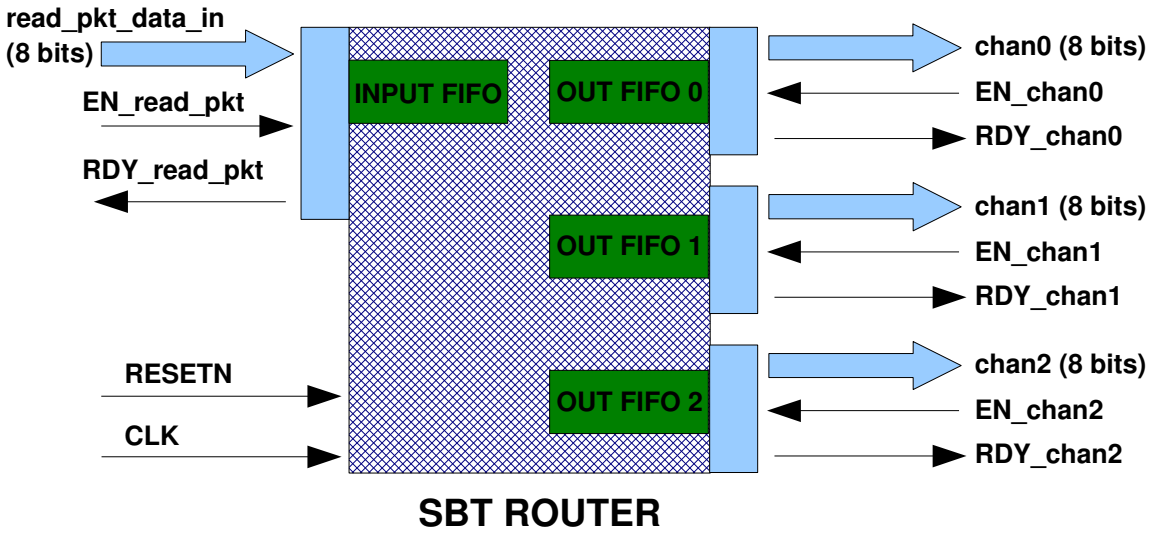
Small Router Design Using Bluespec SystemVerilog

Daian-Sova M. Filip
Master AAC

1. SBT Router Block Description

The SBT Router is a simple router which has one input port (8 bits wide) and three output ports (also 8 bits wide). Its main functionality is to receive input data on one channel and route it to one of the three output channels.



1.1 Data Packets

Input data is organized in 8 bit packets, with the first two bits being the address bit, and the other six being the payload (useful data). Since there are only 3 output lanes (chan0, chan1 and chan2) and the address is 2 bits wide, address 3 is illegal and packets sent to this address are discarded (ignored).

PAYLOAD	ADDR
7 2	1 ... 0

SBT PACKET

1.2 Input Protocol

The router has a FIFO to hold the input data (8 bits wide and two bytes deep). Data is driven on the input bus (read_pkt_data_in) as long as RDY_read_pkt signal is asserted. If this signal is deasserted by the router, this means the FIFO is full and no more data can be put into the FIFO. When data is driven on the input bus the EN_read_pkt signal is asserted by the data driver to let the router know new data is available on the input bus.

1.3 Output Protocol

When the router receives data, it routes it to the proper lane (chan0, chan1 or chan2). When it places new data on one of the output lanes, it asserts RDY_chanX signal to let the receiver know that new data is available on the corresponding lane. When the data is sampled by the receiver, the EN_chanX signal is asserted by the receiver to let the router know that data has been sampled. When the data is sampled, the router deletes the data from the FIFO and places the next item from the FIFO on the proper chanX lane.

2. Implementation using Bluespec SystemVerilog (BSV)

2.1. Interface Implementation

The BST Router is implemented in Bluespec SystemVerilog (BSV). The interface of the SBT Router is implemented as a series of methods: read_pkt as input, and chan0, chan1 and chan2 as outputs.

```
interface SBT_Router_Interface;
   // data input (8 bits, 1 channel)
   method Action read_pkt (Byte data_in);

   // data output (8 bits, 3 channels)
   method ActionValue#(Byte) chan0;
   method ActionValue#(Byte) chan1;
   method ActionValue#(Byte) chan2;
endinterface
```

The input interface is an Action method – it has an enable signal (EN_) which causes the related actions to take place in the clock cycle when the EN_ signal is high, it has a ready RDY_ signal that signals to the data driver when data can be stored into the FIFO, and it has an 8-bit argument which is the 8-bit input data bus.

The output interface is comprised of three ActionValue methods chanX, one for each output channel. Since they are Action methods they have an EN_ enable signal, an RDY_ ready signal, and they return an 8-bit value which is the 8-bit output data bus.

2.2. Rules

The logic of the SBT Router is implemented using rules, which call the interface methods.

```
   rule route_pkt;
      let first_elem = input_fifo.first;
      let addr = first_elem[1:0];

      if (addr == 2'b00) chan0_fifo.enq(input_fifo.first);
      if (addr == 2'b01) chan1_fifo.enq(input_fifo.first);
      if (addr == 2'b10) chan2_fifo.enq(input_fifo.first);

      if (addr != 2'b11)
         $display ("*%d* [SBT] Forwarding to CHANNEL:%h - DATA:%h",
                  clock, addr, first_elem[7:2]);

      else
         $display ("*%d* [SBT] Discarding packet with bad address -
                  DATA:%h | ADDR:%h", clock, first_elem[7:2], addr);

      input_fifo.deq;
   endrule
```

The route_pkt rule implements the forwarding of data from the input FIFO to one of the output FIFOs, if the address is valid (0, 1 or 2).

