

OpenHW2012开源硬件大赛

项目论文

项目名称：基于 ZED-Board 实现的宽带实时自适应均衡器

学校名称：中国科学院电子学研究所

指导老师：吕晓德

团队成员：赵永科、蒋柏峰、李纪传、龚黎明

电子邮箱：zhaoyongke@yeah.net

联系电话：010-58887103 18201017305

网址：www.openhw.org

视频展示链接：

http://v.youku.com/v_show/id_XNTczODM1MzUy.html

目录

OpenHW2012 开源硬件大赛	1
项目论文	1
1. 前言	1
2. 总体方案设计	1
2.1 均衡算法选择	1
2.2 HDL 实现方式选择	4
2.3 裸机系统与 Linux 系统选择	4
2.4 PS 与 PL 接口选择	4
2.5 系统总体结构	5
3. PL 上的硬件设计	7
3.1 GAL 在 PL 上的实现	7
3.1.1 编码及位宽	8
3.1.2 多节 GAL 级联	11
3.1.3 硬件初步优化	12
3.1.4 硬件深度优化	16
3.2 AXI-Stream IP 设计	18
3.2.1 初探 AXI	18
3.2.2 AXI-Stream 接口说明	19
3.2.3 my_stream_ip 设计实例	21
3.2.4 总线与算法模块整合	30
4. PS 上的软件设计	34
4.1 基于 Linux 的驱动程序设计	34
4.1.1 AXI_DMA 模块接口	34
4.1.2 驱动程序模块入口与出口	36
4.1.3 驱动程序文件接口	37
4.1.4 驱动程序小结	41
4.2 基于 Qt 的应用程序设计	41
4.3 NLMS 算法模块	43

5. 系统调试和测试	49
5.1 平台测试	49
5.2 GAL 算法调试	51
5.3 目前性能指标	53
6. 总结	53
7. 参考文献	54

1. 前言

目前,无线通信技术进入了其有史以来发展最快的时期,3G、4G 等宽带移动通信技术正在逐步走进我们的生活。通信系统发展到 3G 后,几十甚至上百 Mbps 的宽带数据传输速率对接收端信号处理技术是个巨大的挑战。在 2G 时代,几乎所有移动终端处理器都采用 DSP 通用处理器实现信号处理,但其传统顺序执行的 CPU 工作模式难以胜任当前高速实时信号处理工作。FPGA 具有大规模并行处理能力,可以出色地完成上述大规模、大数据量的计算,相比 DSP 可以获得更高的计算性能。

Zynq-7000 系列是世界上第一套完全可编程的片上系统,该系列产品融合了一颗工业标准 ARM 双核 Cortex™-A9 MPCore™ 处理系统与 Xilinx 28nm 通用可编程逻辑架构。该完全嵌入式处理平台以处理器为核心,具有 ASIC 级别的低功耗,FPGA 的灵活性以及微处理器的易于编程特性。ZED-Board 是 Xilinx Zynq™-7000 可扩展处理平台中的一块低成本开发板,该板包含了创建基于 Linux, Android, Windows 或其他操作系统/实时操作系统的设计的所有必需内容。本文利用 Xilinx 提供的 ZED-Board 实现了通信中常用的 GAL-NLMS 均衡算法,是一套具备高效、实时性的宽带自适应均衡器,对当前和今后的宽带通信具有很强的实用价值。

2. 总体方案设计

2.1 均衡算法选择

宽带通信最大的挑战是使用各种方法克服信道不理想对原信号造成的破坏,尽可能恢复发送信号,实现无失真传输。在带宽受限(频率选择性)且时间扩散的信道中,由于多径影响而导致的符号间干扰会使被传输的信号产生失真,从而在接收机中产生误码。符号间干扰被认为是在无线信道中传输高速率数据时的主要障碍,而均衡正是克服符号间干扰的一种技术。均衡器可分为两类:带训练序列的均衡器和不带训练序列的均衡器(盲均衡器)。盲均衡器需要对信号有先验知识,如具有恒包络性质的信号可用 CMA 盲均衡器,该类均衡器只适合特定的信号,不具有普遍意义,本文不做深入讨论,只关注带训练序列的均衡器。由于移动信道具有随机性和时变性,要求均衡器必须能够实时地跟踪移动通信信道的时变特性,因此这种均衡器又称为自适应均衡器。通信系统使用自适应均衡器的框图如图 2-1 所示。

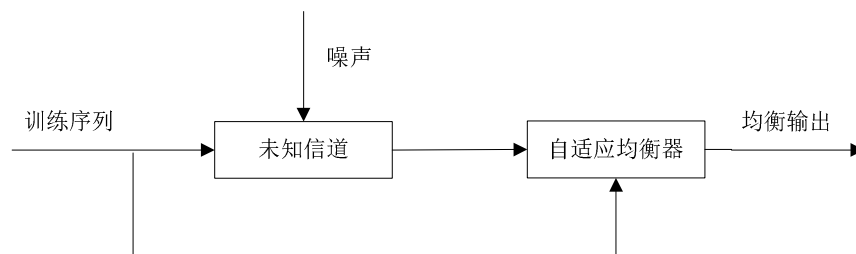


图 2-1 自适应均衡器框图

自适应均衡器一般包括两种工作模式,即训练模式和跟踪模式。首先,发射机发射一个已知的、定长的训练序列,以便接收机中的自适应均衡器可以调整恰当的权值,使均衡输出

与训练序列的均方误差最小（MMSE 准则）。典型的训练序列是一段二进制伪随机信号（PN 序列）或是一串预先指定的数据比特，而紧跟在训练序列后被传送的是用户数据。接收机中的自适应均衡器将通过迭代算法来评估信道特性，修正滤波器系数以对多径造成的失真做出补偿。均衡器要求在最差信道条件下（如最快车速移动、最长时延扩展、深度衰落等）也能通过训练序列收敛到适当的滤波系数，这样可以保证在接收用户数据时，均衡器的自适应算法就可以跟踪不断变化的信道。当均衡器得到很好的训练后，就说它已经收敛。

针对本系统要求，有三种均衡方案可供选择。

方案一：LMS 均衡算法具有较小的计算量和简单的结构，是较为常用的自适应均衡器算法；其缺点是收敛慢。

方案二：RLS 均衡算法有较快的收敛速度，但计算量很大且数值特性不好，需要在双精度浮点处理器上才能使均衡器权值很好的收敛。

方案三：GAL-NLMS 算法克服了 LMS 收敛慢的缺点，输入数据首先进行正交化处理，然后送入 NLMS 训练器，可以很快收敛到最优权值。

本系统为宽带系统，在进行均衡时需要在较短时间内完成系数训练，需要快速收敛的特性，而方案一不满足这一点，方案二需要双精度运算，不适合在 FPGA 上实现。方案三兼顾了算法复杂度与均衡性能，且 GAL 对有限字长效应不敏感，更适合在 FPGA 上采用并行计算、数据流水线处理方式实现高效均衡。因此本文选择方案三。该算法的结构如图 2-2 所示。

