Contents

1 Introduction

The StreamLoopback128 sample project demonstrates how to use the streaming data flow model to communicate with your firmware in the FPGA. The Pico API docs explain the concept and motivation for streams. You may wish to keep a copy of those docs handy while you read this.

This sample sets up one stream going from the host to the FPGA, and another stream going from the FPGA back to the host. The logic keeps a running sum of the data received from the host. For each piece of input data, it generates an output consisting of the sum of all input data seen so far.

2 Firmware Project

This project uses two streams: one in and one out. We use the PicoDefines.v file to tell the Pico framework that we need these two streams. The framework then takes care of impelementing the support logic to connect these streams to the PCIe interface, which gives them access to the host computer or other FPGAs.

Once we have the two streams setup, we'll simply connect the incoming stream to the outgoing stream. In your application, of course, you'll want to transform the data before you send it back. This is the point at which you would insert your application logic.

2.1 Using the Stream Modules

Here's what our PicoDefines.v file looks like:

```
`define USER_MODULE_NAME StreamLoopback128

`define STREAM1_IN_WIDTH 128
`define STREAM1_OUT_WIDTH 128

// We define the type of FPGA and card we're using.
`define PICO_MODEL_M505
`define PICO_FPGA_LX325T

`include "BasePicoDefines.v"
```

You can see the relevant STREAM1_IN_WIDTH and STREAM1_OUT_WIDTH defines that specify both streams are 128b wide. The Pico framework sees these defines and connects the two streams to our module. Here's what the port list of our module looks like:

```
module StreamLoopback128 (
    input               clk,
    input               rst,

    input               s1i_valid,
    output              s1i_rdy,
    input [127:0]       s1i_data,

    output reg          s1o_valid,
    input               s1o_rdy,
    output [127:0]      s1o_data
    );
```

Note the common clock and reset signals, and the two stream interfaces.

When s1i_valid is high and we assert s1i_rdy, we'll grab the incoming data on s1i_data. We'll know our output has been accepted on s1o_data when we're asserting s1o_valid and we see s1o_rdy go high.

2.2 Application Logic

As we mentioned earlier, the "application logic" in this example sums the incoming data. It generates an output stream consisting of the sum of the input at each point in the input stream. For example, given an input stream of five values: 0, 1, 2, 3, 4, it will produce an output stream of five values: 0, 1, 3, 6, 10.

We sum the low 32 bits of the input stream rather than the whole 128. We report the last value received on the low 32 bits and use the high 64 bits of the output stream to inject a "tag." This tag is a just a constant value in this sample, but in practice it could be additional computed statistics for the input stream beyond just a sum. Here's how we construct the output data from our running sum and the tag data:

```
assign s1o_data = {32'h42424242, 32'hdeadbeef, sum[31:0], last_input[31:0]};
```

The datapath in this module computes the sum and stores the input value when we receive new data from the input stream. Note that the datapath does not determine when it's ready to receive new input, but instead it only listens to when the controller portion of this module says that it should accept new input (see next section). We keep track of the sum in the "sum" register, and we remember the input that we received on the previous cycle in the "last_input" register. This introduces a single-cycle pipeline in the design.

```
always @(posedge clk) begin
    if (rst) begin
        sum         <= 0;
    end else if (s1i_valid && s1i_rdy) begin
        sum         <= sum + s1i_data[31:0];
        last_input  <= s1i_data[31:0];
    end
end
```

We control the flow of data in this system using the following logic. We use the "s1o_valid" register to track whether the pipeline contains valid data or a bubble. For example, while we're receiving input (s1i_valid==1) the pipeline stays full, but when s1i_valid goes low, a bubble gets into the pipeline. When this happens, "s1o_valid" will get set to 0, and we will not have any valid data ready to pass to the output. When s1i_valid goes high again, the pipeline runs full.

```
always @(posedge clk) begin
    if (rst) begin
        s1o_valid   <= 0;
    end else if (s1i_rdy) begin
        s1o_valid   <= s1i_valid;
    end
end

assign s1i_rdy = ~s1o_valid | s1o_rdy;
```

3 Software

The software portion of this sample sends a sequence of data to the FPGA and reads the results back.

