Method of High-Speed Data Acquisition and Continuous Data Transfer using Altera® Stratix® II EP2S60 DSP Development Board

Qian Liu and S.W. Ellingson

April 27, 2009

Contents

1 Introduction 2

2 Basic Architecture 2

3 System Development 2
   3.1 Nios II hardware development ........................................ 2
   3.2 Nios II software development ........................................... 4
      3.2.1 Initialization task .................................................. 5
      3.2.2 Application task .................................................... 5
   3.3 Host PC Post-Processing ................................................... 5

4 Networking Performance Optimization 5

5 System Performance Test 7
   5.1 Throughput Rate .......................................................... 7
   5.2 Transfer Reliability ....................................................... 7

6 User Manual 7
   6.1 Run the Project ............................................................ 7
   6.2 Flash Program ............................................................. 8
   6.3 Recompile the Project .................................................... 8

A ADC Interface Module 10

B Nios II Application Software 16

C Server Application Software 17

D Data Processing Program 19
1 Introduction

Ethernet has become a standard data transport paradigm for embedded systems in many applications since the transport technology is cheap, abundant, mature and reliable. This report has implemented an Altera® Nios® II based system for data acquisition and transfer over Ethernet using user datagram protocol (UDP). The network client (the data source) is implemented in FPGA using Verilog HDL and C languages; and the network server (the data recorder) is running on the host PC using Python code.

The goal is to be capable of transferring continuous data at high throughput rate, where continuous here means that no data will be lost during the sampling and transfer process. The designed system has a throughput rate of 80 Mb/s; it acquires 8-bit samples at the rate of 10 MSPS.

2 Basic Architecture

The system is built on the Altera® Stratix® II EP2S60 DSP development kit, as shown in Figure 1. The on-board 12-bit analog-to-digital converter (ADC) AD9433 is used for the data sampling; while the 10/100 single chip PHY/MAC SMSC LAN91C111 is used for the UDP communication between the development board and the host PC.

Figure 2 demonstrates the basic architecture of the system, consisting of ADC, FPGA, and Ethernet controller. The Nios II processor runs the MicroC/OS-II real-time operating system (RTOS) and provides the drivers for the devices. The ADC interface controls the data sampling, and the Ethernet interface is related with the Ethernet controller.

It is also shown in Figure 2 that the ADC interface is implemented in Verilog HDL. The Nios II/f processor and Ethernet interface are instantiated in SOPC Builder and configured in the Nios II IDE. The Python code runs on the host PC.

3 System Development

3.1 Nios II hardware development

As an Altera® Nios® II based system, Quartus® II software can be used to create and process the Nios II system design that interfaces with components on development board [1]. The designed system includes the following components:

Nios II/f core with 64 Kbytes of instruction/data cache
16 Kbytes on-chip memory
256 Kbytes SRAM
32 Mbytes SDRAM
Figure 1: Image of Stratix II EP2S60 DSP Development Board.\textsuperscript{a}

\textsuperscript{a}http://www.altera.com/products/devkits/altera/kit-dsp-2S60.html

Figure 2: Block diagram of the system architecture.
16 Mbytes Flash memory
ADC interface
Ethernet interface
Timer
JTAG UART
System ID component

All the components except the ADC interface – composed of a data width bridge, a swing buffer and a cross clock bridge – can be configured as intellectual property (IP) cores using Altera’s MegaWizard software. The Nios II/f core, the on-chip memory, and the timer are used for the RTOS. The JTAG UART is used for the USB-Blaster\textsuperscript{TM} download cable to configure the Stratix II FPGA directly using an SRAM Object File (.sof). The SRAM and the SDRAM stores the executable code, function call parameters and temporary data; while the Flash memory stores the firmware for bootload. The Ethernet interface makes the control registers and data registers of the SMSC LAN91C111 accessible through application programming interface (API).

Both the data width bridge and the swing buffer of the ADC interface use the a finite state machine (FSM). The data width bridge converts 8-bit data to 32 bits, matching with the width of the Nios II processor’s 32-bit data bus. The swing buffer is used to prevent underrun and overflow conditions. When one FIFO is in write operation, the other one is in read operation; then the two FIFOs swap the operation. The cross clock bridge is an asynchronous FIFO, interfacing two clock domains. The details about the implementation of the ADC interface are listed in Appendix A.

