RSoC Framework

www.rsoc-framework.com

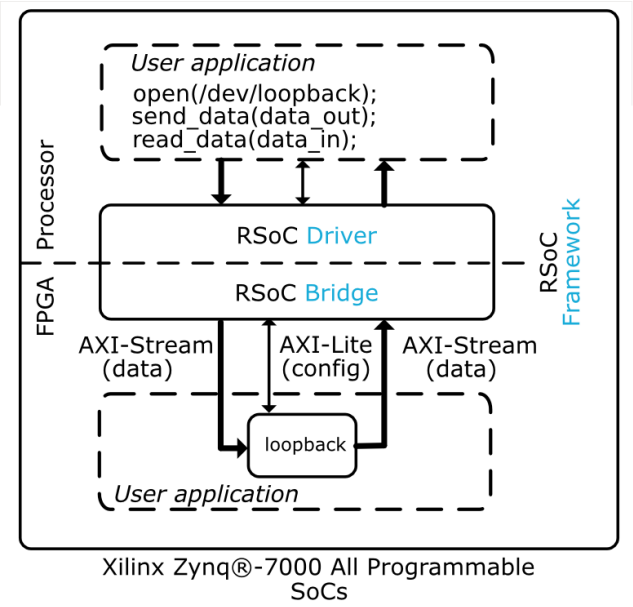
Loopback demonstration

Application Note

Version 1.0
October 2014

Introduction

The goal of this demonstration is to describe contents of the simple FPGA loopback demonstration based on the RSoC Framework for Zynq. The application consists of FPGA and processor parts. The FPGA part is used to forward data coming from CPU back to the CPU through RAM (loopback). The processor controls sending of data. The demonstration works on Zedboard, however, it may be ported to another board. After reading this document the reader is expected to understand the architecture of the demonstration and to be able to modify both the software and hardware parts of the application.



Requirements

The demonstration is prepared for Avnet Zedboard. To customize the application you may need:

- ARM toolchain that is part of the Xilinx SDK. It is also possible to use another ARM toolchain, for example, the Sourcery CodeBench ARM 2013.11.
- Xilinx Vivado (tested on 2013.4).

Contents of demonstration

The demonstration can be downloaded from `rsoc-framework.com/files/loopback_demo_bin.zip` (binaries for Quick start) and `rsoc-framework.com/files/loopback_demo_src.zip` (source code for custom modifications).

Quick start

The application binaries are located at `rsoc-framework.com/files/loopback_demo_bin.zip`.

Copy the following files to the root of the Zedboard's SD card:

+ `zedboard/rootfs.cpio.uboot` filesystem,
+ `zedboard/uImage` the Linux Kernel,
+ `zedboard/devicetree.dtb` device-tree describing the hardware and FPGA firmware,
+ `zedboard/BOOT.BIN` Zynq specific bootloader and FPGA design,
+ `zedboard/uEnv.txt` U-Boot environment.

Boot Zedboard with the SD card prepared in the previous step. If booting fails for some reason, try to reset the U-Boot environment and boot manually by *sdboot* command.

```
> env default -a
> run sdboot
```

After the Linux is booted, login as *root* with password *root*. Now, you can run the `loopback` application to send data into the hardware and receive the results.

```
# echo "Hello World!" | loopback > hello
     0.0 MB/s      0.0 MB/s 00000002 words 00000001 frames
# cat hello
     Hello World!
# loopback < /dev/urandom > result
          0.8 MB/s      0.6 MB/s 00523778 words 00001024 frames
# wc -c result
  4190208 result
```

You can check that 4190208 B = 523778 · 8 B − 16 B. Note that the *Hello World!* string is send as two 64 bit words and that is 16 B.

Software application sources

The application sources are located at `rsoc-framework.com/files/loopback_demo_src.zip`.

The Loopback demo application consists of the following sources:

+ `loopback/main.c` Function `main()` and other general initialization stuff.
+ `loopback/loopback.c` Implementation of communication with the Loopback Accelerator.
+ `loopback/Makefile` Makefile to build the application.
+ `loopback/README` Build instructions.

