DATA TRANSMISSION THROUGH GIGABIT ETHERNET FROM A LVDS INTERFACE USING A SOC (SINGLE BOARD COMPUTER + FPGA)

David G. Shatwell
Department of Electrical Engineering
University of Engineering and Technology–UTEC
Lima 15063, Peru
david.shatwell@utec.edu.pe

Joaquin Verastegui
Department of Research, Development and Innovation
Jicamarca Radio Observatory
Geophysical Institute of Peru
Lima 15464, Peru
jverastegui@igp.gob.pe

John Rojas
Department of Research, Development and Innovation
Jicamarca Radio Observatory
Geophysical Institute of Peru
Lima 15464, Peru
jrojas@igp.gob.pe

ABSTRACT

The objective of this project was to design and implement a system capable of transmitting data at high speeds from the JARS 2.0 radar to a remote computer through Gigabit Ethernet using a system on chip (SoC). The system has two main stages: (i) data acquisition from the LVDS interface and (ii) data transmission to the computer through a communication protocol. In order to acquire data from the LVDS interface, the FPGA was used to implement a system capable of multiplexing and copying the data to a memory shared with the processor. Then, a program running on the processor was used to read the data from the shared memory and send it to the PC with the UDP protocol.

Keywords JARS 2.0 · Gigabit Ethernet · SoC · LVDS · UDP

1 Introduction

The JARS system acquires data from the main radar of the Jicamarca Radio Observatory and transmits it via a LVDS interface. Custom hardware is then used to collect the data from the interface and send it to a remote computer using a standard communication protocol. In JARS 2.0, the UDP protocol stack was implemented on a Spartan 6 FPGA for this purpose. However, there are several disadvantages with this approach: the evaluation board and the software used to program it (ISE Design Suite) are discontinued, the LVDS connector is proprietary, the communication bandwidth is limited by the FPGA, and the communication protocol is limited to UDP only.

There are now better alternatives, such as SoCs (System-on-Chip), which have an FPGA and a microprocessor. The main advantage of a SoC for the implementation of an LVDS-Ethernet interface is that it can use a distribution of the Linux operating system that handles the entire network load. In other words, by using a microprocessor, it is not necessary to implement the TCP/IP and UDP protocol stacks directly on the FPGA fabric. Other advantages of modern SoCs are their low cost (around $100), high frequencies (600 MHz), non-proprietary LVDS connectors and compact size.

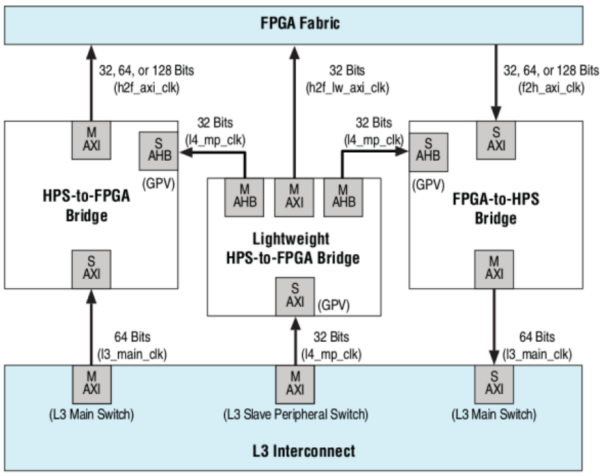

Figure 1: HPS-FPGA communication system.

1.1 JARS 2.0 data transmission

The JARS system sends data acquired from the main radar as 32-bit words. When using the 8 antenna channels, JARS is capable of sending data at a frequency of 1 MHz, which represents a speed of 256 Mbps. The signals are sent from JARS through a low-voltage differential signaling (LVDS) interface. An advantage of LVDS over other types of single-ended schemes is that it is less susceptible to common mode noise, because noise coupled onto the interconnect is seen as common mode modulations by the receivers and is rejected [Huq and Goldie, 1998].

1.2 DE0-Nano-SoC

The DE0-Nano-SoC is a development board from Terasic containing a Cyclone V SoC from Intel (formerly Altera). The Cyclone V consists of an FPGA and a dual-core ARM A9 processor. The FPGA and microprocessor communicate with each other via three AXI buses, a control bus (32-bit) and two data buses (up to 128 bits), as shown on figure 1. Additionally, the SoC includes multiple peripherals, such as general purpuse inputs/outputs (GPIO), an analog-to-digital converter (ADC), one 1GB DDR3 SDRAM (32-bit data bus), one Gigabit Ethernet RJ45 connector, among others [Terasic, 2019].