### 3.2 Nios II software development

The prototype for the Nios II application was the Nios II C/C++ application project template \texttt{simple_socket_server}, using the NicheStack TCP/IP Stack on the basis of the RTOS multithreaded environment [2]. The NicheStack TCP/IP Stack provides the interrupt-based driver support for the SMSC LAN91C111 MAC/PHY device. Our application has modified the C design file \texttt{simple_socket_server.c}, which is detailed in Appendix B, to implement the UDP communication between the FPGA (as a client) and the host PC (as a server).

There are two fundamental tasks in our application: the SSSInitialTask instantiates all of the RTOS resources; and the SSSSimpleSocketServerTask manages the socket server connection and calls UDP socket routines to transfer data. The SSSInitialTask should have the higher priority to ensure that the application code does not attempt further initialization until the RTOS is running and I/O drivers are available [2].

Note that the IP address of where the data would like to be sent to and the port number should be assigned in client program. As illustrated in Appendix B, the IP address of the host PC in our application is \texttt{192.168.1.213}, and port 1739 has been used.
3.2.1 Initialization task

The SSSInititalTask initializes the NicheStack TCP/IP Stack at the beginning. It calls the `alt_iniche_init` function to initialize the stack for use with the RTOS and the `netmain` function to start the iniche-specific network tasks as well as initialize the network devices. When the global variable `iniche_net_ready` is set to a non-zero value, the NicheStack stack has completed initialization. It is time to perform the application initialization steps, such as the initialization of the SSSSimpleSocketServerTask.

3.2.2 Application task

The SSSSimpleSocketServerTask includes the code for the network client and continues forever after that it first establishes a UDP socket. It sends the data located in the swing buffer to the server (host PC) using the `sendto` function continuously. As UDP is a connectionless protocol, the client (DSP development board) does not establish a connection with the server [3].

In order to have the maximum throughput rate, this application task does nothing but sends the data generated from the ADC.

3.3 Host PC Post-Processing

On the host PC, Python code that has been detailed in Appendix C implements the network server using UDP. For a UDP server, Python code creates a socket, sets the socket options and binds the IP address with the port number. Both the IP address and port number of the host PC should be the same as those assigned in the Nios II application software. Then it is ready to receive the data using the `recvfrom` function. Though UDP is connectionless, it can solve the problem of lost packets in local area network (LAN) with appropriate size of the receive buffer. Once a packet has been received successfully, the data would be written into a binary file.

4 Networking Performance Optimization

There are many factors affecting the total throughput of an embedded networking system, such as the user application, networking stack, Ethernet device and its driver, as well as the physical connection for the networking link [4]. Therefore, there are several effective methods to optimize the application to relieve the load of the Nios II CPU. We configured the software and hardware settings as follows.

The socket buffer size for the Socket’s `sendto` and `recvfrom` functions should be set large enough to alleviate the system call overhead. Here, we allocate a 1 MB for the socket buffer.
Nios II compiler optimization level: -O3
SSSInitialTask priority: 1
InetMainTask priority: 2
ClockTickTask priority: 3
SSSSimpleSocketServerTask priority: 4
Computational efficiency: Nios II/f core
Instruction/Data Caches: 64 KB
Clock frequency: 150 MHz
Data packet size: 1468 bytes
Socket buffer size: 1 MB

The maximum packet size depends on the maximum transmit unit (MTU), which is fixed by standards (as is the case with Ethernet) or decided at connect time (as is usually the case with point-to-point serial links). For Ethernet at the network layer, the MTU is normally 1500 bytes; therefore, the maximum data payload is 1472 bytes considering the IP header and UDP header. However, with the increase of data packet size, the following experiment shows that the highest throughput rate increases to a threshold and then decreases as shown in Figure 3. The optimum data payload is determined as 1468 bytes by comparing the throughput rate versus different data payload, resulting in a throughput rate of 80 Mb/s for 100baseT (100 Mb/s) Ethernet.