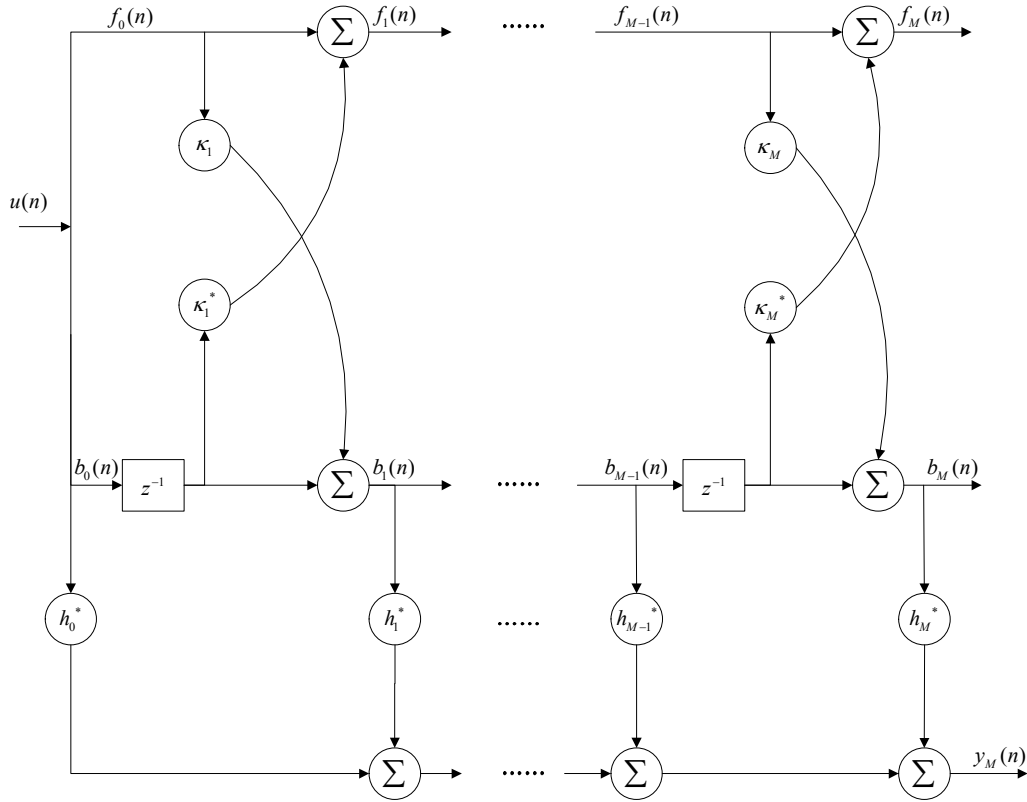


图 2-2 GAL-NLMS 算法框图

上图中， $u(n)$ 为均衡器输入信号； $f_i(n)$ 为第 i 阶前向预测误差信号， $b_i(n)$ 为第 i 阶后向预测误差信号；利用后向预测误差信号可以合成 $y_M(n)$ 估计给定的期望信号 $d(n)$ （即图 2-1

中的已知训练序列）。根据 GAL-NLMS 滤波器性质^[7]，各级后向预测误差 $b_i(n)$ 相互正交，

从而实现了输入数据前后级解耦合, $b_i(n)$ 相当于 $u(n)$ 在 M 维正交基上的投影, 利用这些投影来逼近期望信号, 具有更好的逼近性能。

GAL-NLMS 迭代更新算法如公式(1)~(8)所示, 算法在硬件上的实现见本文第三章。

表 1 GAL-NLMS 算法流程

初始化参数:

M =最终预测阶次; β 范围(0,1);

$\hat{\mu} < 0.1$; δ : 很小的正常数; a : 很小的正常数。

对所有预测阶次 $m=1,2,\dots,M$, 令 $f_m(0)=b_m(0)=0$,

$\xi_{m-1}(0)=a, \hat{\kappa}_m(0)=0, \hat{h}_m(0)=0$;

对所有时间索引 $n=1,2,\dots$ 令 $f_0(n)=b_0(n)=u(n), y_{-1}(n)=0$,

$\|\mathbf{b}_{-1}(n)\|^2 = \delta$;

多级格型预测器 (GAL 部分) 迭代:

$$\xi_{m-1}(n) = \beta \xi_{m-1}(n-1) + (1-\beta)(|f_{m-1}(n)|^2 + |b_{m-1}(n-1)|^2) \quad (1)$$

$$f_m(n) = f_{m-1}(n) + \hat{\kappa}_m^*(n-1)b_{m-1}(n-1) \quad (2)$$

$$b_m(n) = b_{m-1}(n-1) + \hat{\kappa}_m(n-1)f_{m-1}(n) \quad (3)$$

$$\hat{\kappa}_m(n) = \hat{\kappa}_m(n-1) - \frac{\hat{\mu}}{\xi_{m-1}(n)} [f_{m-1}^*(n)b_m(n) + b_{m-1}(n-1)f_m^*(n)] \quad (4)$$

期望响应估计器 (NLMS 部分) 迭代:

$$y_m(n) = y_{m-1}(n) + \hat{h}_m^*(n)b_m(n) \quad (5)$$

$$e_m(n) = d(n) - y_m(n) \quad (6)$$

$$\|\mathbf{b}_m(n)\|^2 = \|\mathbf{b}_{m-1}(n)\|^2 + |b_m(n)|^2 \quad (7)$$

$$\hat{h}_m(n+1) = \hat{h}_m(n) + \frac{\hat{\mu}}{\|\mathbf{b}_m(n)\|^2} b_m(n)e_m^*(n) \quad (8)$$

2.2 HDL 实现方式选择

本系统中，PL 硬件设计有三种方案可以选择：

方案一：采用 ISE 设计，基于传统 FPGA 的硬件描述语言方法设计，开发难度较大，对硬件优化的余地较大。

方案二：采用 Vivado HLS 将 System C 转化为 HDL 语言的设计方式，开发难度一般，不需要对 HDL 有了解，硬件优化空间很小。

方案三：采用 System Generator 结合 Matlab Simulink 搭建模块设计系统，不需要了解 HDL 编程，开发难度最小，仿真方便快捷，硬件优化空间小。

其中方案二和方案三是比较快捷的开发方式，但自动生成的 HDL 优化起来较为困难，硬件效率不高，而且总线接口不够灵活，不能满足我们系统的要求，因此这里采用方案一，采用 Verilog HDL 和 VHDL 混合语言设计 IP，并与 PS 部分通过 AXI 总线互联，实现较高的传输效率。