Firmware (FPGA) application sources

+ `fpga/cores/loopback_accelerator_v1_00_a`	Loopback Accelerator IP Core.
+ `fpga/cores/rsoc_bridge_v1_00_a`	Dependencies of the Loopback Accelerator.
+ `fpga/cores/rsoc_bridge_zynq_v1_00_a`	RSoC Bridge for Zynq IP Core*.
+ `fpga/project/loopback_acc`	Xilinx Vivado project with a loopback design*.

The Loopback Accelerator consists of the Vivado IP Core (a wrapper) and source code that is located in the `rsoc_bridge_v1_00_a` library as `hdl/vhdl/test/loopback_acc.vhd` (the core of the accelerator).

* To obtain the Vivado project (free of charge) and the RSoC Bridge for Zynq (a trial version is free of charge) please contact us at `info@rsoc-framework.com`.

Customization of application

The software and firmware application sources can be freely modified to suit your needs. To build modified firmware of the application, you need the prepared Vivado project with the RSoC Framework included. The Vivado project with a trial version of the RSoC Framework can be obtained from `info@rsoc-framework.com` free of charge.

Next steps show how to customize this application software and firmware to suit your needs with the trial version of the RSoC Framework. You can see how to extend the Loopback Accelerator to touch the data going through it and how to change the software to reflect those changes. Then, it is shown how to extend the address space of the Loopback Accelerator and how to access it from the software application.

Open the Loopback Accelerator project in Vivado. It is located in the obtained archive at

`fpga/project/loopback_acc/loopback_acc.xpr`

Modify data path in Loopback Accelerator (firmware)

In this step we change the Loopback Accelerator to modify incoming data by *bit inversion* operation (¬0x01234567 = 0xfedcba98). Open file `loopback_acc.vhd` in your favourite editor and make the following changes.

- **Step 1** Define new signal `neg_tdata` of type `std_logic_vector(C_DATA_WIDTH - 1 downto 0)`.

```
73          signal read_req      : std_logic_vector( 1 downto 0);
74          signal read_ack      : std_logic;
75          signal write_req     : std_logic_vector( 1 downto 0);
76          signal write_ack     : std_logic;
77
78          signal neg_tdata     : std_logic_vector(C_DATA_WIDTH - 1 downto 0);
79
80 begin
81
82          ---
83          -- Generate standardized INFO vector:
```

