TCP 开发总结

1 开发目的

针对组内现有数据通信技术不可靠以及速率低的问题，开发基于 NIOS 软核的 TCP 通信技术，以实现稳定可靠的高速数据通信。

2 TCP 通信技术开发流程

2.1 简介

TCP 是一种面向连接的、可靠的、基于 IP 的传输层协议，面向连接意味着两个使用 TCP 的应用在彼此交换数据之前必须先建立一个 TCP 连接。当应用层向 TCP 层发送用于网间传输的、用 8 位字节表示的数据流，TCP 则把数据流分割成适当长度的报文段，最大传输段大小（MSS）通常受该计算机连接的网络的数据链路层的最大传送单元（MTU）限制。之后 TCP 把数据包传给 IP 层，由它来通过网络将包传送给接收端的 TCP 层。

TCP 为了保证报文传输的可靠，就给每个包一个序号，同时序号也保证了传送到接收端的数据包能被按序接收。然后接收端对已成功收到的字节发回一个相应的确认(ACK)；如果发送端在合理的往返时延(RTT)内未收到确认，那么对应的数据（假设丢失了）将会被重传。

在数据正确性与合法性上，TCP 用一个校验和函数来检验数据是否有错误，在发送和接收时都要计算校验和。在保证可靠性上，采用超时重传和捎带确认机制。在流量控制上，采用滑动窗口协议，协议中规定，对于窗口内未经确认的分组需要重传。在拥塞控制上，采用 TCP 拥塞控制算法。

TCP 通信技术分为服务端和客户端，其通信过程大致如图 2.1 所示。

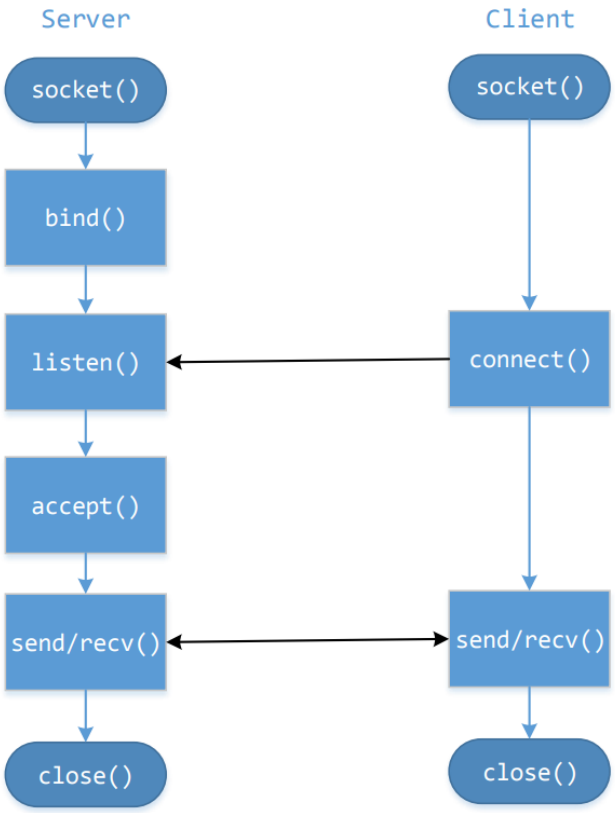

图 2.1 TCP 通信过程

整个过程可分为以下几个步骤：

TCP 服务端：

1）创建 TCP 套接字（使用 socket()函数）；

2）把本地协议地址绑定到套接字（使用 bind()函数）；

3）监听客户端的连接（使用 listen()函数）；

4）接收客户端的连接（使用 accept()函数）；

5）与客户端交互（使用 send()/recv()函数）；

6）关闭 TCP 服务（如果需要关闭服务），关闭 socket 描述符并退出（使用 close()函数）。

TCP 客户端：

1)创建 TCP 套接字（使用 socket()函数）；

2)建立与服务器的连接（使用 connect()函数）；

3)与服务器交互（使用 send()/recv()函数）；

4)关闭套接字描述符并退出（使用 close()函数）。

2.2 硬件设计

参考文档《开拓者 NiosII 开发指南_V1.2》中第二十一章于《基于 NicheStack 的简单 socket 服务器实验》。

根据 GSA_B 板上的 sdram 芯片手册进行相关参数修改以符合设计要求（参考文档《开拓者 NiosII 开发指南_V1.2》中第七章于《SDRAMIP 核》。），并添加 FIFOIP 完成硬件 verliog 与软核之间的数据通信，如图 2.2 所示。

