MultiStream Sample Documentation

February 6, 2013

Contents

1 Overview **1**

2 Firmware Project **1**

3 Application Logic **2**

4 Firmware Simulation **3**

5 Software **5**

6 Variables **8**

7 Sample Output **8**

1 Overview

The purpose of this sample is to demonstrate the proper way for a user to interact with multiple streams on multiple FPGAs both from the firmware and from the host software. This sample shows the user how to write and read from software via the streaming model using multiple threads, and also how to write from software using the Picobus model.

The overall goal of this sample is to connect multiple FPGAs together to build a prime sieve. Each FPGA will have a divisor value stored on it. We will stream sequential values to the first FPGA in the chain, and the last FPGA will be left with values that are not divisible by any of the divisors along the chain. For each FPGA, we set up one stream going from the host to the FPGA, and two streams going from the FPGA back to the host. The application logic will handle reading data from the host and sending it back to one of these two streams. If a value is not divisible by the divisor stored on the FPGA, then it will be sent back on the first output stream, otherwise it will be sent to the second output stream. Then in our software we can connect the first output stream of an FPGA to the input stream of the next FPGA in the chain to construct our sieve.

This document begins by describing the logic implemented on the FPGA firmware to write and read from multiple streams. Next, it explains the testbench and how it tests the firmware. Then

it describes the test software, which is similar to the firmware testbench but is also able to handle multiple FPGA's chained together. It also operates on each stream in parallel using the pthreads library.

2 Firmware Project

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

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

```
// PicoDefines.v - here we configure the base firmware for our project

// This includes a placeholder "user" module that you will replace with
// your code. To use your own module, just change the name from
// PicoBus128_counter to your module's name, and then add the file to
// your ISE project.

`define USER_MODULE_NAME MultiStream

// Stream 1 into the firmware will hold all sequential values
`define STREAM1_IN_WIDTH 128
// Stream 1 out of the FPGA will hold non-divisible values
`define STREAM1_OUT_WIDTH 128
// Stream 2 out of the FPGA will hold divisible values
`define STREAM2_OUT_WIDTH 128
// Divisor value stored on the PicoBus
`define PICOBUS_WIDTH 32

// We define the type of FPGA and card we're using.
`define PICO_MODEL_M503
`define PICO_FPGA_LX240T

`include "BasePicoDefines.v"
```

You can see that all the streams are 128b wide, and the PicoBus is 32b wide. The Pico framework sees these defines and connects the three streams and the PicoBus to our application module in MultiStream.v. For more information about the PicoBus and Stream models, see the PicoAPI documentation.

Once we have the three streams setup, we'll simply connect the incoming stream to one of the outgoing streams. In your application you can change the application logic to manipulate the data however you like.

3 Application Logic

The application in this example first reads a value into the "divisor" register. Then it checks the divisibility of each input value to determine on which of the two output streams to write the value to. For example, given an input stream of five values: 0, 1, 2, 3, 4, and a divisor value of 2, it will

produce one output stream with three even values 0, 2, 4 and another output stream with the two odd values 1, 3.

The first piece of logic handles reading the divisor in from the PicoBus. If PicoWr is asserted, we store the value of the divisor from the PicoBus into a register. For more information about using the PicoBus, see the PicoAPI documentation or the PicoBus HelloWorld example.

```
// read the divisor into PicoAddr
always @(posedge PicoClk) begin
    if (PicoRst) begin
        div_pico <= 0;
    end else begin
        if ( PicoWr && PicoAddr[31:0] == 32'h00 )
            div_pico <= PicoDataIn;
    end
end
```

