Chiselizing SDR Blocks

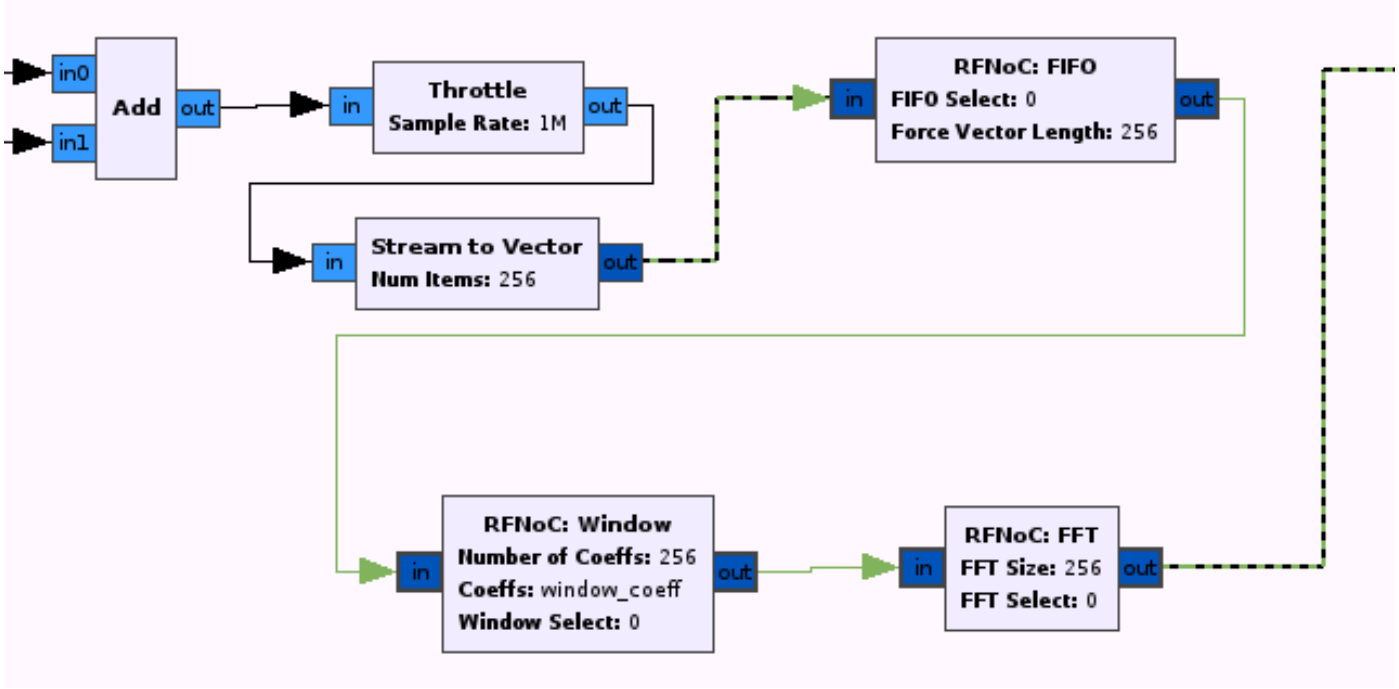
Albert Magyar

March 18, 2015

Overview

- Intro to RFNoC
- RFNoC Blocks
- Example: `addsub`
- Recurring patterns in Verilog and Chisel
- Goals

Intro to RFNoC



- FPGA toolkit for GNU radio
- Allows chaining of hardware and software blocks

Intro to RFNoC

- Accelerators implemented in Verilog
- NoC: packet headers on top of AXI4 Streaming crossbar
- Designers write parametrized blocks with ready/valid/last and settings registers

Intro to RFNoC

- Designers write parametrized blocks
- GNU Radio frontend elaborates instances
- RFNoC configurator assigns endpoint addresses and parametrizes RFNoC shim modules (`noc_shell`)
- Parametrizes and generates Verilog for NoC

RFNoC Blocks

- Simple, fixed-function DSP blocks
- Blocks have simple FIFO interfaces
- Export parameters to the GNU Radio space
- Identity as RFNoC endpoint transparent to designer

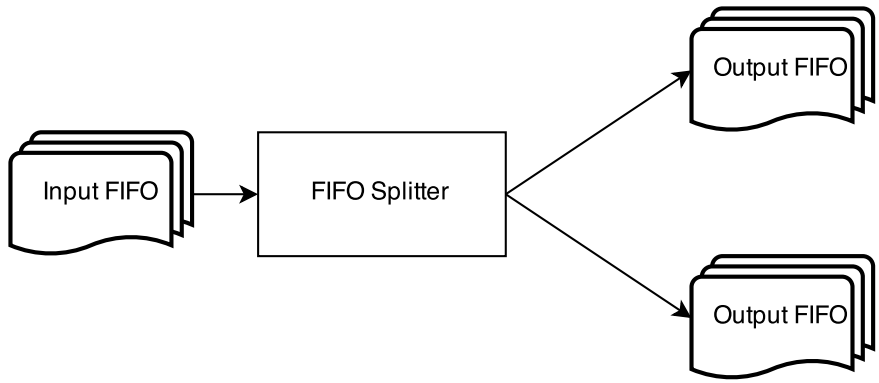
RFNoC Blocks

- Simple, fixed-function DSP blocks
- Blocks have simple FIFO interfaces
- Export parameters to the GNU Radio space
- Identity as RFNoC endpoint transparent to designer
- *We want people to write these in Chisel*

