl_gen (feeding the pipeline),
rl_0, rl_1 and rl_2 (stages in the pipeline), and
rl_drain (draining the pipeline)
all can fire and will fire only every other cycle

Waveforms: using MyFIFOF_CReg.bsv

Picture file: Waves_CReg.tiff



rl_gen (feeding the pipeline),
rl_0, rl_1 and rl_2 (stages in the pipeline), and
rl_drain (draining the pipeline)
all can fire and will fire on other cycle

Suggested exercises

- Note: the mkMyFIFOF module already exists in the BSV library, and is called mkPipelineFIFOF (see Lec_Regs_and_RWires, and also Section C.2.2 in the Reference Guide).

- When you use MyFIFOF_reg.bsv:
  - What are the scheduling constraints between the “put” and “get” methods of the shifter?
  - What is the longest combinational path in the circuit?

- When you use MyFIFOF_creg.bsv:
  - What are the scheduling constraints between the “put” and “get” methods of the shifter?
  - What is the longest combinational path in the circuit?

- Lec_CRegs_and_RWires describes another concurrent FIFOF: mkBypassFIFOF.
  - What is the behavior of the program if you use this FIFO in the Shifter instead?
  - What are the scheduling constraints between the “put” and “get” methods of the shifter?
  - What is the longest combinational path in the circuit?

Summary

- CRegs enable a tighter scheduling of rules into clocks, i.e., greater concurrency.
- By replacing Regs with CRegs you can fine-tune any micro-architecture to have any desired concurrency
  - With experience, one often pro-actively uses CRegs, anticipating the need for a certain level of concurrency.
- CRegs are semantically "clean" in that they will work with any schedule (if different ports are not used concurrently, it behaves like an ordinary register).

End

```
import FIFO::*;

typedef Bit#(32)  DataT;

module ex_in1_out2_bs(Empty);

  Integer  fifo_depth = 16;

  function Bit#(1)   determine_queue(DataT val );
   return (val[0]);
  endfunction

  FIFO#(DataT) inbound1();
  mkSizedFIFO#(fifo_depth) the_inbound1(inbound1);
  FIFO#(DataT) outbound1();
  mkSizedFIFO#(fifo_depth) the_outbound1(outbound1);
  FIFO#(DataT) outbound2();
  mkSizedFIFO#(fifo_depth) the_outbound2(outbound2);

  rule enq1 (True);
     DataT in_data  = inbound1.first;
     FIFO#(DataT) out_queue   =
         determine_queue(in_data    ) == 0 ? outbound1 : outbound2;
     out_queue.enq(in_data    );
    inbound1.deq;
  endrule : enq1

endmodule  : ex_in1_out2_bs
```