The main piece of logic in this module controls the input and output streams. It consumes the next input as long as the output stream is able to accept data. We keep track of whether our input data is divisible by the divisor or not in the "remainder" register. We use the "empty" register to track whether the pipeline contains valid data or a bubble. For example, while we're receiving input the pipeline stays full, but when s1i_valid goes low, a bubble gets into the pipeline and we set "empty" true. When s1i_valid goes high again, "empty" goes low and the pipeline runs full.

```
always @ (posedge clk) begin
    if (rst) begin
        empty <= 1;
        remainder <= 0;
    end else begin
        // If getting new input, we will not be empty.
        // If outputting a value and not getting new input, we will be empty
        if (s1i_valid)
            empty <= 0;
        else if ((s1o_rdy & s1o_valid) | (s2o_valid & s2o_rdy))
            empty <= 1;
        // If there's more input data for us (s1i_valid) and
        // we've decided we're ready for it (s1i_rdy), then
        // calculate the remainder from the division
        if (s1i_rdy && s1i_valid) begin
            remainder <= remainder_wire;
        end
    end
end
```

For the purpose of this sample, we mask only the 4 low order bits of the input data to compute the division.

```
// calculate the remainder
assign remainder_wire =
s1i_data[NUM_BITS-1:0] - ((s1i_data[NUM_BITS-1:0]/divisor)*divisor);
```

We also have some combinational logic for our s1o_valid, s2o_valid and s1i_rdy signals. Since we only want the value from the input stream to be written to one of the output streams, we create our logic so that only either the s1o_valid or s2o_valid signals will be asserted based on the remainder.

```
// Stream 1 valid if value is not divisible and not empty
assign s1o_valid = ~empty & ~(remainder == 0);

// Stream 2 valid if value is divisible and not empty
assign s2o_valid = ~empty & (remainder == 0);

// Input  is ready when we are empty or either output is valid and ready
assign s1i_rdy = empty | (s1o_valid & s1o_rdy) | (s2o_valid & s2o_rdy);
```

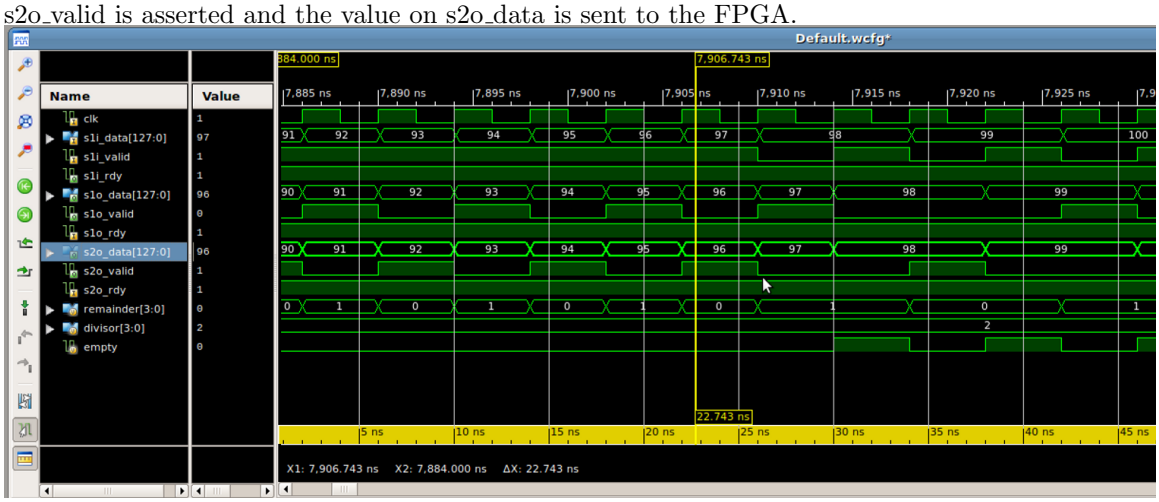
4 Firmware Simulation

The test module that has been implemented, which is called PicoTestbench.v, is designed to stress the reading and writing capability of the multiple stream system on the FPGA. The test module mirrors most of the function of the software test, with a few exceptions. The simulation only sends a small amount of data to avoid a long simulation time.