1.3 Direct access memory controller

A direct access memory (DMA) controller is a circuit that allows peripherals to access the system's memory independently of the CPU. Without a DMA controller, the CPU is fully occupied during read/write operations and cannot perform other operations concurrently. Thus, the DMA, is able to boost the overall performance of the system in which it is implemented [Ahmed et al., 2019].

2 Project development

In order to develop the data transmission system, the first thing needed was to establish flow of data from the FPGA to the microprocessor. Figure 2 shows the proposed system, where the thin arrows represent control signals while the thick arrows represent data transmission. After establishing the necessary connections and flow of data, the rest of the system design can be separated into three parts: (i) creation of FPGA blocks written in VHDL and Verilog, (ii) instantiation of IP cores and connections between the FPGA and the processor with the Platform Designer tool, and (iii) software development using the C language.

2.1 FPGA

A finite state machine (FSM), which is referred in this paper as SRAM_FSM, was created in the FPGA. The main purpose of the FSM is to generate sequences of data and memory addresses for the SRAM at a frequency of 8 MHz. It should be noted that in the real system, data from JARS arrives in 8-byte packets and must be demultiplexed before

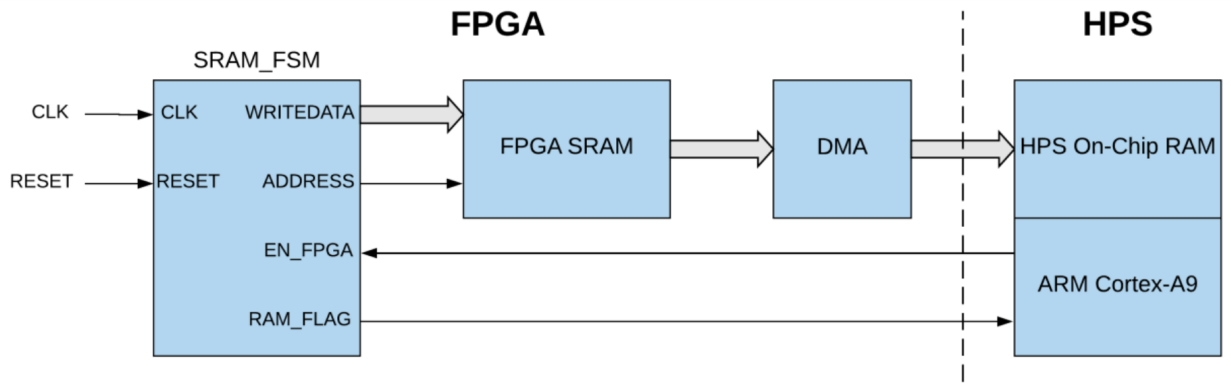

Figure 2: Data flow of the proposed system.

being stored in memory. However, for testing purposes it is easier to use a number generator connected directly to the SRAM memory.

2.2 SoC

The following Intel IP cores were instantiated on the SoC:

2.2.1 PLL

the first IP core is a PLL that generates 2 clock signals from a 50 MHz reference. The first clock has a frequency of 100 MHz and is connected to the HPS-FPGA buses, the SRAM and the PIOs. The second clock has a frequency of 400 MHz and is used exclusively for the "NUM_GENERATOR" block to generate numbers at a rate of 8 MHz.

2.2.2 HPS

The hard processor system (HPS) block refers to the ARM Cortex-A9 processor. Despite the large number of inputs and outputs, the processor only uses two of them. The "h2f_axi_slave" bus is connected to the "write_master" port of the DMA and is used to acquire data from the FPGA. On the other hand, the "f2h_axi_master" bus is connected to the DMA control port and to the "en_fpga" and "sram_flag" PIOs, which serve as control signals between the FPGA and the processor.

2.2.3 DMA

The DMA block is used to copy the data from the SRAM memory in the FPGA to the On-Chip RAM of the processor. This block reads data from the SRAM through the "read_master" port and writes to the On-Chip RAM through the "write_master" port. In addition, it has a software-controlled port that is used to indicate the memory locations to be transferred from the FPGA to the microprocessor.

2.2.4 SRAM

The SRAM block is a 64 KB memory with 32-bit words instantiated in the FPGA. This memory has two slave ports: one for reading and one for writing. As mentioned before, the read port is connected to the DMA while the write port is exported to the FPGA and controlled by the "SRAM_FSM " block.