![Figure 3: Data packet size vs. throughput rate over 100baseT Ethernet.](image-url)
5 System Performance Test

5.1 Throughput Rate

To make a good approximation of the maximum throughput achievable, we did a test in which the application did very little apart from sending or receiving data through the networking stack and calculates the “raw” throughput rate of UDP data transaction. Averaging over a large amount of data, we find that the throughput rate is about 80 Mb/s when the SMSC LAN91C111 is configured in 100baseT mode.

5.2 Transfer Reliability

To test data integrity, a MATLAB program is used to read the data from the recorded file. After sending 8000 packets of 1468 8-bit samples, the received data are plotted in Figure 4. Figure 4 shows that the received data are recovered without fracture; that is, the data can be transferred continuously.

![Eye Diagram](image)

Figure 4: Eye diagram of ramp signal.

6 User Manual

6.1 Run the Project

The steps to run the project in FPGA through JTAG cable are as follows:

1. Download Hardware Design (standard.sof) to Target FPGA using the Programmer in Quartus II software.
2. Run the Program using Nios II IDE Tool
   (1) Right-click the *simple_socket_server* project, point to *Run As*, and then click *Nios II Hardware*. The IDE downloads the program to the FPGA on the target board and starts execution.
   (2) Click *Terminate* (the red square) on the toolbar at the upper-right hand corner of the Console view to terminate the run session. At this time, the IDE disconnects from the target hardware and leaves the Nios II processor running.
3. Run the Host PC program (Python Code).
4. Run the Matlab Code for data post-processing.

### 6.2 Flash Program

Both the hardware design (standard.sof) and the software project (*simple_socket_server.prj*) can be downloaded into the flash memory using *Flash Programmer* in Nios IDE as the following instruction.

1. Download Hardware Design (standard.sof) to Target FPGA using the Programmer in Quartus II software.
2. Click *Flash Programmer* from the Tools menu in Nios IDE. If reusing an existing flash configuration, click *Load JDI File*. If it is the first time to program flash, complete the following steps:
   (1) Select *Flash Programmer* at the left side of the dialog box.
   (2) Click the *New launch configuration* button in the upper left corner of the flash programmer window.
3. Check the box titled *Program software project into flash memory*; check the box titled *Program FPGA configuration data into hardware image region of flash memory*. In the FPGA Configuration (SOF) field, type or browse to *standard.sof*. In the Hardware Image field, select the preset location at which you wish to program the .sof file, or select Custom. For example, we choose *Custom* here and specify the memory name as *ext_flash* and offset as *0x800000*. Also check the box titled *Validate Nios II system ID before software download*.
4. Click *Program Flash* button. When it completes, the non-volatile configuration is done.
5. Disconnect JTAG cable and power off the board. When setting the switch 1 and switch 4 of SW2 open, we can use the S60 board independently to communicate with host PC via Ethernet after powering on the board.

### 6.3 Recompile the Project

In the appendix, the IP address of “192.168.1.213” and the port “1739” have been used for the UDP socket. If either the IP address or the port number is needed to be revised, just update the Python code `host = ‘192.168.1.213’` and the sentence `cliaaddr.sin_addr.s_addr = inet_addr(“192.168.1.213”);` in `simple_socket_server.c` file for the IP address renewal; and modify the Python code `port=1739` and the definition `UDP_PORT=1739` in `simple_socket_server.h` file. After the revision, the project should be recompiled before running it on the Target FPGA or downloading it into the Flash memory. It can be implemented by clicking *Project*
on the Tools menu and then clicking Clean with checking the box titled Start a build immediately in the dialog box.

References


Appendices: Source Code

A  ADC Interface Module

Listing 1: a2d.v file

```
module a2d (adclk, rdclk, cs_n, rst_n, rd, a2dc,
            addr, waitreq, a2do);
// --- Port Definition
input adclk;
input rdclk;
input cs_n;
input rst_n;
input rd;
input [11:0] a2dc;
input [9:0] addr;
output waitreq;
output [31:0] a2do;
wire waitreq;
wire [31:0] a2do;

// --- Intermediate Variable
wire [7:0] a2du;
reg [1:0] current_byte, next_byte;
reg [7:0] a2di0, a2di1, a2di2, a2di3;
wire [31:0] ad_q0, ad_q1;
wire [31:0] ccb_q;
wire wfull0, rempty0;
wire wfull1, rempty1;
reg wrreq, rdreq;
reg [2:0] current_ad, next_ad;
reg [7:0] counter;
wire wrclk;
reg byte_cntr;
reg [15:0] datiL, datiH;
reg [31:0] dati;
wire datready;
wire clkfull, clkrfull;

// --- clock instantiation
bufclk pll0 (n
```
.c0 (wrclk),
 .locked ()
);