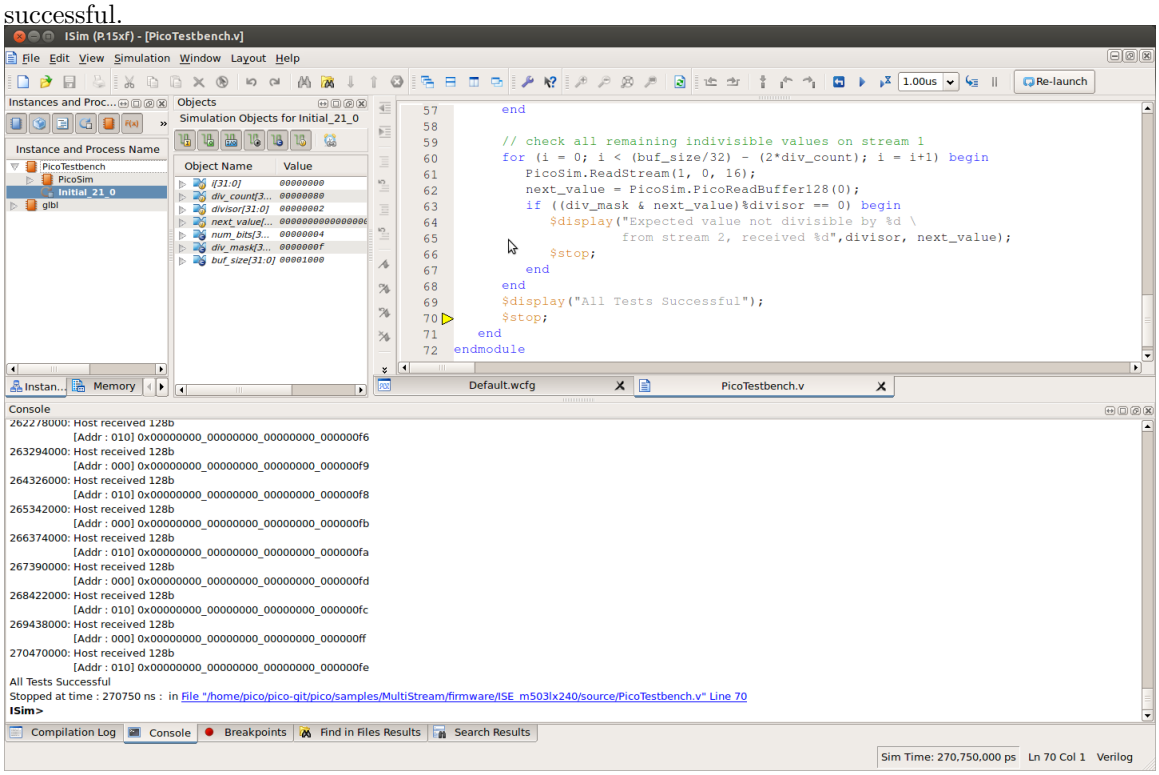
The first step in the simulation is to write a divisor value to the FPGA. Next, sequential values are sent to the input stream. The values are written to a 4kB buffer, and then written to the FPGA using the WriteStream function.

```
div_count = 0;
// stream 4kB of data to the memory at address 0
for (i = 0; i < buf_size; i=i+16) begin
    // calculate next sequential value
    next_value = i/16;
    // count number of divisible values to check later
    if (next_value % divisor == 0)
        div_count = div_count+1;
    // write next_value to next space on the buffer
    PicoSim.PicoLoadBuffer128(i, next_value );
end
// Write entire block of 4kb to input stream
PicoSim.WriteStream(1,0,buf_size);
```

Here is what the waveform looks like of the module processing the data. When the remainder is 0, s2o_valid is asserted and the value on s2o_data is sent to the FPGA.



Next the simulation reads back the write data in order to verify that the write ended up on the correct stream. The first output stream should contain all the values that are not divisible by the divisor, and the second output stream should contain all the values that are divisible. The simulation reads one value from the divisible stream and one from the non-divisible stream until there are no more on the divisible stream. If the divisor is greater than 2, then there will be more values left on the non-divisible stream, which are now read and verified. If all of the data written to the input stream ends up on the correct output stream, the module will finish and print out that the test was successful.