2.2.5 PIO: FPGA enabler and SRAM flag

The "en_fpga" block works as a 1-bit control signal that is used to enable the FPGA blocks from the main program. Its "s1" port is connected to the "h2f_axi_master" port of the processor. On the other hand, the "RAM_flag" blockworks as a 6-bit control signal that is used to tell the processor which section of SRAM memory is being written. Its "s1" port is also connected to the processor's "h2f_axi_master" port.

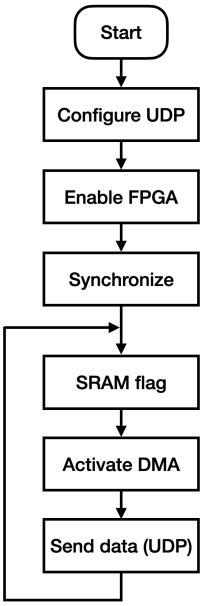

Figure 3: SoC program flow chart.

Table 1: Virtual memory directions used in the program.

Component	Port name	Base direction	Offset
F2H AXI Slave	hps_0.f2h_axi_slave	0xFFFF0000	0x00000000
DMA Control Port	dma.control_port_slave	0xC0000000	0x00000000
SRAM	SRAM_0.s1	0xC0000000	0x00000000
FPGA Enabler	en_fpga.s1	0xC0000000	0x00000020
SRAM Flaf	sram_flag.s1	0xC0000000	0x00000030

2.3 ARM Cortex-A9 microprocessor

In order to collect the data from the SRAM and send it over Ethernet, a program in the C language was written. The flowchart of this program is shown on Figure 3.

2.3.1 Address acquisition from virtual memory

The first thing to do in the program is to define macros with the memory addresses to which the FPGA components are mapped (Table 1). Since Linux does not allow the user to directly access the physical memory, pointers storing virtual memory addresses were also defined.

Then, within the main function, the `/dev/mem` file is opened to obtain a file descriptor and the virtual memory address of the LW HPS-to-FPGA Bridge bus is acquired with the `mmap` function. Pointers pointing to the virtual memory address of the DMA control port and the "sram_flag" and "en_fpga" components are defined as the bus address with an offset. In this case, the memory offset of the components is shown in Table 1.

2.3.2 UDP configuration

In order to make the UDP configurations, the first thing needed was to open a socket to transmit data (Tx) with the `socket` function and specify that IPv4 and the UDP protocol were going to be used. In addition, a "talker" structure was created with the IP address and the port to which the data was going to be sent.

2.3.3 Synchronization

Before the main loop, the control signal "en_fpga" is used to enable the FPGA so that the first data arriving is at the first memory location. Then, the program waits for a change in the SRAM flag for the DMA to start transferring data to the

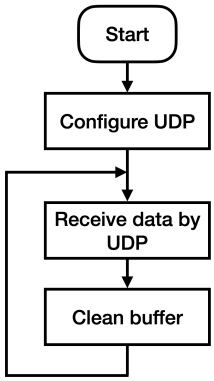

Figure 4: PC program flowchart.

On-Chip RAM. This synchronization process ensures that the following three processes are running in parallel at all times:

- Processor sends data on section n of On-Chip RAM through UDP
- DMA copies section $n + 1$ of SRAM to On-Chip RAM
- Finite state machine stores data coming from JARS on section $n + 2$ of SRAM

2.3.4 Main loop

In the main loop, the program reads the SRAM flag again and starts transferring the corresponding memory section with the DMA. At the same time that the data is being transferred, the program determines the memory addresses of section n and uses the `sendto` function to send the data of that section via UDP. The `sendto` function uses as arguments the port file descriptor, the pointer to the array and the "server" structure created in previous parts of the program.

2.4 Target PC

In order to acquire data sent by UDP from the SoC, a program written in the C language similar to that of the UDP server was also created. The flowchart of the program is shown in Figure 4.

2.4.1 UDP configuration

To configure the UDP protocol, the program opens a socket to receive data using the `socket` function and specifies that the IPv4 and UDP protocols will be used. In addition, a "listener" structure is created with the IP address and the port through which the data is received and a 1024 bytes buffer to store the data.

2.4.2 Main loop

In the main loop of the UDP client, the program receives packets with the `receivefrom` function, which receives as arguments the file descriptor generated by the `socket` function, the buffer to store the data and the "listener" structure created previously. Once the packet is received, the buffer is cleared and the process is repeated.