Example: addsub

- Simple variable-width complex adder/subtractor
- FIFO inputs

Patterns: splitting FIFOs



- Hard to do right
- Don't let ready depend on valid

Verilog

```
generate
if(ACTIVE_MASK[0])
axi_fifo_short #(.WIDTH(WIDTH+1)) short_fifo0
  (.clk(clk), .reset(reset), .clear(clear),
   .i_tdata({o0_tlast_int, o0_tdata_int}), .i_tvalid(o0_tvalid_int), .i_tready(o0_tready_int),
   .o_tdata({o0_tlast, o0_tdata}), .o_tvalid(o0_tvalid), .o_tready(o0_tready));
if(ACTIVE_MASK[1])
axi_fifo_short #(.WIDTH(WIDTH+1)) short_fifo1
  (.clk(clk), .reset(reset), .clear(clear),
   .i_tdata({o1_tlast_int, o1_tdata_int}), .i_tvalid(o1_tvalid_int), .i_tready(o1_tready_int),
   .o_tdata({o1_tlast, o1_tdata}), .o_tvalid(o1_tvalid), .o_tready(o1_tready));
if(ACTIVE_MASK[2])
axi_fifo_short #(.WIDTH(WIDTH+1)) short_fifo2
  (.clk(clk), .reset(reset), .clear(clear),
   .i_tdata({o2_tlast_int, o2_tdata_int}), .i_tvalid(o2_tvalid_int), .i_tready(o2_tready_int),
   .o_tdata({o2_tlast, o2_tdata}), .o_tvalid(o2_tvalid), .o_tready(o2_tready));
if(ACTIVE_MASK[3])
axi_fifo_short #(.WIDTH(WIDTH+1)) short_fifo3
  (.clk(clk), .reset(reset), .clear(clear),
   .i_tdata({o3_tlast_int, o3_tdata_int}), .i_tvalid(o3_tvalid_int), .i_tready(o3_tready_int),
      .o_tdata({o3_tlast, o3_tdata}), .o_tvalid(o3_tvalid), .o_tready(o3_tready));
endgenerate
```

Problems

- Classic generate block
- No indirection of signals

Module-level reuse

```
module split_stream_fifo
  #(parameter WIDTH=16,
    parameter ACTIVE_MASK=4'b1111)
   (input clk, input reset, input clear,
    input [WIDTH-1:0] i_tdata, input i_tlast, input i_tvalid, output i_tready,
    output [WIDTH-1:0] o0_tdata, output o0_tlast, output o0_tvalid, input o0_tready,
    output [WIDTH-1:0] o1_tdata, output o1_tlast, output o1_tvalid, input o1_tready,
    output [WIDTH-1:0] o2_tdata, output o2_tlast, output o2_tvalid, input o2_tready,
    output [WIDTH-1:0] o3_tdata, output o3_tlast, output o3_tvalid, input o3_tready);
```

Problems

- Module-level reuse seems nice
- Does it make sense for hardware?

Problems

- Module-level reuse seems nice
- Does it make sense for hardware?
- RTL designers want to reuse implementations
- RTL-level designs are tightly coupled
- Interface-level inheritance often gets in the way

Module-level reuse

```
module split_stream_fifo
  #(parameter WIDTH=16,
    parameter ACTIVE_MASK=4'b1111)
   (input clk, input reset, input clear,
    input [WIDTH-1:0] i_tdata, input i_tlast, input i_tvalid, output i_tready,
    output [WIDTH-1:0] o0_tdata, output o0_tlast, output o0_tvalid, input o0_tready,
    output [WIDTH-1:0] o1_tdata, output o1_tlast, output o1_tvalid, input o1_tready,
    output [WIDTH-1:0] o2_tdata, output o2_tlast, output o2_tvalid, input o2_tready,
    output [WIDTH-1:0] o3_tdata, output o3_tlast, output o3_tvalid, input o3_tready);
```

Compounded by fixed portlists in Verilog!

Chisel: flexible implementation reuse

```
object SplitDecoupled {
  def apply[T <: Data](in: DecoupledIO[T], n: Int): Seq[DecoupledIO[T]] = {
    val outs = Seq.fill(n){ in.clone }
    var iready = Bool(true)
    for (i <- 0 until n) {
      iready = iready && outs(i).ready
      var ovalid = in.valid
      for (j <- 0 until n) {
        if (i != j) ovalid = ovalid && outs(j).ready
      }
      outs(i).valid := ovalid
      outs(i).bits := in.bits
    }
    in.ready := iready
    outs
  }
}
```