2.3 裸机系统与 Linux 系统选择

软件设计方式有两种方案可以选择：

方案一：采用裸机系统进行软件开发，不需要操作系统支持，对硬件可以进行直接访问，执行效率较高。缺点是不支持用户界面，访问文件系统、网络的能力有限，自己从底层开发难度较大，周期较长。

方案二：采用 Digilent 公司提供的 ZedBoard_OOB_Design，将自定义 IP 添加进 OOB 工程，并编写相应的驱动程序，实现在 Linux 下对硬件进行控制，并可以通过移植 QT 实现用户图形界面，支持鼠标、键盘、U 盘、HDMI 显示器等设备，支持网络文件系统。这样带来好处是用户体验较好，但开发难度也较大，主要因为涉及到 Linux 系统相关的内容，如果没有 Linux 基础，开发起来也比较费时。

由于本项目组成员拥有较好的 Linux 基础，之前有过开发 Linux 驱动的经验，所以这里选择了方案二。而方案一并没有彻底放弃，而是用于算法的初期调试和验证，尤其是调试 AXI 通信接口时，由于直接操作硬件，可以排除操作系统对 DMA 的影响，容易成功。在这之后将软件移植到 Linux 下，专门针对 Linux 系统进行改动，可以大大缩短调试时间，推进项目进度。

2.4 PS 与 PL 接口选择

在 PS 与 PL 部分进行通信时，有三种接口可供选择：

方案一：AXI-Lite 接口。该接口适合传输少量数据传输，占用硬件资源较少。

方案二：AXI4 接口，适合大批量数据传输，占用硬件资源较多。

方案三：AXI-Stream 接口，适合大批量、连续数据传输，占用硬件资源较少。

综合成本，资源状况等因素，我们采用性价比最高的 AXI-Stream 接口来实现 PS 与 PL 数据传输。

2.5 系统总体结构

经过前面论证，各方案最终将在 Zynq-7000 SoC 上实现。作为算法实现的核心，Zynq-7000 SoC 由 PS 与 PL 两部分构成，内部资源^[9]如表 2 所示：

表 2 ZYNQ-7000 SoC 资源列表

PS 部分：

多功能 ARM 双核 Cortex-A9 (PS)，采用 28nm 架构；
NEON 协处理器，支持向量与标量浮点计算（乘、加）；
256KB 片上双口 SRAM；
DDR3 接口；
千兆网口；
USB2.0 接口；
SD 卡接口；
I2C, SPI, CAN, UART 等接口；

PL 部分：

XC7Z020 与 Artix-7 系列 FPGA 相同；
总计 85K Logic Cells；
LUTs:53200；
FF:106400；
36Kb BRAM: 560KB；
220 个 DSP Block:18X25 MACs；
对称 FIR 峰值 DSP 性能：158GMACs；
12bit XADC 双通道，最高采样率 1MSPS；
集成 PCIE 接口（7030,7045），速率 2.5Gbps/5Gbps；
36Kb Block RAM（双口，72bit 宽，可配置为双 18bit）；

可编程 I/O 块：
$$\begin{cases} LVC MOS / LVDS / SSTL \\ 1.2V \sim 3.3V \\ \text{可编程 } I / O \text{ 延迟和 } SerDes \end{cases} ;$$

支持 JTAG Std 1149.1；

串行收发器：12.5Gb/s X16 收发器；

安全模块：AES, SHA 256b 用于 bootloader 和 PL 配置加密；

前面论述的 GAL-NLMS 算法分为两个步骤：GAL 部分和 NLMS 部分，GAL 部分的结构采用格型结构，具有对称性和模块化的特点，对有限字长效应不敏感，适合并行实现；而 NLMS 部分计算量较小，仅涉及乘累加以及减法运算，数据动态范围很大，权值更新对数据精度要求较高，适合采用双精度浮点处理器实现。从上表看出 PS 部分由双核 ARM Cortex-A9, FPU 等计算资源构成，适合做高精度、串行算法实现，本项目用于实现 NLMS 更新部分；而 PL 部分包含 220 个 18X25bit 的 DSP48E1 计算核心，适合做定点运算，对于精度要求不高但有并行性的 GAL 算法较为适合，本项目最终确定 GAL 部分在 PL 上实现。

解决了 PS 与 PL 的任务划分问题，剩下的问题就是如何在 PS 与 PL 之间实现高性能通

信。当数据带宽为 8MHz 时, 30 阶 GAL-NLMS 算法的数据吞吐率要求达到 7.68Gbps, 一般的外部总线互联方案不能满足要求, 而 ZYNQ-7000 创新性地实现将 PS 与 PL 集成在一个芯片内, 通过 AXI 总线进行高速双向数据交互, 具有设计成本降低、设计整体功耗降低、设计体积减小、设计风险降低、设计更具灵活性等优点^[3]。

设计高效的 PL 与 PS 数据交互通路是 Zynq 芯片设计的重中之重, 是产品成败的关键。在参考了为了达到项目需求, 结合图 2-3^[5]的接口理论带宽值, 可以确定采用 AXI_HP 或 AXI_ACP 接口作为数据通道, 这样单通道可实现 $1.2\text{GBps} \times 8 = 9.6\text{Gbps}$ ($>7.68\text{Gbps}$) 的单向传输带宽, 满足项目需求。

Interface	Type	Bus Width (bits)	IF Clock (MHz)	Read Bandwidth (MB/s)	Write Bandwidth (MB/s)	R+W Bandwidth (MB/s)	Number of Interfaces	Total Bandwidth (MB/s)
General Purpose AXI	PS Slave	32	150	600	600	1,200	2	2,400
General Purpose AXI	PS Master	32	150	600	600	1,200	2	2,400
High Performance (AFI) AXI_HP	PS Slave	64	150	1,200	1,200	2,400	4	9,600
AXI_ACP	PS Slave	64	150	1,200	1,200	2,400	1	2,400
DDR	External Memory	32	1,066	4,264	4,264	4,264	1	4,264
OCM	Internal Memory	64	222	1,779	1,779	3,557	1	3,557

图 2-3 ZYNQ 互联接口理论带宽

项目最终采用 AXI_HP0 接口作为数据通道, 同时使能 PS 部分的 AXI-GP0 作为控制链路, 用于配置 PL 部分的 AXI_DMA 模块。

AXI_DMA 模块是 Xilinx EDK、CORE Generator、Vivado Design Suite 中的软核 IP, 该 DMA 引擎提供内存映射存储器与 AXI4-Stream 型外设之间高带宽 DMA, 可以大大解放 CPU 的数据搬运任务^[10]。AXI_DMA 模块负责将 PS 内存中的输入数据转为 Stream 流输出, 交给位于 PL 部分的 GAL_Module, 并将处理结果通过 Stream 流收回, 搬运回 PS 内存。