2.3. Simulating

The SBT Router is instantiated in a top-level module mkTOP. This module acts as driver (sends data to the router) and as receiver (takes data from the router). This behavior is implemented using rules.

```
rule send_data;
   $display ("*%d* [TOP I] Sending DATA:%h | ADDR:%h", clock, data[7:2],
               data[1:0]);
   router.read_pkt (data);
   router.packet_valid(1);

   if (data[1:0] != 2'b11) count <= count + 1;

   data <= data + 5;
endrule

rule read_chan0;
   Byte a <- router.chan0;
   count <= count - 1;

   $display ("*%d* [TOP 0] Chan0 receiving DATA:%h | ADDR:%h", clock, a[7:2],
               a[1:0]);
endrule
```

The send_data rule prepares a value data to be sent to the router and on every cycle when it can send (the RDY_ signal of the read_pkt method of the router is asserted) it places the data on the input bus.

The read_chanX rule reads a value from the corresponding channel of the router when the RDY_ signal of the chanX method is asserted.

These rules will only fire is the their implicit conditions are met. These RDY_ signals are part of the implicit conditions of these rules because the RDY_ signals show when the intended actions can be performed.

3. BSV Code

3.1. SBT_Router.bsv

```
// File:        SBT_Router.bsv

// Description: contains BSV code for the SBT Router

package SBT_Router;

import FIFO :: *;

typedef bit[7:0] Byte;


interface SBT_Router_Interface;
   // data input (8 bits, 1 channel)
   method Action read_pkt (Byte data_in);

   // data output (8 bits, 3 channels)
   method ActionValue#(Byte) chan0;
   method ActionValue#(Byte) chan1;
   method ActionValue#(Byte) chan2;
endinterface


(* synthesize *)
module mkSBT_Router (SBT_Router_Interface);

   // FIFO definitions
   FIFO#(Byte) input_fifo <- mkFIFO;
   FIFO#(Byte) chan0_fifo <- mkFIFO;
   FIFO#(Byte) chan1_fifo <- mkFIFO;
   FIFO#(Byte) chan2_fifo <- mkFIFO;

   Reg#(Byte)  clock      <- mkReg(1);
   Reg#(Byte)  input_byte <- mkReg(0);

   rule incr_clock;
      clock <= clock + 1;
   endrule

   rule route_pkt;
      let first_elem = input_fifo.first;
      let addr = first_elem[1:0];

      if (addr == 2'b00) chan0_fifo.enq(input_fifo.first);
      if (addr == 2'b01) chan1_fifo.enq(input_fifo.first);
      if (addr == 2'b10) chan2_fifo.enq(input_fifo.first);

      if (addr != 2'b11)
         $display ("*%d* [SBT] Forwarding to CHANNEL:%h - DATA:%h", clock, addr,
                   first_elem[7:2]);
      else
```

```
         $display ("*%d* [SBT] Discarding packet with bad address - DATA:%h |
                  ADDR:%h", clock, first_elem[7:2], addr);

      input_fifo.deq;
   endrule

   method Action read_pkt (data_in);
      input_fifo.enq (data_in);
   endmethod

   method ActionValue#(Byte) chan0;
      let first_elem = chan0_fifo.first;
      chan0_fifo.deq;

      return first_elem;
   endmethod

   method ActionValue#(Byte) chan1;
      let first_elem = chan1_fifo.first;
      chan1_fifo.deq;

      return first_elem;
   endmethod

   method ActionValue#(Byte) chan2;
      let first_elem = chan2_fifo.first;
      chan2_fifo.deq;

      return first_elem;
   endmethod

endmodule

endpackage
```

3.2. SBT_Top.bsv

```
// File:        SBT_top

// Description: contains top module to simulate the SBT_Router

import SBT_Router :: *;

(* synthesize *)
module mkTop (Empty);

   SBT_Router_Interface router <- mkSBT_Router;
   Reg#(Byte) data  <- mkReg(0);
   Reg#(Byte) count <- mkReg(0);
   Reg#(Byte) clock <- mkReg(1);

   rule send_data;
      $display ("*%d* [TOP I] Sending DATA:%h | ADDR:%h", clock, data[7:2],
                  data[1:0]);
      router.read_pkt (data);
      router.packet_valid(1);

      if (data[1:0] != 2'b11) count <= count + 1;

      data <= data + 5;
   endrule

   rule read_chan0;
      Byte a <- router.chan0;
      count <= count - 1;

      $display ("*%d* [TOP 0] Chan0 receiving DATA:%h | ADDR:%h", clock, a[7:2],
                  a[1:0]);
   endrule

   rule read_chan1;
      Byte a <- router.chan1;
      count <= count - 1;

      $display ("*%d* [TOP 1] Chan1 receiving packet DATA:%h | ADDR:%h", clock,
                  a[7:2], a[1:0]);
   endrule

   rule read_chan2;
      Byte a <- router.chan2;
      count <= count - 1;

      $display ("*%d* [TOP 2] Chan2 receiving packet DATA:%h | ADDR:%h", clock,
                  a[7:2], a[1:0]);
   endrule

   rule incr_clk;
      clock <= clock + 1;
   endrule
```

```
   rule end_simulation;
      if (clock >= 20)
         if (count == 0) $finish(0);
   endrule

endmodule
```