5 Software

The software portion of this sample writes a sequence of data to the FPGA and reads the results back. Each stream in and out of the FPGA is run in parallel using threads in software. By default, the software only runs on one FPGA and has 3 threads running concurrently (one for the input stream, two for the output streams), but it is able to chain multiple FPGA's together with the PicoDrv function TieStreams. With multiple FPGAs we connect the first output stream from the first FPGA directly to the input stream of the next FPGA. In this way we can load different divisor values into each FPGA in our chain and the output stream of the last FPGA will hold values that are not divisible by any of the divisors along the chain.

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. If multiple FPGA's are being used, the software will try to load all of the FPGA's with the same bit file.

```
err = RunBitFile(bitFileName, &cur_pico);
```

Next, as in the firmware testbench, we write the value of our divisor to the FPGA at address 0 using the PicoBus function WriteDeviceAbsolute.

```
if((err = cur_pico->WriteDeviceAbsolute(0, buf, 16)) < 0) {
    fprintf(stderr, "Error Writing: %d\n", err);
    exit(1);
}
```

After loading divisors via the PicoBus, we open two streams to the FPGA, and store their handles so we can identify which stream to write to or read from later. The CreateStream method takes a stream number and returns a stream handle.

```
// Open stream 1, used for input and output
handle1 = cur_pico->CreateStream(1);
if (handle1 < 0) {
    fprintf(stderr, "couldn't open stream 1! (return code: %i)\n", handle1);
    exit(1);
}

// Open stream 2, used for output only
handle2 = cur_pico->CreateStream(2);
if (handle2 < 0) {
    fprintf(stderr, "couldn't open stream 2! (return code: %i)\n", handle2);
    exit(1);
}
```

If we are using multiple FPGA cards, we connect the first (non-divisible) stream from the previous FPGA in the chain to the input stream of the current FPGA using the TieStreams method. We make a call to TieStreams from the previous PicoDrv object in the chain and we pass it the stream IDs (not handles) of the two streams we want to connect. After this call, stream 1 out of last_pico will be connected directly to stream 1 into cur_pico.

```
if (last_pico != NULL) {
    err = last_pico->TieStreams(1,cur_pico,1);
```

We will have a total of two more threads than the number of FPGAs in the system. One thread writes a block of sequential values to the input stream on the first FPGA. For every FPGA in the system, we have one thread reading data from the "divisible" output stream. There is also one thread which reads off the "non-divisible" output stream of the last FPGA. These are the values that were not divisible by any of the divisor values assigned to the FPGAs.

```
// Start all the input and output threads
for (card = 0; card < num_cards; card++) {
    pthread_create(&threads[card], NULL, start_stream, (void*) &info[card]);
}
pthread_create(&threads[num_cards],NULL, start_stream, (void*) &input_info);
pthread_create(&threads[num_cards+1], NULL, start_stream, (void*)  &
    output_info);

// start a new thread
void* start_stream(void* arg) {
    StreamInfo* info = (StreamInfo*) arg;
    if (info->is_input) {
        printf("input stream thread started with stream handle %d\n",info->
            stream_handle);
```

```
        stream_input(info->pico, info->stream_handle);
    } else {
        printf("output stream thread started with stream handle %d\n",info->
            stream_handle);
        stream_output(info->pico, info->stream_handle, info->divisor, info->
            divisible);
    }
    return 0;
}
```

The stream_input and stream_output functions that are called by start_stream work similarly to our firmware testbench explained earlier. The stream_input function writes blocks of sequential values to the stream using the PicoDrv::WriteStream method. When it has written all the data, it asserts a flag that lets the output stream threads know that no more data will be written.