// ——— 2’s complement to binary
assign a2du[7] = ~a2dc[11];
assign a2du[6:0] = a2dc[10:4];

// ——— 8–bit to 32–bit
// 1. synchronous sequential block
always @(negedge adclk) begin
    current_byte <= next_byte;
end

// 2. combination logic block
always @(current_byte) begin
    case (current_byte)
        2'b00: next_byte = 2'b01; // BYTE1;
        2'b01: next_byte = 2'b10; // BYTE2;
        2'b10: next_byte = 2'b11; // BYTE3;
        2'b11: next_byte = 2'b00; // BYTE0;
        default: next_byte = 2'b01; // BYTE1;
    endcase
end

// 3. synchronous sequential block: give output
always @(negedge adclk) begin
    case (next_byte)
        2'b00: begin
            a2di3 <= a2du;
        end
        2'b01: begin
            a2di0 <= a2du;
            dati <= {a2di3, a2di2, a2di1, a2di0};
        end
        2'b10: begin
            a2di1 <= a2du;
        end
        2'b11: begin
            a2di2 <= a2du;
        end
        default: begin
            a2di0 <= a2du;
            a2di1 <= a2du;
            a2di2 <= a2du;
            a2di3 <= a2du;
        end
    endcase
end
```vhdl
endcase
end

// ------------------ Cross Clock Bridge Begin ------------------ //
wire rempty;
adFIFO crossclk ( 
  .data (dati),
  .wrclk (wrc1k),
  .wrreq (1'b1),
  .wrfull ( ),
  .q (cc1b_q),
  .rdclk (rdcl1k),
  .rdreq (~rempty),
  .rdfull ( ),
  .rdempty (rempty)
);

// ------------------ Cross Clock Bridge End ------------------ //

// ------------------- Swing Buffer Begin ------------------- //
reg [1:0] current_buf, next_buf;
reg idle;
reg wren0, wren1, rden;
// Control FSM
parameter
  INITIAL = 2'b00,
  BUFFER0 = 2'b01,
  BUFFER1 = 2'b10;
// synchronous sequential block
always @(posedge rdclk or negedge rst_n) begin
  if (~rst_n)
    current_buf <= INITIAL;
  else
    current_buf <= next_buf;
end
// combinatorial block
always @(current_buf or cs_n or wfull0 or wfull1
  or rempy0 or rempy1) begin
  case (current_buf)
    INITIAL: begin
      if (wfull0 & ~cs_n)
        next_buf = BUFFER0;
      else
        next_buf = INITIAL;
    end
  endcase
```
BUFFER0: begin
  if (wfull1 & rempty0)
      next_buf = BUFFER1;
  else
      next_buf = BUFFER0;
end
BUFFER1: begin
  if (wfull0 & rempty1)
      next_buf = BUFFER0;
  else
      next_buf = BUFFER1;
end
default: begin
  next_buf = INITIAL;
end endcase
end // sequential block
always @(posedge rdclk) begin
  case (next_buf)
    INITIAL: begin
      wren0 <= 1'b1;
      wren1 <= 1'b0;
      rden  <= 1'b0;
      rdreq <= 1'bz;
      idle  <= 1'b0;
    end
    BUFFER0: begin
      wren0 <= 1'b0;
      if (wfull1 & ~rempty0) begin
        wren1 <= 1'b0;
      end
      else begin
        wren1 <= 1'b1;
      end
      if (rempty0 & ~wfull1) begin
        rden <= 1'b0;
        idle <= 1'b1;
      end
      else begin
        rden <= 1'b1;
        idle <= 1'b0;
      end
      rdreq <= 1'b1;
    end
  end
end
BUFFER1: begin
    if (wfull0 & ~rempty1) begin
        wren0 <= 1'b0;
    end
    else begin
        wren0 <= 1'b1;
    end
    wren1 <= 1'b0;
    if (rempty1 & ~wfull0) begin
        rden <= 1'b0;
        idle <= 1'b1;
    end
    else begin
        rden <= 1'b1;
        idle <= 1'b0;
    end
    rdreq <= 1'b0;
end
default: begin
    wren0 <= 1'b1;
    wren1 <= 1'b0;
    rden <= 1'b0;
    rdreq <= 1'b0;
    idle <= 1'b0;
end
endcase
end