PS 部分运行 Linux 系统, 其中 GalEqLz 驱动程序负责将 PL 部分的数据打包给应用程序, 并接受应用程序控制命令 (如打开设备、关闭设备、控制设备等)。应用程序有 NLMS 模块负责将 GAL 的输出做进一步处理, 实现 NLMS 权值更新并对有效数据进行滤波, 实现整体均衡。QT 界面用于接受用户输入并显示结果。

为了便于同用户交互, PS 安装了 USB 鼠标和键盘, HDMI 接口输出接显示器来显示用户图形界面, 文件系统采用 RAMDISK (系统启动) 和 SD 卡 (存储应用程序和数据) 结合的方式, 具有很高的性价比。本项目总体方案如图 2-4 所示。

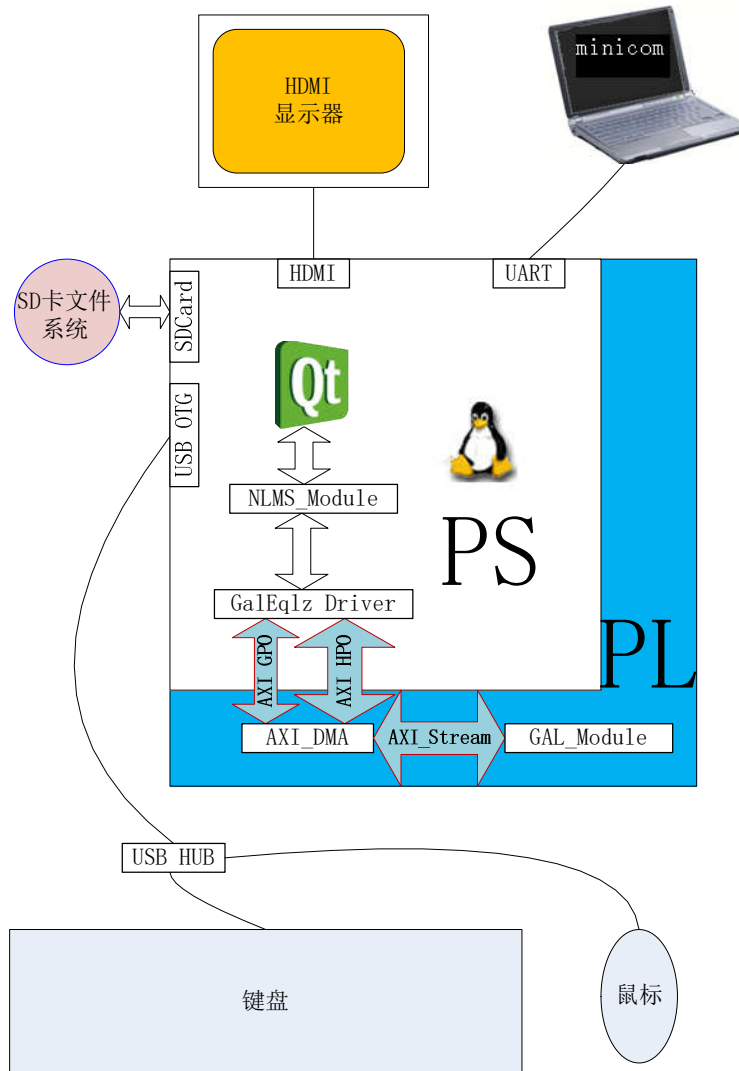


图 2-4 系统总体方案

经过近三个半月的程序调试和不断硬件、软件优化，本项目的最终版本程序在 ZED-Board 板卡上运行成功（见视频）。

3. PL 上的硬件设计

本章主要介绍 PL 端硬件设计，具体分为 GAL 算法实现、AXI_Stream IP 实现这两部分。

3.1 GAL 在 PL 上的实现

首先回顾 GAL 算法更新公式 (1) ~ (4)，其中 (4) 式是求解反射系数 $km(n)$ 的公式，(2) (3) 式是求解前向和后向预测误差的公式。为了便于简称并与代码中的称谓对应，我们将 (4) 式中的分子带乘累加的部分称为 sigema ，分母称为 squar_add ，整个 $km(n)$ 叫做 kappa 。很明显，在输入数据都是复数的情况下， sigema 也是复数，而 squar_add 则是实数， kappa 为复数。

图 3-1 是基本的 GAL-NLMS 算法图示，很明显，多级格型滤波器中的每一格都是一个基本单元（红框所示），每个基本单元在结构上是一致的，该单元接收上一级的前向预测误差连续输入，我们称之为 fn_in , bn_in ，等接受完 n 点之后，会求解出本级的 $kappa$ ，然后利用这个 $kappa$ ，再利用 fn_in , bn_in 求解该级的后向预测误差，我们称为 fn_out , bn_out 。单从结构图上来看，我们用 FPGA 设计时的直觉就是：先设计一阶 GAL，然后不断例化，再首尾相连，最终得到我们需要的多级 GAL。但是在把这个想法付诸行动时，有一些问题我们先要解决。

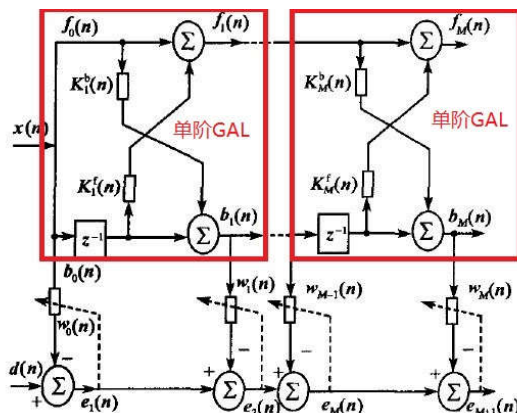


图 3-1 基本 GAL-NLMS 结构图

3.1.1 编码及位宽

GAL 算法受限于 PL 硬件，实现起来有以下难点：

- 1、输入数据带有小数，FPGA 只能处理整数，所以需要将小数转化为整数才能输入；
- 2、输入数据动态范围大，采用 FPGA 实现受限于资源只能使用有限字长，有限字长效应会带来量化误差；
- 3、输入数据是复数，包含实部跟虚部两个部分，运算复杂度高；
- 4、数据跟运算都是有符号数，而 FPGA 一般意义上都是处理无符号数，对于有符号数需要进行补码编码之后才能处理。

基于以上这些问题，我们在写一阶 GAL 之前必须对输入数据进行处理才行，而在 verilog 实现过程中，需要有相当的技巧才能避免各种字长效应。我们经过研究和仿真之后，最终做了如下处理：