The stream_output function checks if there is data on the output stream to read by the PicoDrv::GetBytesAvailable method. Then it reads values from the output stream using the PicoDrv::ReadStream method, and checks that the values were written to the correct stream. The output stream threads terminate when the total number of bytes read reaches the total number of bytes written and when the input stream thread has finished writing. Since we are using a shared variable to keep track of the total amount of data read among all output stream threads, we need to use a mutex to ensure that only one thread is accessing it at a time. Here is what the main loop in stream_output looks like:

```
// function called by the output thread which reads numbers from the output
    stream until
// input stream is done writing
while ((total_read < total_write) || !done_writing)  {
    read_size = pico->GetBytesAvailable(stream_handle, true);
    // only read as much room as we have in buf
    if (read_size > sizeof(buf)) {
        read_size = sizeof(buf);
    }
    err = pico->ReadStream(stream_handle, buf, read_size);
    if (err != read_size) {
        fprintf(stderr, "read error. returned %i, but should have been %d\n"
            , err, read_size);
        exit(1);
    }
    // make sure all of the integers in buf were on the correct stream
    for (i = 0; i < read_size/4; i+=4) {
        if (((buf[i] & DIV_MASK) % divisor == 0) != divisible) {
            printf("unexpected value %d found\n",buf[i]);
            exit(1);
        }
    }
    // update total_read to reflect the read that has been completed
    pthread_mutex_lock(&mutex);
    total_read += err;
    pthread_mutex_unlock(&mutex);
    // print out final "prime" values off non divisible stream
    if (!divisible) {
        print128(stdout, buf, read_size/16);
    }
}
```

Once all the data comes through the streams, the program joins all the threads and exits.

```
// Join all threads and terminate
for (i = 0; i < num_cards+2; i++) {
    pthread_join(threads[i], NULL);
}
printf("All tests successful!\n");
return 0;
```

For more information about communicating with streams in software, you can look at the PicoAPI documentation, the StreamLoopback sample project, and the MultiStream.cpp source file for this project.