swBUF buf0 (  
clock (rdclk),
.sclr (~rst_n),
data (ccb_q),
.wrreq (~rempty & wren0),
.full (wfull0),
.rdreq (rden & rd & rdreq),
.empty (rempty0),
.q (ad_q0)
);

swBUF buf1 (  
clock (rdclk),
.sclr (~rst_n),
data (ccb_q),
.wrreq (~rempty & wren1),
.full (wfull1),
.rdreq (rden & rd & ~rdreq),
.q (ad_q1)
.empty (rempty1),
.q (ad_q1)
);

assign waitreq = idle;
assign a2do = rdreq ? ad_q0 : ad_q1;

endmodule
B Nios II Application Software

Listing 2: simple_socket_server.c file

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "includes.h"
#include "simple_socket_server.h"
#include "alt_error_handler.h"
#include "ipport.h"
#include "tcpport.h"

void SSSSimpleSocketServerTask( )
{
    int udp_socket;
    struct sockaddr_in cliaddr;
    int udp_rc = SSS_BUF_LEN;
    int nSendBufSize = SSS_TX_BUF_SIZE;
    void* rx_buff = A2D_INST_BASE;

    cliaddr.sin_family = AF_INET;
    cliaddr.sin_port = htons(UDP_PORT);
    cliaddr.sin_addr.s_addr = inet_addr("192.168.1.213");

    while (1)
    {
        udp_socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
        setsockopt(udp_socket, SOL_SOCKET, SO_SNDBUF,
                   (const char*)&nSendBufSize, sizeof(int));

        while (! (udp_socket <0) & udp_rc==SSS_BUF_LEN)
        {
            udp_rc = sendto(udp_socket, rx_buff, SSS_BUF_LEN, 0,
                           (struct sockaddr *)&cliaddr, sizeof(cliaddr));
        }

        if (udp_rc!=SSS_BUF_LEN)
            printf("Transmit failed!");
        close(udp_socket);
    }
}
```
C    Server Application Software

Listing 3: server.py file

0 from socket import *
     from ctypes import *

2 import binascii

4 # Set the socket parameters
    host = '192.168.1.213'
6 port = 1739
    addr = (host, port)
8 reclen = 1468

10 # Set the packet parameter
    PacketNum = 8000
12
14 BufferSize = reclen*PacketNum

16 # Create the socket
    UDPSock = socket(AF_INET, SOCK_DGRAM)
18
20    # Set options
    UDPSock.setsockopt(SOL_SOCKET, SO_RCVBUF, BufferSize)
    UDPSock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
22
24 # Create a binary file
    filename = 'test.dat'
    FILE0 = open(filename, 'wb+', 0)
26
28    # Receive messages
29 for j in range(0, looptime*PacketNum):
30        data, address = UDPSock.recvfrom(reclen, 0)
32        FILE0.write(data)
34
36 UDPSock.close()
    FILE0.close()

38    # Post-processing
40 FILE1 = open('test.dat', 'rb');
    data_bytes = FILE1.read(reclen*PacketNum*looptime)
42 FILE1.close()
44    data_string = bytes.decode(binascii.hexlify(data_bytes))

17
FILE2 = open('test.txt', 'w')
42 FILE2.write(data_string)
FILE2.close()
44
print('Complete!')
D Data Processing Program

Listing 4: fileIO.m file

```matlab
0  clear all;
   close all;

2  fid = fopen('test.txt', 'r');
4  data = fscanf(fid, '%2X');
   fclose(fid);
6  eyediagram(data, 200);
```