- 1、确定输入输出字长

经过仿真，我们确定采用 16bit 来量化输入输出的实部和虚部是足够的。16bit 的最高 bit 是符号位，低 15bit 是数据位。这样， fn 和 bn 这些复数可以用 32bit 表示出来，其中[31:16]位是实部，[15:0]位是虚部，实部虚部都带符号。确定了输入数据的位宽，抱着运算不溢出的基本原则再配合编码方案，sigema, squar_add 以及 $kappa$ 的位宽都可以确定。

- 2、对输入数据进行深度归一化，然后进行 Q15 编码。

确定了输入的位宽为实部和虚部各 16bit，就要想办法让输入数据适应这个位宽。我们的处理是：首先对输入数据进行归一化，除以各数中最大的绝对值，使输入全部变成小于 1 的数，然后乘以 2^{15} （2 的 15 次方），再用 MATLAB 转换为补码，输入给 FPGA。这种将输入数据归一化然后乘以 2^n 的处理方案称为 Qn 编码。

实际处理中，我们在归一化时除了除以各数中最大的绝对值之外，还除以 1.5，原因在于求解 fn_out , bn_out 的公式带有加法，存在溢出可能，为了让输出控制在 16 位之内，有

必要让输入更小一些。

3、确定其余数据的位宽和编码

Q 型编码遵循以下准则：Qm 编码乘以 Qn 编码，得到的结果是 Q (m+n) 编码，相除则是 Q (m-n) 编码；如果两种编码的数想要做加法或减法，需要先移位，使得编码一致之后才能计算。据此，我们确定 sigema 为 Q24 编码，32bit；squar_add 为 Q15 编码，26bit；kappa 为 Q15 编码，16bit。这种确定遵循以下几个原则：

- (a) 位宽必须满足数据范围，保证数据不溢出；
- (b) 加法两端的输入编码应该一致，否则还要统一编码才能送入加法器；
- (c) 乘法要根据精度需求对输出截尾。乘法器的输出如果不截尾的话，长度会是两个输入的长度之和，编码也是两者之和。这样每算一次乘法结果的位宽就会越来越长，这是不行的，对乘法器输出截尾可以降低输出字长，并且降低编码；截尾应当舍去低位，从最高符号位开始保留，具体截多少位合适，需要考虑截尾后精度会损失多少；
- (d) 最大限度保障 kappa 精度。kappa 是该算法的关键，它的精度是对误差求解影响很大的，求解 kappa 的最后一步运算是除法，xilinx 的除法器 IP 核采用 radix2 算法最大只支持 32 位除法，我们有必要让 32 位全部用上，所以 sigema 的位宽是确定的，据此可以反推其他数据的编码和位宽。除法器理论上也可以采用 high radix 算法的 IP 核，该算法支持更高位的除法，但是消耗的 DSP48E1 资源太多，不利于我们做到高阶。

在确定了编码及位宽之后，我们实现了一阶 GAL。该模块的接口如下：

```
module my_gal(
    clk, ce, reset,  fn_in, fn_out,  bn_in, bn_out ,refresh_begin    //该算法只完成单级 GAL
)
    input  clk;                //接入全局时钟
    input reset;              //ce 是使能端，当 ce 为 1 时，输入才能被接收
    input ce;                 //使能，当上一级的误差开始更新的时候，本级开始使能，所以 ce 连接到上一级的 refresh_begin

    input [31:0] fn_in;       //前向预测误差输入
    input [31:0] bn_in;       //后向预测误差输入
    output [31:0] fn_out;     //前向预测误差更新后的输出
    output [31:0] bn_out;     //后向预测误差更新后的输出
    output refresh_begin;     //误差更新即将输出的信号线
```

该模块的内部实现原理如图 3-2 所示。

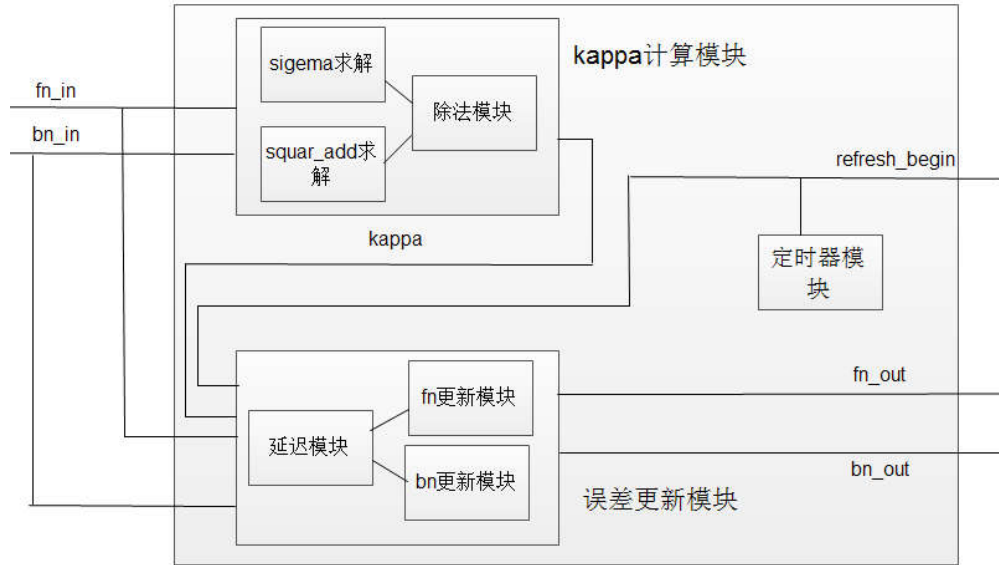


图 3-2 GAL 单节框图

fn_in 和 bn_in 为上一级的预测误差输入，这个时候 $kappa$ 计算模块开始计算，定时器模块开始计时，当定时器值为 648 的时候， $kappa$ 计算出来，并且从此保持不变，此时 $refresh_begin$ 信号线被拉高为 1。接下来误差更新模块开始工作，接收 fn_in 和 bn_in ，并且利用刚刚计算出来的 $kappa$ 进行本阶误差更新的计算并产生 fn_out 和 bn_out 。

该算法的 verilog 代码中，因为都是有符号数运算，所以都调用 IP 核。单阶的 GAL 算法要用到 16bit*16bi 的复数乘法 IP，各种位宽相加的 IP 以及 32bit/26bit 的 IP。调用 IP 核的时候，要注意 IP 的不同输入端数据是否同时到达。比如，求解 $kappa$ 的时候，最后一步除法要通过仿真确定 $sigema$ 和 $squar_add$ 是否同时到达，经测试， $squar_add$ 要早到达 3 个 clock，所以需要手动调节求解 $squar_add$ 的 IP 核延时，延缓 3 个 clock，使除法器的分子分母同步。误差更新模块也面临这个问题： $bn_out=bn_in+kappa*fn_in$ ，加法器两端一个是直接输入，另一个是复数乘法的输出，复数乘法 IP 核在使用 3 个 DSP48E1 的时候延时是 6 个 clock，所以需要对 bn_in 也做 6 个 clock 的延时才能保证加法器的两个操作数同步。