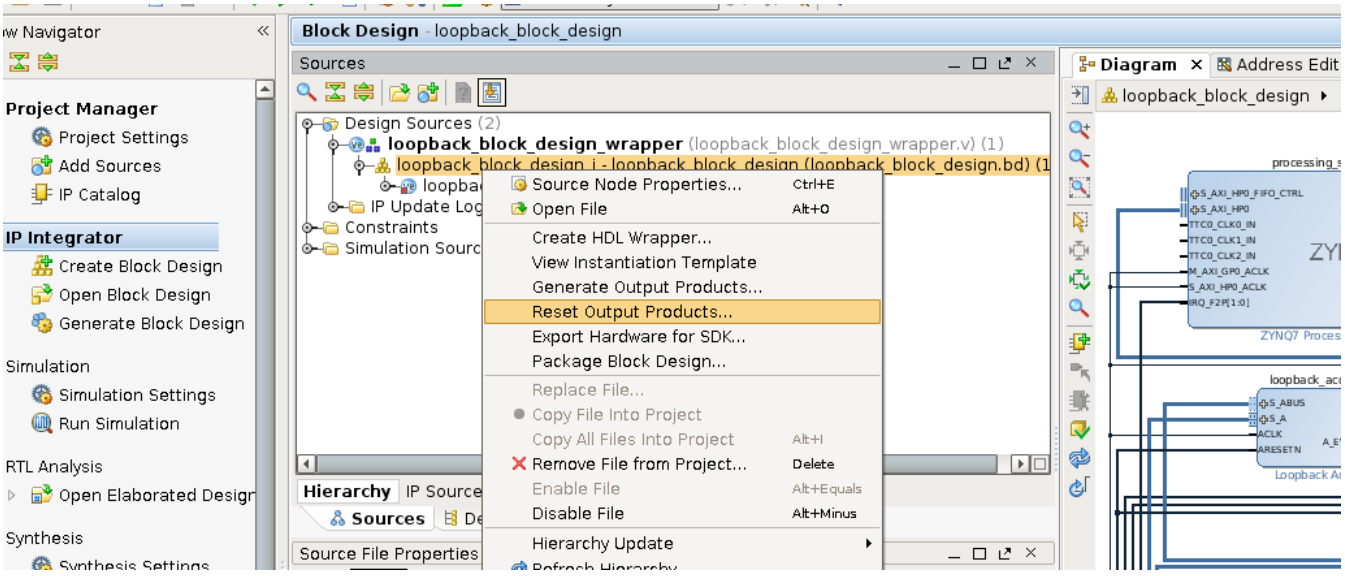
- **Step 2** Change assignment of the input port `S_TDATA` (`reg_stage` unit) to `neg_tdata`.

```
100         port map (
101                 ACLK     => ACLK,
102                 ARESETN  => ARESETN,
103
104                 S_TDEST  => (-1 downto 0 => '0'),
105                 S_TID    => S_A_TID,
106                 S_TUSER  => S_A_TUSER,
107                 S_TDATA  => neg_tdata,
108                 S_TKEEP  => S_A_TKEEP,
109                 S_TVALID => S_A_TVALID,
110                 S_TREADY => S_A_TREADY,
```

- **Step 3** Assign inversion of `S_A_TDATA` into signal `neg_tdata`.

```
118                 M_TREADY => M_A_TREADY,
119                 M_TLAST  => M_A_TLAST
120         );
121
122         neg_tdata <= not S_A_TDATA;
123
124         ---
125         -- Counter of beats with valid data seen by the core.
126         ---
127         cnt_beatsp: process(ACLK)
```

- **Step 4** Synthetize the design with the applied changes. Perform *Reset Output Products* and then follow the common Vivado flow: *Run Synthesis*, *Run Implementation* and *Generate Bitstream*.



- **Step 5** Copy the synthetized design `loopback_block_design_wrapper.bit` located in the implementation folder `project/loopback_acc/loopback_acc.runs/impl_1` to the SD card for Zedboard.
- **Step 6** Reload the design with `reload_fpga.sh` script and test by the original program `loopback`. It now produces data with inverted bits (special file /dev/zero produces only zeros).

```
# reload_fpga.sh /mnt/loopback_block_design_wrapper.bit
(omitted)
# head -c 32 /dev/zero | loopback | hexdump -C
00000000  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
     0.0 MB/s      0.0 MB/s 00000004 words 00000001 frames
^C
```

Modify software to reinvert data from accelerator

The Loopback Accelerator inverts all data. We now modify the loopback program to return the inverted data back to the original form. Open file `loopback.c` in your favourite editor and follow the steps.

- **Step 1** Modify function `copy_fds` to invert data in the buffer *b* when writing to *stdout* (we must invert either while reading from *stdin* or while writing to *stdout*, not both!).

```
34              }
35      } while(errno == EAGAIN);
36
37      if(fout == STDOUT_FILENO) {
38              for(ssize_t i = 0; i < rlen; ++i)
39                      b[i] = ~b[i];
40      }
41
42      size_t total = 0;
43
44      while(total < rlen) {
45              ssize_t wlen = write(fout, b + total, rlen - total);
```

- **Step 2** Compile the program with an appropriate ARM toolchain. The Xilinx SDK toolchain is used in this example.

```
loopback/ $ PATH=$PATH:/opt/Xilinx/SDK/gnu/arm/lin/bin
loopback/ $ make CC=arm-xilinx-linux-gnueabi-gcc
loopback/ $ ls -l loopback
-rwxr-xr-x 1 user users 13660 Oct 24 18:34 loopback
```