First, we load the bit file into the FPGA and obtain a handle using the RunBitFile function, taking the file name from the command line:

```
// specify the .bit file name on the command line
if (argc < 2) {
    fprintf(stderr, "Please specify the .bit file on the command line.\n"
                    "For example: pbc ../firmware/
                        M501_LX240_StreamLoopback128.bit\n");
    exit(1);
}
bitFileName = argv[1];

printf("Loading FPGA with '%s' ...\n", bitFileName);
err = RunBitFile(bitFileName, &pico);
if (err < 0) {
    fprintf(stderr, "RunBitFile error: %s\n", PicoErrors_FullError(err, ibuf
        , sizeof(ibuf)));
    exit(1);
}
```

Next, we use this handle to open the stream to communicate with the FPGA:

```
// data goes out to the firmware on stream #1 and also comes back on stream
    #1
printf("Opening streams to test counter\n");
stream = pico->CreateStream(1);
if (stream < 0) {
    fprintf(stderr, "CreateStream error: %s\n", PicoErrors_FullError(stream,
         ibuf, sizeof(ibuf)));
    exit(1);
}
```

Now we fill our buffer with an increasing sequence, and send it to the card. Before we send it, we verify that the card is ready to handle the data.

```
// fill the buffer with data we'll recognize when it comes back.
for (i=0; i < sizeof(buf)/sizeof(buf[0]); ++i)
    buf[i] = 0x42000000 | i;

// In order to know how much data we can send right now, we first call
    GetBytesAvailable.
printf("%i bytes of room in stream to firmware.\n", i=pico->
    GetBytesAvailable(stream, false /* writing */));
if (i < 0){
    fprintf(stderr, "GetBytesAvailable error: %s\n", PicoErrors_FullError(i,
        ibuf, sizeof(ibuf)));
    exit(1);
}

// we will cap our call to WriteStream at 256 words of 16 B each (4kB total)
    .
if (i > 256*16)
    i = 256*16;

// However, all calls to WriteStream must be 16B aligned (even for 4B wide
    streams)
i &= ~0xf;

// This writes i bytes of data from our buffer (buf) to the stream specified
    by our stream handle (e.g. 'stream')
printf("Writing %i B\n", i);
err = pico->WriteStream(stream, buf, i);
if (err < 0) {
    fprintf(stderr, "WriteStream error: %s\n", PicoErrors_FullError(err,
        ibuf, sizeof(ibuf)));
    exit(1);
}
```

At this point the data has been sent to the FPGA, and we can ask the system to read back the returned data from the FPGA.

```
printf("%i B available to read from firmware.\n", i=pico->GetBytesAvailable(
    stream, true /* reading */));
if (i < 0){
    fprintf(stderr, "GetBytesAvailable error: %s\n", PicoErrors_FullError(i,
        ibuf, sizeof(ibuf)));
    exit(1);
}

// In this case, we choose to only read 512B out of the 4kB that should be
    available
i = 32*16;

// Again, all our streams must operate upon 16B words
i &= ~0xf;

// This reads i bytes of data from the output stream specified by our stream
    handle (e.g. 'stream')
```

```
// into our software buffer (buf)
printf("Reading %i B\n", i);
err = pico->ReadStream(stream, buf, i);
if (err < 0) {
    fprintf(stderr, "ReadStream error: %s\n", PicoErrors_FullError(err, ibuf
        , sizeof(ibuf)));
    exit(1);
}

// Now that we have received all of our read data back from the firmware, we
     print it out for the user
printf("Data received back from firmware:\n");
print128(stdout, buf, i/16);
```

That's it! You can look at the values printed out and see the running sum the FPGA computed.