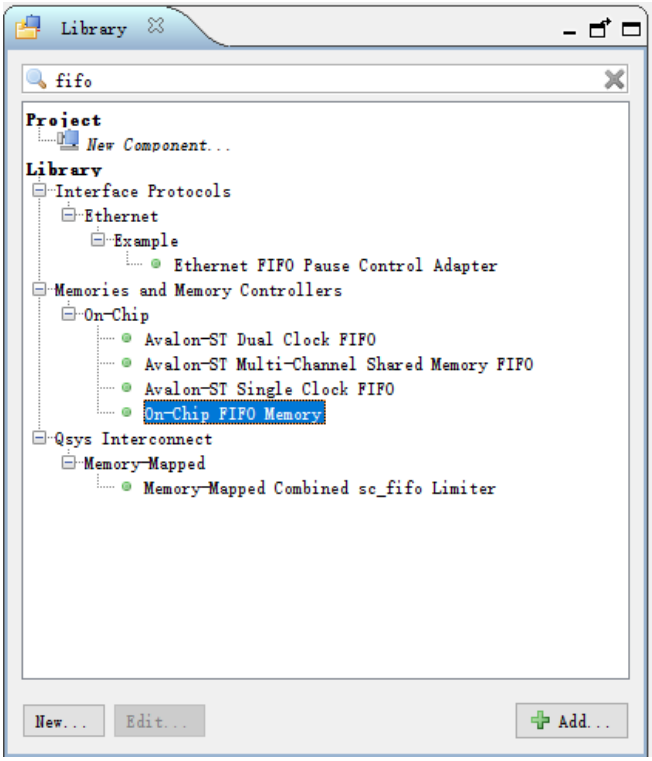

图 2.2 搜索 FIFO IP

我们选择 On-ChipFIFOMemory，即 FPGA 片上 FIFOIP 核。点击 Add 后，弹出图 2.3 所示界面。

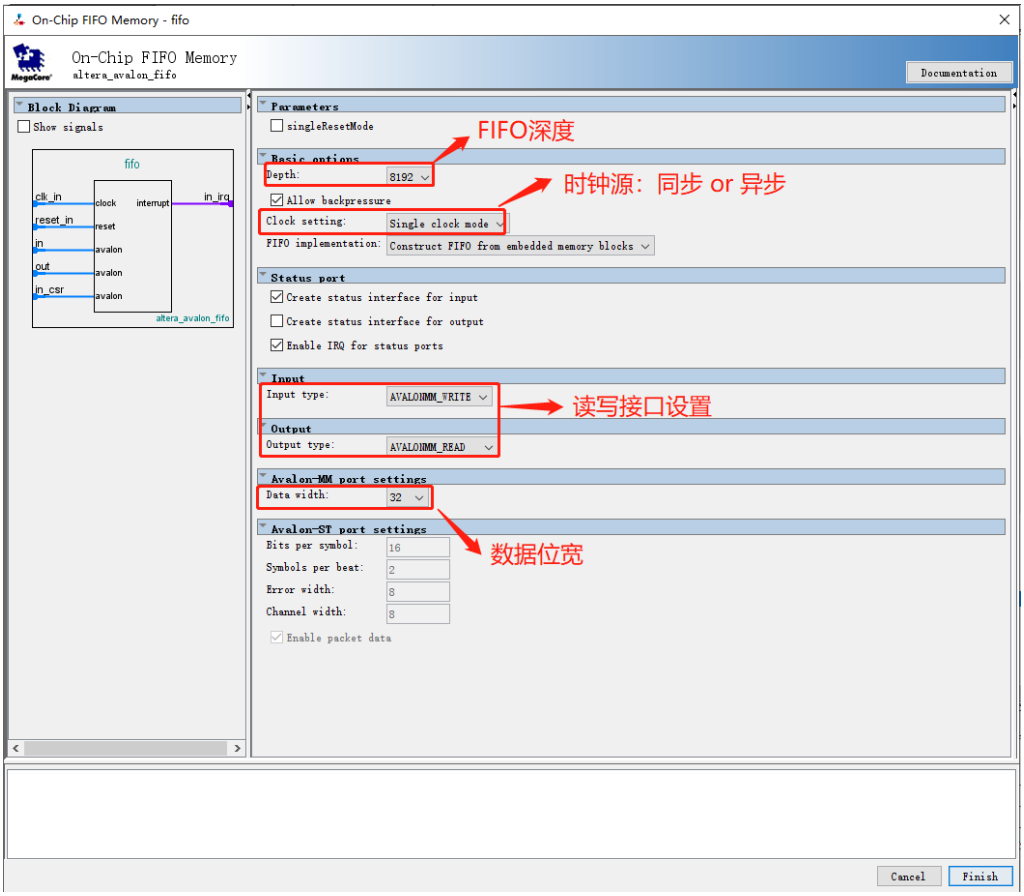

图 2.3 设置 FIFO 配置

2.3 软件设计

软件设计部分可以参考《开拓者 NiosII 开发指南_V1.2》第二十一章《基于 NicheStack 的简单 socket 服务器实验》进行 MAC 地址、IP 地址、端口号等参数设置，并在第二十四章《基于 NicheStack 的 TCP 客户端实验》的基础上进行修改实现具体功能。

选中文件 tcp_client.c，并将其内容替换如下：

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "includes.h"                 //MicroC/OS-II definitions
#include "simple_socket_server.h"     //Simple Socket Server definitions
#include "alt_error_handler.h"        //错误处理
/* Nichestack definitions */
#include "ipport.h"
#include "tcpport.h"

#include "altera_avalon_pio_regs.h"  //pio 头文件
```

```c
#include "altera_avalon_fifo_regs.h" //fifo 头文件
#include "altera_avalon_fifo_util.h"
#define ALMOST_EMPTY 2
#define ALMOST_FULL FIFO_IN_CSR_FIFO_DEPTH-5
volatile int input_fifo_wrclk_irq_event;

/*
 * sss_reset_connection()
 * 复位 SSSConn 结构体成员
 */
void sss_reset_connection(SSSConn* conn)
{
    memset(conn, 0, sizeof(SSSConn));

    conn->fd = -1;
    conn->state = READY;
    conn->rx_wr_pos = conn->rx_buffer;
    conn->rx_rd_pos = conn->rx_buffer;
    return;
}
/*
 * Hex_to_Dec()
 * 进制转换
 */
int Hex_to_Dec(INT8U hex_a,INT8U hex_b,INT8U hex_c,INT8U hex_d)
{
    int hex = 0;

    //最高位
    hex = hex + hex_a*16777216;