- **Step 3** Load the design implemented in the previous step and run the modified version of the loopback program from SD card. The program produces the same output as the generated input.

```
# reload_fpga.sh /mnt/loopback_block_design_wrapper.bit
(omitted)
# head -c 32 /dev/zero | /mnt/loopback | hexdump -C
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000020
     0.0 MB/s      0.0 MB/s 00000004 words 00000001 frames
^C
```

Modify address space of the firmware Accelerator

In this step we modify again the Loopback Accelerator. We add a new register into its address space. The RSoC Framework provides access to the address space over Accelerator's Bus (ABus) based on the AXI-Lite specification. This address space is accessible from the loopback program by `mmap` function call.

In this example we compute a simple checksum of the data going through the Loopback Accelerator. We add a sum register sum with clear-on-write ability. The register is 64 bits wide (represented as two 32 bits wide registers on ABus). Writing anything to the lower part of the register clears its contents to zero.

We extend the Loopback Accelerator by few new signals, define a new VHDL process that performs the sum of data. At the same time, we fork the `S_A_TREADY` output pin for internal use (restriction of VHDL language) and increase number of registers seen by the AXI-Lite Endpoint component.

Open again the `loopback_acc.vhd` in your favourite editor and follow the steps:

- **Step 1** Define signals for a new register `reg_sum`. At this step we also update width of signals related to the ABus – `read_data`, `read_req` and `write_req`. Finally, it is necessary to make output port `S_A_TREADY` readable. For that purpose, we define an internal helper signal `s_a_tready_x`.

```
69          signal cnt_frames_clr: std_logic;
70          signal cnt_frames    : unsigned(31 downto 0) := (others => '0');
71
72          signal read_data     : std_logic_vector(127 downto 0);
73          signal read_req      : std_logic_vector( 3 downto 0);
74          signal read_ack      : std_logic;
75          signal write_req     : std_logic_vector( 3 downto 0);
76          signal write_ack     : std_logic;
77
78          signal neg_tdata     : std_logic_vector(C_DATA_WIDTH - 1 downto 0);
79
80          signal reg_sum       : unsigned(63 downto 0) := (others => '0');
81          signal reg_sum_we    : std_logic;
82          signal reg_sum_clr   : std_logic;
83
84          signal s_a_tready_x  : std_logic;
85
```

- **Step 2** Insert the `s_a_tready_x` signal between `reg_stage` unit and the output pin `S_A_TREADY`.

```
111                 S_TID    => S_A_TID,
112                 S_TUSER  => S_A_TUSER,
113                 S_TDATA  => neg_tdata,
114                 S_TKEEP  => S_A_TKEEP,
115                 S_TVALID => S_A_TVALID,
116                 S_TREADY => s_a_tready_x,
117                 S_TLAST  => S_A_TLAST,
118
119                 M_TID    => M_A_TID,
120                 M_TUSER  => M_A_TUSER,
121                 M_TDATA  => M_A_TDATA,
122                 M_TKEEP  => M_A_TKEEP,
123                 M_TVALID => M_A_TVALID,
124                 M_TREADY => M_A_TREADY,
125                 M_TLAST  => M_A_TLAST
126         );
127
128         neg_tdata <= not S_A_TDATA;
129
130         S_A_TREADY <= s_a_tready_x;
131
```

- **Step 3** Implement process `reg_sump` that performs the sum of input data. Consider that the input data signal `S_A_TDATA` has configurable width. In this example we implement sum of `S_A_TDATA` that is up to 64 bits wide. The register `reg_sum` is updated every time a valid data comes from the AXI-Stream. The clear of the `reg_sum` register is done when writing into the index 2 (the third register in the address space). Note the `write_ack` usage (it is unnecessary in this example) that prevents possible unwanted hold of the `reg_sum_clr` signal.

```
132     reg_sump: process(ACLK)
133     begin
134             if rising_edge(ACLK) then
135                     if reg_sum_clr = '1' then
136                             reg_sum <= (others => '0');
137                     elsif reg_sum_we = '1' then
138                             reg_sum <= reg_sum + resize(unsigned(S_A_TDATA), 64);
139                     end if;
140             end if;
141     end process;
142
143     reg_sum_we  <= S_A_TVALID and s_a_tready_x;
144     reg_sum_clr <= write_req(2) and write_ack;
145
```