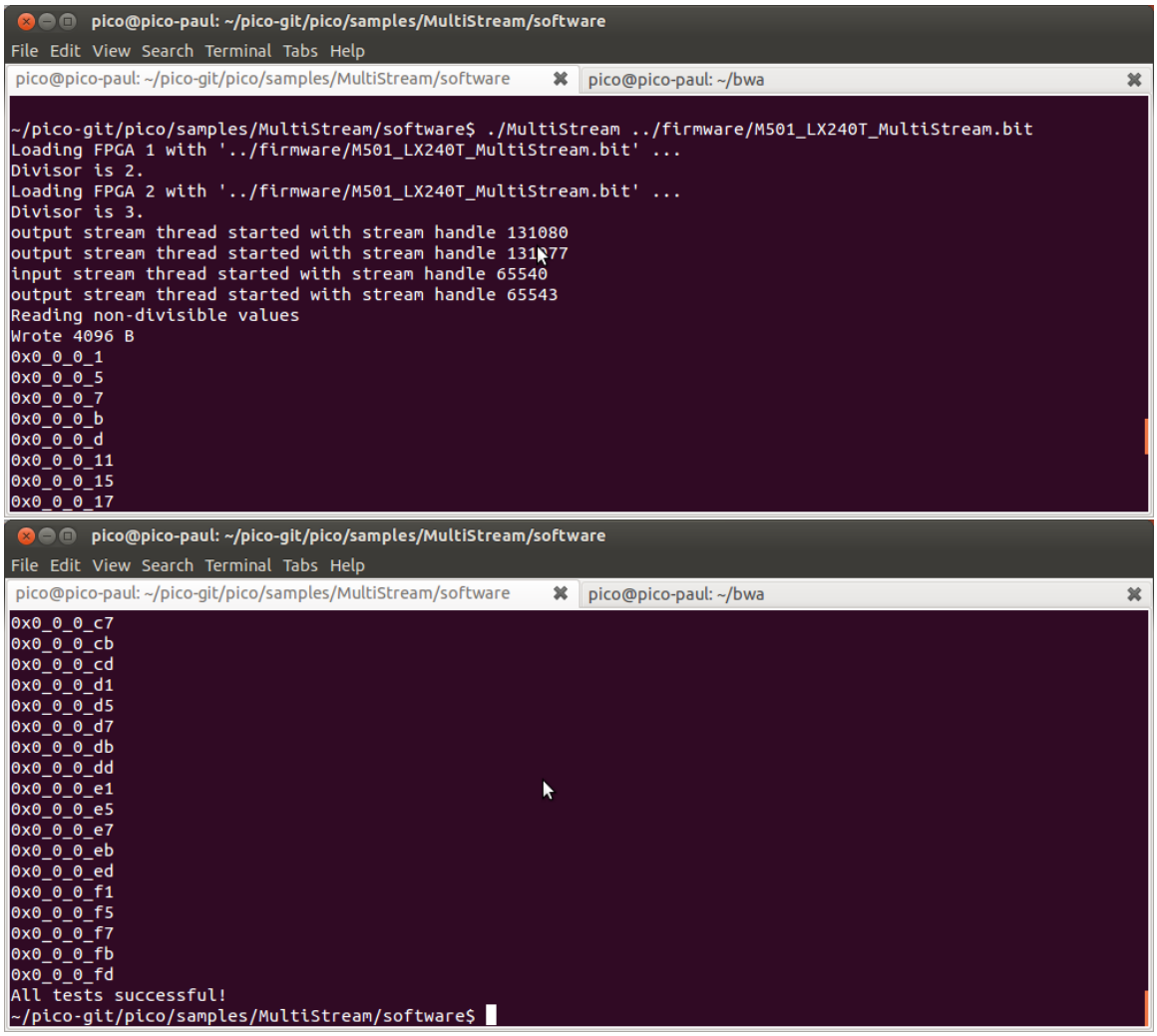
6 Variables

In our software program, there are few values that can be set by the user to experiment with different running modes of the system. These are defined in MultiStream.h, and can be changed by the user. NUM_CARDS lets us define how many FPGAs to chain together in our system. You should only change this to more than one if you have multiple Pico FPGAs that will be able to load the same bitfile. The value NUM_DIV_BITS should be the same as the NUM_BITS in our application logic. NUM_BYTES lets us choose how many bytes of data will be written to the input stream. Each value in our software is 4B wide. NUM_BYTES must be set to at least 4096, and will be rounded down to the nearest block size that can be written to the FPGA.

```
// number of FPGAs
#define NUM_CARDS 2
// number of bits that the division can handle in firmware
#define NUM_DIV_BITS  4
// number of bytes of data written to input stream
#define NUM_BYTES  4096
// initial value for the divisor. If multiple FPGA's are used,
// divisor will increment by 1 for each FPGA in the chain
#define DIV 2
```

7 Sample Output

Here is what a successful software run will look like. This example has two M–503 FPGA modules, and the variables match what is shown above. The software will print out how many bytes have been written to the FPGA, and then print out every value that was not divisible by any divisor value along the FPGA chain.


pico@pico-paul: ~/pico-git/pico/samples/MultiStream/software
File Edit View Search Terminal Tabs Help
pico@pico-paul: ~/pico-git/pico/samples/MultiStream/software
pico@pico-paul: ~/bwa
~/pico-git/pico/samples/MultiStream/software$ ./MultiStream ../firmware/M501_LX240T_MultiStream.bit
Loading FPGA 1 with '../firmware/M501_LX240T_MultiStream.bit' ...
Divisor is 2.
Loading FPGA 2 with '../firmware/M501_LX240T_MultiStream.bit' ...
Divisor is 3.
output stream thread started with stream handle 131080
output stream thread started with stream handle 131077
input stream thread started with stream handle 65540
output stream thread started with stream handle 65543
Reading non-divisible values
Wrote 4096 B
0x0_0_0_1
0x0_0_0_5
0x0_0_0_7
0x0_0_0_b
0x0_0_0_d
0x0_0_0_11
0x0_0_0_15
0x0_0_0_17
pico@pico-paul: ~/pico-git/pico/samples/MultiStream/software
File Edit View Search Terminal Tabs Help
pico@pico-paul: ~/pico-git/pico/samples/MultiStream/software
pico@pico-paul: ~/bwa
0x0_0_0_c7
0x0_0_0_cb
0x0_0_0_cd
0x0_0_0_d1
0x0_0_0_d5
0x0_0_0_d7
0x0_0_0_db
0x0_0_0_dd
0x0_0_0_e1
0x0_0_0_e5
0x0_0_0_e7
0x0_0_0_eb
0x0_0_0_ed
0x0_0_0_f1
0x0_0_0_f5
0x0_0_0_f7
0x0_0_0_fb
0x0_0_0_fd
All tests successful!
~/pico-git/pico/samples/MultiStream/software$