    //次高位
    if (hex_b < 0)
        hex = hex + (16777216+hex_b*65536);
    else
        hex = hex + hex_b*65536;

    //次低位
    if (hex_c < 0)
        hex = hex + (65536+hex_c*256);
    else
        hex = hex + hex_c*256;

    //最低位
```

```
    if (hex_d < 0)
        hex = hex + (256+hex_d);
    else
        hex = hex + hex_d;

    return hex;
}
/*
 * sss_handle_send()
 * 处理接收自客户端的指令信息，并向 verilog 硬件发送该指令；
 * 读取 FIFO 中数据并发送
 * 如果输入 stop 就结束连接
 */
void sss_handle_send(SSSConn* conn)
{
    const char stop[] = "stop";
    int i = 0;int data = 0;
    INT8U     tx_buffer[2912];

    while (conn->state != CLOSE)
    {
        memset(tx_buffer, 0, 2912); //清空 rx_buffer

        if(altera_avalon_fifo_read_level(FIFO_IN_CSR_BASE) >= 728)//检
查 FIFO 中有多少数据
        {
            for(i=0;i<2912;i+=4)
            {
                /*读取 FIFO 数据 32bit*/
                data =
altera_avalon_fifo_read_fifo(FIFO_OUT_BASE,FIFO_IN_CSR_BASE);
                /*32bit 转 8bit*/
                tx_buffer[i]  =((data)     & 0xFF);
                tx_buffer[i+1]=((data>>8)  & 0xFF);
                tx_buffer[i+2]=((data>>16) & 0xFF);
                tx_buffer[i+3]=((data>>24) & 0xFF);
            }
            send(conn->fd, tx_buffer, 2912, 0);
        }

        if (recv(conn->fd, conn->rx_buffer, SSS_RX_BUF_SIZE -1,
MSG_DONTWAIT) > 0)
        {
```