该算法有一个缺点在于：因为没有对输入数据进行保存，所以第一次输入一帧数据只能用来训练 $kappa$ ，此时误差更新模块尚未工作；等 $kappa$ 算完， $refresh_begin$ 被拉高， $kappa$ 求解模块停止工作，误差更新模块开始工作，这个时候需要把刚才那一帧再输入一次，才能求出本阶的预测误差输出。

不管怎样，我们已经有了一阶的 GAL 模块，综合一下，得到如下资源使用情况：

Device Utilization Summary (estimated values)				
Logic Utilization	Used	Available	Utilization	
Number of Slice Registers	7517	106400	7%	
Number of Slice LUTs	3021	53200	5%	
Number of fully used LUT-FF pairs	2641	7897	33%	
Number of bonded IOBs	132	200	66%	
Number of BUFG/BUFFCTRLs	1	32	3%	
Number of DSP48E1s	9	220	4%	

图 3-3 一阶 GAL 模块资源占用情况

可以看到，单阶 GAL 需要用 9 个 DSP48E1 单元，这是因为我们用了 3 个复数乘法 IP。最高时钟可以达到 176MHZ。

3.1.2 多节 GAL 级联

我们延续自己一开始的设计思想：先设计单阶 GAL 模块，然后将其按照原理图那样首尾相连，构成高阶。再看看资源使用情况，一阶的 LUT-FF 使用了 33%，似乎我们只能实现 3 阶（IOB 不用看，因为实际上最后的输出不会通过引脚引出，而是通过内部 AXI 总线传给 PS）。既然如此，我们就写一个三阶模块，接口如下：

```
module gal_cascade(
    Gal_clk, Gal_ce, Gal_reset,
    Gal_in, bn_out1, bn_out2, bn_out3,
    Gal_refresh_begin
);
input Gal_clk;
input Gal_ce;
input Gal_reset;
input [31:0] Gal_in;    //输入端
output [31:0] bn_out1;  //后项预测误差输出
output [31:0] bn_out2;
output [31:0] bn_out3;
output Gal_refresh_begin; //信号线，为高表明误差更新开始输出
综合之后，得到 RTL 如图 3-4 所示。
```

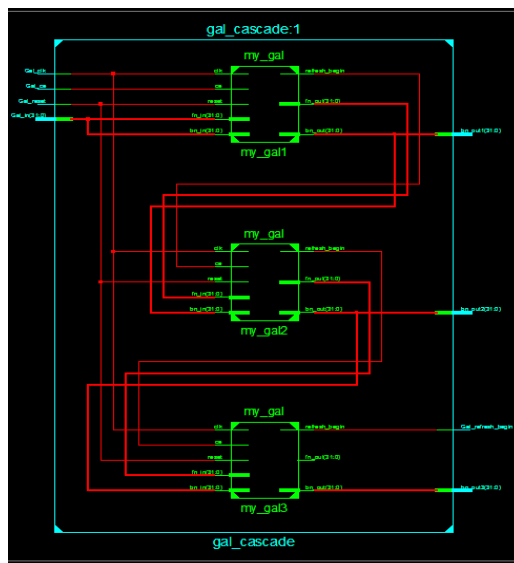
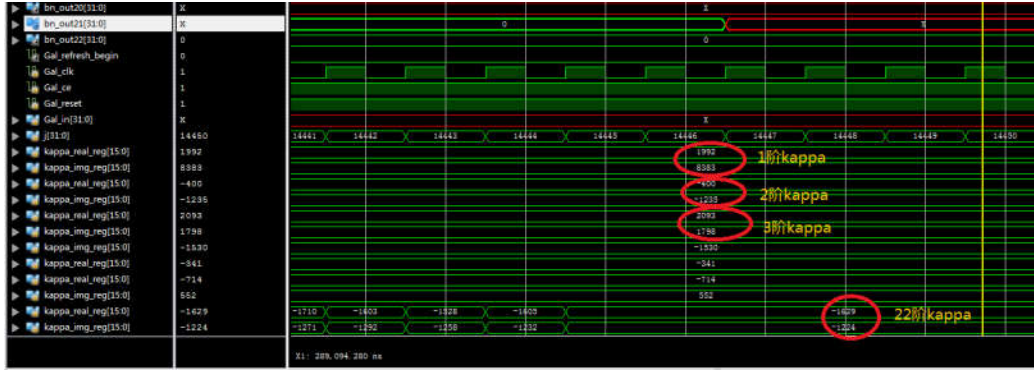


图 3-4 三阶 GAL 综合后 RTL 图

可以看到，很好的实现了我们的级联策略。上一级的输出连接到下一级的输入。有意思的地方在于，该模块综合之后，资源使用量并非单阶 GAL 的 3 倍。除了乘法器变成了 3 倍之外，其余的几乎没有增加。但是该模块的实现却用了 2 天 2 夜。

虽然仅仅 3 阶就用了 2 天 2 夜来实现，但是我们仍然写了 22 阶。单阶 GAL 消耗 9 个 DSP48E1，zynq-7020 总共才 220 个，所以理论极限就是 24 阶。我们成功的综合了一下，并且进行了仿真，原理图仍然跟上面一样，只不过是 22 个首尾相连。

下图是我们仿真得到的每阶的 kappa 值，为 Q15 编码：



将其与 MATLAB 的结果做个对比：

阶数	kappa_real	kappa_img	FPGA仿真的kappa	matlab的kappa
1	1992	8383	0.06079+0.2558i	0.0608 + 0.2558i
2	-400	-1235	-0.01221-0.03769i	-0.0121 - 0.0377i
3	2093	1798	0.06387+0.05487i	0.0639 + 0.0549i
4	-341	-1530	-0.0104-0.0467i	-0.0103 - 0.0468i
21	-714	552	-0.0218+0.016845i	-0.0219 + 0.0170i
22	-1629	-1224	-0.04971-0.03735i	-0.0496 - 0.0374i

可以看到误差都小于 1%，在功能上，我们已经完成了这个算法，写到这里似乎要完工了。但是有一个很不好的消息：22 阶 GAL 在 LUT 的资源消耗上达到了 150%！其他很多资源也都超了 100%，因此根本就实现不了这种高阶的算法！更不要说实现起来需要跑多少天了。怎么办？下面对硬件进行优化。