- **Step 4** To make contents of the `reg_sum` register available on ABus, Note that the width of the `read_data` signal must be 128 bits. The address space of the component is extended by:
  - 0x00000008 reg_sum lower (clean on write)
  - 0x0000000c reg_sum higher

```
228     --    0x00000000  cnt_frames  (clear on write)
229     --    0x00000004  cnt_beats   (clear on write)
230     ---
231     read_data <= std_logic_vector(reg_sum & cnt_beats & cnt_frames);
232
```

- **Step 5** Finally, update count of registers the AXI-Lite Endpoint handles and update the automatic acknowledging logic.

```
185     axi_lite: entity work.axi_lite_endpoint
186     generic map (
187             C_AXI_DATA_WIDTH => 32,
188             C_AXI_ADDR_WIDTH => 32,
189             C_REGISTER_WIDTH => 32,
190             C_REGISTERS      => 4
191     )
192     port map (
```

```
236         read_ackp: entity work.req_ack
237         generic map (
238                 REQ_COUNT => 4
239         )
240         port map (
241                 CLK => ACLK,
242                 REQ => read_req,
243                 ACK => read_ack
244         );
245
246         ---
247         -- Acknowladge generation for write requests.
248         ---
249         write_ackp: entity work.req_ack
250         generic map (
251                 REQ_COUNT => 4
252         )
253         port map (
254                 CLK => ACLK,
255                 REQ => write_req,
256                 ACK => write_ack
```

- **Step 6** Synthetize the design (as was already shown in the previous Loopback Accelerator modification) and copy the resulted bitstream to the SD card used for Zedboard. To see whether the modification is working, modify the software loopback program by following the next example.

Modify software to read and clear the sum

We added a new register to address space of the Loopback Accelerator in the previous step. To access it from the loopback software program we need to make a small change in the code. We report and clear the checksum every time a `read()` from the accelerator is finished (to guarantee that the checksum is complete for the last data frame).

For this, you only need to add a small piece of code that reads the appropriate register from the accelerator's address space. The address space is already mapped as `uint32_t *g_abus` variable.

Edit the `loopback.c` file again and insert the following snipped of code there:

```
115                 if((ret = copy_fds(fd, STDOUT_FILENO, BUFOUT_SIZE)) < 0)
116                         goto leave_loop;
117
118                 if(ret > 0) {
119                         uint64_t sum = ((uint64_t) g_abus[3]) << 32;
120                         sum |= g_abus[2];
121                         fprintf(stderr, "checksum: %016" PRIx64 "\n", sum);
122                         g_abus[2] = 0xdeadbeef;
123                 }
124
125                 bytes_out += ret;
126         } while(1);
```

You can see, the code just reads from the array `g_abus` at indices 3 (the higher part of the sum register) and 2 (the lower part of the sum register). It prints the value on *stderr* (to do not mess it with the actual data output on *stdout*) and finally it clears the sum register by writing an arbitrary value into it.

Compile the modified software program and copy the `loopback` binary to your SD card. Test it simply by running (sometimes the output *Hello* may not be shown because of automatic buffering of the *stdout* and we do not flush it explicitly).

```
# echo Hello | /mnt/loopback
checksum: 00000a6f6c6c6548
     0.0 MB/s      0.0 MB/s 00000001 words 00000001 frames
Hello
^C
# echo Hello | hexdump -C
00000000  48 65 6c 6c 6f 0a                                    |Hello.|
00000006
```

You can check the checksum is correct by the `hexdump` program that outputs the binary representation of the word *Hello*.