```
            IOWR_ALTERA_AVALON_PIO_DATA(PIO_INSTRUCTIONSH_BASE,
Hex_to_Dec(conn->rx_buffer[0],conn->rx_buffer[1],conn->rx_buffer[2],c
onn->rx_buffer[3]));//写入高 32bit 指令
            IOWR_ALTERA_AVALON_PIO_DATA(PIO_INSTRUCTIONSL_BASE,
Hex_to_Dec(conn->rx_buffer[4],conn->rx_buffer[5],conn->rx_buffer[6],c
onn->rx_buffer[7])); //写入低 32bit 指令

            if (strcmp(conn->rx_buffer, stop) == 0)
            {
                int cnt =
altera_avalon_fifo_read_level(FIFO_IN_CSR_BASE);//检查 FIFO 中有多少数据
                for(i=0;i<cnt;i++)
                    data =
altera_avalon_fifo_read_fifo(FIFO_OUT_BASE,FIFO_IN_CSR_BASE);//读取
FIFO 数据 32bit
            }
            memset(conn->rx_buffer, 0, SSS_RX_BUF_SIZE); //清空 rx_buffer
        }
    }
    printf("[sss_handle_receive] closing connection\n");
    close(conn->fd);
    sss_reset_connection(conn);

    return;
}
/*
 * SSSSimpleSocketServerTask()
 * 创建 MicroC/OS-II 任务，实现 TCP 客户端功能
 */
void SSSSimpleSocketServerTask()
{
        int sockfd;
        struct sockaddr_in servaddr;
        static SSSConn conn;

        //创建 TCP 套字节
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sockfd < 0)
        {
            printf("[sss_task] Socket creation failed \n");
            return ;
        }

        memset(&servaddr, '0', sizeof(servaddr));
```

```
        //构建服务器地址信息
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(32768);
        servaddr.sin_addr.s_addr = inet_addr("192.168.1.89");//32 位 IP
地址

        sss_reset_connection(&conn);//
        //客户端通过调用 connect 函数来建立与 TCP 服务器的连接。
        if (connect(sockfd,(struct
sockaddr*)&servaddr,sizeof(servaddr)) != 0)
        {
            printf("[sss_task] connect failed \n");
            return ;
        }

        conn.fd = sockfd;
        printf("connected to %s\n", inet_ntoa(servaddr.sin_addr));