3.1.3 硬件初步优化

解决资源使用过多的问题，需要我们再次仔细研究 GAL 算法。之前我们的设计是根据结构图来看的，结构图很明显的是由多个单阶 GAL 首尾相连得到高阶 GAL 的。接下来，我们将从数据流来分析，找到新的突破口。

我们每阶 GAL 都有两个模块，一个负责求解 kappa，一个负责预测误差更新。两个模块其实是顺序工作的，只有该阶的 kappa 求解出来，该阶的误差更新模块才能正常工作；而只有该阶的误差更新出来了，下一阶的 kappa 才能求解。所以不仅单阶 GAL 内部，阶与阶之间都是顺序的过程。在训练阶段，数据的流程图如图 3-5 所示。



图 3-5 训练阶段流程图

这是一个流水线的过程，在某一时刻，上一阶的误差更新开始的时候，下一阶 kappa 求解将会使能，所以每一时刻都有两个模块在同时工作。不过，每阶的 kappa 求解模块在解出 kappa 之后，都会停止工作，kappa 以后也会保持不变，只有误差更新模块一旦被启用，以后会一直工作，所以每阶 GAL 其实都存在着浪费。

我们从结构上看到的是单阶 GAL 级联构成 22 阶 GAL；我们从模块功能上去看，其实

可以看出，不管多少级 GAL，只有两种模块，一种训练 κ ，一种进行误差更新。训练 κ 其实是个顺序的过程，只有上一级的 κ 训练出来，下一级的 κ 才可能训练出来。基于此，我们想到了新的设计方案，对 κ 训练模块进行分时复用。只例化误差更新模块。经过深入思考，我们发现有三种分时复用的方法，各有优缺点，它们分别为：

（1）无预置 κ 的单阶 GAL 分时复用

单阶 GAL 内部包含一个求解 κ 的模块和一个进行误差更新的模块。我们对输入的一帧数据用 BRAM 进行保存，等该阶的 κ 求解出来了，再将保存的数据输入给误差更新模块，误差更新会在 8 个 clock 之后输出，此时将这些输出写入 BRAM 中。等这一阶的误差更新算完了，再将 BRAM 中的数据输入给 κ 求解模块求解下一阶的 κ ，依次往下。用数据流来看，是下面的过程：



图 3-6 κ 求解次序之一

在中间的某一时刻 BRAM 同时进行读写操作，输入和输出仅仅相差 8 个 clock，整个数据流构成一个环路，像一条贪吃蛇一样。这种设计的缺点在于：上一阶的误差更新和下一阶的 κ 求解不再是同时进行的，因此训练阶段所需的时间要比之前的设计增大一倍才能训练出所有的 κ 。

（2）预置 κ_1 的单阶 GAL 分时复用

上面讨论的是用 BRAM 寄存单阶 GAL 的输入，假如我们寄存单阶 GAL 的输出的话，将会得到另外一种方案，这种方案必须预先知道 κ_1 。此时，输入的数据先进行一阶误差更新，然后用 BRAM 寄存一阶误差更新输出，同时这些输出会输入给 κ 求解模块，求解二阶 κ 。等二阶 κ 求解完了，将 BRAM 中的数据输入给误差更新模块，同时利用刚求解的二阶 κ ，得到二阶误差更新，依次往下。用数据流来看，是下面的过程：



图 3-7 κ 求解次序之二

该方案的优点在于，能同时进行某一阶的误差更新和下一阶的 κ 求解，因此训练阶段的速度能提高一倍，缺点在于必须知道一阶 κ 。可是实际使用中， κ 都是训练出来的，产生数据的环境不同 κ 就不同，不可能预先知道，所以从可行性上需要斟酌。

（3）用两个单阶 GAL 实现倍速训练

从方案（2）很明显可以得到推论，如果我们方案（2）前面增加一个单阶 GAL 模块，让它求解 κ_1 ，就可以实现倍速训练。这种方案的缺点就在于使用资源增加了很多，而且让前面的单阶 GAL 只求解 κ_1 是很浪费的。

经过权衡，我们采用了方案（1）。基于此，我们对之前设计的单阶 GAL 略微做了修改，增加了 κ 的输出以及内部清 0，同时在外部分时复用的调度层，构成新的模块，称之为训练模块，接口如下：


```

module new_train_module(           //训练模块
clk,ce,reset,train_in,gal_refresh_begin,
kappa1,kappa2,kappa3,kappa4,kappa5,kappa6,
kappa7,kappa8,kappa9,kappa10,kappa11,kappa12,
kappa13,kappa14,kappa15,kappa16,kappa17,kappa18,
kappa19,kappa20,kappa21,kappa22,kappa23,kappa24,
kappa25,kappa26,kappa27,kappa28,kappa29,kappa30,
kappa31,kappa32
);
input  clk;
input  reset;
input  ce;
input  [31:0] train_in;
output [31:0]           //输出 32 个 kappa
kappa1,kappa2,kappa3,kappa4,kappa5,kappa6,
kappa7,kappa8,kappa9,kappa10,kappa11,kappa12,
kappa13,kappa14,kappa15,kappa16,kappa17,kappa18,
kappa19,kappa20,kappa21,kappa22,kappa23,kappa24,
kappa25,kappa26,kappa27,kappa28,kappa29,kappa30,
kappa31,kappa32 ;
output gal_refresh_begin;    //信号线，标志着 kappa 训练完毕，误差更新开始

```

从代码上看，很容易看到，我们采用分时复用之后，目的是想设计出 32 阶的 GAL 算法。这对于之前连 22 阶都做不到的情况，无疑是很大的提升。该训练模块作用在于根据一帧的输入数据，直接训练出 32 阶 kappa，并且输出给更新模块。该模块内部例化了单阶 GAL，但是在它上层加了复杂的时序调度。我们先看原理图：