3 Results

3.1 Hardware validation

To verify the operation of the finite state machine created in the FPGA, a testbench was created with the Modelsim tool. The results of the testbench are shown in Figure 5. It can be seen that, with a clock signal of 350 MHz, a data and memory address is generated every 125 ns (8 MHz). This prooves that the state machine connected to the SRAM memory is working properly.

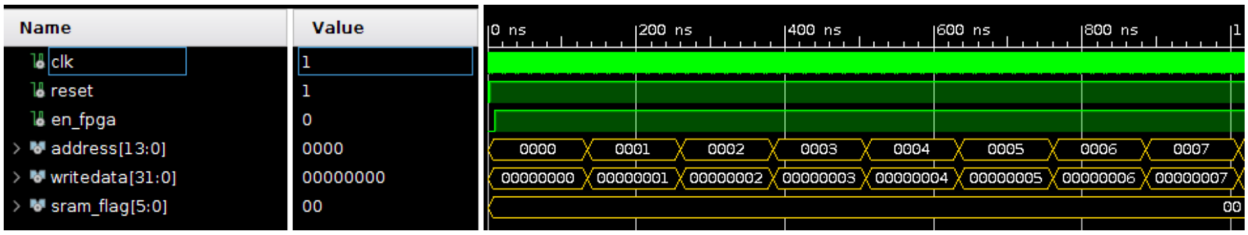

Figure 5: Finite state machine simulation.

3.2 Software validation

3.2.1 Packet content validation

In the first test used to validate the software, it was verified that the contents of the packets received by the PC are the same as those sent by the SoC. For this purpose, some modifications were made to the SoC and PC codes. In the SoC, a vector was created to store the contents of 10 consecutive packets, which is equivalent to the total contents of the memory. Once the main loop was executed 10 times, the program printed the contents of the vector along with the status of the SRAM flag on the terminal. From this test it was determined that the contents of all packets on the target PC are identical to those sent by the SoC.

3.2.2 Percentage of received packets validation

The objective of the second test was to verify the percentage of packets received by the PC. To do this, 10 tests were performed consisting of the SoC sending 100 packets and, with a counter, verifying how many packets reached the destination PC. With this test it was determined that the percentage of packets received by the PC is approximately 96.5%.

4 Discussion

To make the data forwarding system work properly, it is necessary to take into account the clock frequency connected to the data buses and system components, especially the DMA. A very low frequency (<50 MHz) causes the data transfer between the FPGA and the processor to be slow, which in turn causes packets to be lost, while a very high frequency (>150 MHz) causes the SoC processor to hang.

As for the percentage of packets received, it was observed that there was no variation between the intermediate frequencies (50 to 100 MHz). It is believed that this happens because the bottleneck is not in the AXI buses or in the DMA, but in the SoC program that sends the data by the UDP protocol.

5 Conclusions

In the development of this project, it was possible to work with the DE0-NanoSoC development board to send data generated in the FPGA over Ethernet at a speed of 256 Mbps. The system writes the data arriving at the SoC to SRAM memory, and then reads and sends the data over Ethernet.

The use of the Platform Designer tool of the Quartus Prime program proved to be very useful for the development of the project. This tool is not only necessary to design the buses that connect the FPGA to the processor, but also to easily instantiate IP cores in the FPGA. However, in order to create the buses it is necessary to have a good understanding of how the data flow inside the SoC works. For this purpose, it was very useful to read the Cyclone V manual, which explains in detail how the SoC works.

6 Recommendations

In a future work, it is recommended to find a way to optimize the code that runs on the SoC in order to improve the percentage of packets sent correctly. In addition, it is recommended to make a shield for the DE0-Nano-SoC where the LVDS wires of the JARS can be connected and make more accurate tests.

7 Acknowledgements

We would like to thank the Jicamarca Radio Observatory and the Geophysical Institute of Peru for having borrowed the necessary equipment for this project.

References

Syed B Huq and John Goldie. An overview of lvds technology. *National Semiconductor Application Note*, 971:1–6, 1998.

Terasic. De0-nano-soc user manual. 2019.

Mohammed Altaf Ahmed, Abdullah Aljumah, and M Gulam Ahmad. Design and implementation of a direct memory access controller for embedded applications. *International Journal of Technology*, 10(2):309–319, 2019.