        altera_avalon_fifo_init(FIFO_IN_CSR_BASE, 0,
ALMOST_EMPTY,ALMOST_FULL);//初始化 FIFO
        while (1)
        {
            sss_handle_send(&conn);
        }
}
```

经过以上修改替换后，软件设计部分就完成了。

2.4 下载验证

修改完软件工程，接下来我们就将该实验下载至 GSA_B 进行验证。

首先我们用一根网线将开发板和电脑进行连接，然后连接 JTAG 和电源，开发板上电后我们在 QuartusII 软件中将 nios_eth.sof 文件下载至我们的开拓者开发板，nios_eth.sof 下载完成后，我们就将 nios_eth.elf 文件系统下载至我们的开拓者开发板，下载完成后打开上位机软件 TCP_SDemo.exe，IP 地址和端口号设置如图 2.4 所示。点击“建立连接”按钮，当来与 FPGA 客户端连接成功后，指示灯变绿，此时点击“开始采集”按钮进行数据传输。

图 2.4 上位机软件设置

本次实验可实现 10Mbps 的稳定可靠的数据通信。

2.5 工程代码移植

代码移植过程，NIOS 软核需要进行相应修改否则工程无法正常运行。工程代码移植具体步骤如下：

1）拷贝现有工程到新目录；

2）打开 Quartues II 工程文件；

3）打开 NIOS II EDS 软件；

4）切换工作空间到当前的新工程根目录；

5）在 NIOS II EDS 软件中，将已有的软件工程先删除掉（不要勾选从硬盘上删除文件）；

6）新建软件工程或重新导入该工程根目录下的已有的软件工程文件；

7）修改 setting.bsp 文件中第七行和第九行的内容为当前工程的相应位置；

8）clean 当前工程并删除 mem_init 文件；

9）refresh 两个工程；

10）在工程 properties 中，将 app 工程和 bsp 工程关联起来。

2.6 工程代码固化

基于 NIOS 的 TCP 通信工程分为硬件部分和软件部分，因此固化的时候需要通过脚本将硬件代码部分（sof 文件）和软件代码部分（elf 文件）合并成统一的固化文件（jic 文件）进行代码烧写。具体步骤如下：

1）将 generate_jic.tcl、generate_jic.sh、generate_jic.cof 文件拷贝到 nios ii 软件工程下；

2）在 eclipse 中选中应用工程，右键->NIOS II->NIOS command shell；

3）NIOS command shell 中输入"./generate_jic.sh"；（运行完成后，会在 Quartus II 工程根目录下生成一个 myoutput_files 的文件夹，同时将 generate_jic.tcl、generate_jic.cof 文件拷贝到工程根目录下。）

4）在 quartus ii 中点击 Tools -> Tcl Scripts，选中 generate_jic.tcl，点击 run；（运行成功，会在 myoutput_files 目录下生成名叫 hs_combined.jic 的文件）

5）烧写 hs_combined.jic 到 FPGA 中，对板卡断电重新上电，新固件就可以开始运行了。

3 TCP 通信性能优化

3.1 优化方案

通过文献调研，提高 TCP 通讯传输性能有以下几种方案：

（1）使用具有快速存取时间和低延迟的内存

内存访问时间和延迟会对整个系统性能产生很大影响，为了实现更快的存储器访问，建议使用 FPGA 内部的存储模块或 SRAM 等快速片外存储器。

（2）增加指令缓存和数据缓存的大小

由于内存带宽通常对最大化系统性能至关重要，因此数据和指令高速缓存的大小会对性能产生重大影响。如前所述，典型地，TCP/IP 应用程序涉及许多数据拷贝和相同数据的操作。增加数据缓存大小可以最大限度地减少缓存未命中和内存访问导致的延迟。增加指令高速缓存可以提高性能，因为它改善了从内存中提取指令的延迟。Nios II/f 处理器具有可配置的指令和数据缓存，允许设计人员选择每个应用所需的大小。如果主程序和数据存储器具有快速访问时间，那么增加高速缓存大小可能不会显著提高整体性能。

（3）提高时钟频率

某些架构决策会对时钟频率重大影响。例如，通过隔离从 FPGA 到以太网 MAC/PHY 设备的数据路径，可以获得更快的时钟。另一方面，如果电路板设计为 MAC/PHY 器件与其它器件共享数据、地址和控制线，则 FPGA 内部会有一个宽多路复用器。这种宽多路复用器使得难以在保持快速时钟频率的同时对设计进行布局布线。因此，为了获得更好的性能，设计系统时应使以太网接口在 FPGA 和 MAC/PHY 设备之间有专用的 I/O 信号。

（4）TCP/IP 堆栈优化

通常，TCP/IP 协议栈的实现为设计人员提供了配置协议栈的灵活性。这允许设计者修改实现以适合设计的要求。例如，轻量级 IP(lwIP) TCP/IP 堆栈有许多可以修改以提高性能的参数。通常，性能的代价是更大的内存占用。

3.2 实验验证

针对 TCP 性能优化的方案，在 GSA-B 电路板上进行了实验验证，测试结果如表 1-1 所示。

表 1-1 不同工况下的 TCP 传输性能比较

NIOS 主频	内存	指令缓存	数据缓存	传输性能
100MHz	片上 RAM(368kB)	4kB	16kB	18Mbps
100MHz	外部 SDRAM	4kB	16kB	10Mbps
100MHz	外部 SDRAM	64kB	64kB	18Mbps
120MHz	外部 SDRAM	64kB	64kB	21Mbps
130MHz	外部 SDRAM	64kB	64kB	23Mbps

通过分析表中数据可以发现，同一 NIOS 主频下，使用 FPGA 内部存储器（片上 RAM）的 TCP 传输性能要优于使用外部 SDRAM；在使用外部存储器 SDRAM 时，随着指令缓存和数据缓存空间的增大，TCP 传输性能有了明显提升。此外，在使用 SDRAM 时，当 NIOS 主频从 100MHz 增大到 130MHz，传输速率呈线性提高趋势。因此理论上通过继续提高主频可以实现更高的传输性能。

3.3 实验遇到的问题

（1）虽然轻量级 IP(lwIP)TCP/IP 堆栈有许多可以修改以提高性能的参数，但是其相关移植方法都是基于 STM32，与 NIOS 相差较大，移植起来较为复杂。因此，在本次实验中未能移植成功，无法评估其对传输性能的影响。

（2）当使用外部 SDRAM 作为 NIOS 内存时，随着 NIOS 主频的增大，整个系统的稳定性有所下降，例如，当 NIOS 主频为 130MHz 时，偶尔会出现“跑飞”现象。