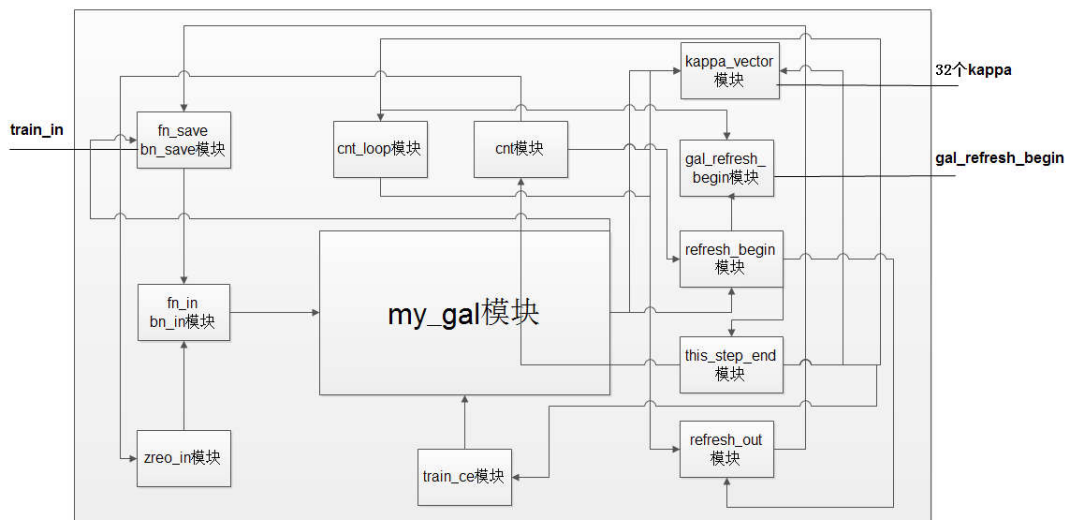


图 3-8 训练模块框图

原理图的复杂性很高，主要在于里面的时序调度很复杂。大致的工作过程是这样的：
train_in 是训练模块的输入，输入的一帧数据首先被 fn_save 和 bn_save 模块保存起来，同时保存的数据通过 fn_in,bn_in 模块输入给 my_gal 模块。fn_in,bn_in 模块在输入整个训练模块未使能，或者系统复位的时候，输入为 0。fn_in,bn_in 模块在输入一帧数据之后，my_gal

模块经过 648 个时钟周期完成 κ 训练, refresh_begin 信号线被拉高, 意味着即将进行误差更新。此时 fn_save 模块和 bn_save 模块会将保存的数据传输给 my_gal 内部的更新模块, 经过 8 个 clock 之后, 误差更新就会输出, 此时 refresh_out 信号线拉高, fn_save 和 bn_save 模块再保存误差更新的输出。经过一帧数据输入完后, 该阶的 κ 和误差更新都有了, 此时 this_step_end 信号线会被拉高, 标志该阶运算完毕, 即将进入下一阶的周期。此时 cnt_loop 会加 1, 表示阶数加 1, 而计时器 cnt 会清 0, 重新计数。同时 train_ce 模块是控制 my_gal 使能的, 在当前阶结束的时候, my_gal 模块会重新使能一下, 目的是清除 my_gal 模块内部的数据, 以便进入下一阶计算。同时, kappa_vector 这个寄存所有 κ 的模块会在此时永久记录下当前阶计算出来的 κ 。接下来进入下一阶, 重复以上过程。

让我们看一下该训练模块综合之后的结果:

Device Utilization Summary (estimated values)				[1]
Logic Utilization	Used	Available	Utilization	
Number of Slice Registers	46643	106400	43%	
Number of Slice LUTs	56830	53200	106%	
Number of fully used LUT-FF pairs	41802	61671	67%	
Number of bonded IOBs	1060	200	530%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number of DSP48E1s	9	220	4%	

图 3-9 训练模块综合后结果

可以很明显看到, 仅仅当前的训练模块就消耗了 106% 的 LUT 资源! 可是我们的 my_gal 模块仅仅消耗了 7% 的 LUT, 可见加了分时复用的时序调度之后, 模块的复杂性增加了很多。可是我们还有更新模块没有写! 下面是我们的更新模块, 它是从 my_gal 里面分离出来的。

```

module fnbn_refresh( clk,reset,           //更新模块, 完成单阶的误差更新
    fn_in,bn_in,kappa,
    fn_out,bn_out
);
    input clk;
    input reset;
    input [31:0] fn_in;           //预测误差输入
    input [31:0] bn_in;
    input [31:0] kappa;          //kappa 从训练模块那里得到
    output [31:0] fn_out;
    output [31:0] bn_out;
    reg [31:0] fn_out;            //预测误差输出
    reg [31:0] bn_out;

```

至此, 训练模块和更新模块都有了, 我们已经可以写出 32 阶的 GAL 算法了。32 阶 GAL 是基于我们之前介绍的训练模块和误差更新模块写出来的。我们添加了一个最顶层模块, 主要工作在于完成训练模块和 32 个更新模块的互连。整个系统的工作原理是: 首先接收系统输入端发送一帧数据, 然后等待 Gal_refresh_begin 信号线被拉高, 紧接着就可以发送连续的滤波序列。相比于未优化前的设计, 旧的设计在模块内部没有存储机制, 我们需要对训练序列反复发送, 才能得到多个 κ , 但是如今只需要一帧输入即可, 大大减轻了 PS 部分的工作量。而且新的代码极易扩展, 理论上, 只要让其一直运行, 无论多少阶 κ 都可以求出来, 只不过这里没有引出更高阶的 κ 罢了。

以下是其结构图:

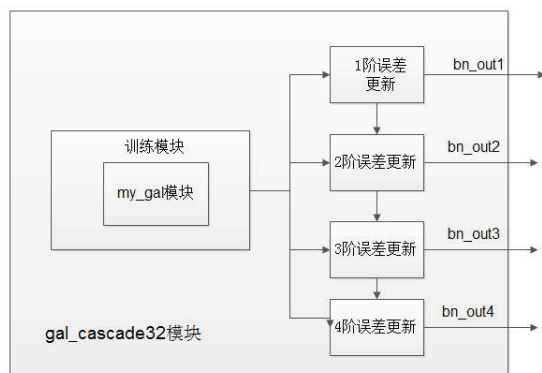


图 3-10 32 阶 GAL 算法框图

误差更新模块总共被例化了 32 个，上图只画了 4 个。这些更新模块首尾相连，并且使用训练模块得到的 32 个 kappa。32 阶 GAL 算法综合的结果如图 3-11 所示。

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	55860	106400	52%
Number of Slice LUTs	82434	53200	117%
Number of fully used LUT-FF pairs	46855	71439	65%
Number of bonded IOBs	1091	200	545%
Number of BUFPG/BUFCTRLs	1	32	3%
Number of DSP48E1s	201	220	91%

图 3-11 32 阶 GAL 算法资源占用情况

可以很明显看到，乘法器几乎已经消耗殆尽，而 LUT 已经超量 17%！在之前介绍训练模块的时候，已经说到单分时复用的训练模块就已经消耗了 106% 的 LUT！更为困难的是，zynq-7020 内部模块本身会消耗 19% 的 LUT，所以实际可用的 LUT 只有 81%！这样看来，训练模块似乎不能用，如果不能用，那么我们的优化就是失败的，因为我们连一阶 GAL 都无法成功实现，甚至还不如没优化之前的算法，它至少能实现 3 阶。

3.1.4 硬件深度优化

采用 ISE 设计，基于传统 FPGA 的硬件描述语言方法设计代码好处之一就在于自己可以直接操作底层硬件，对硬件实现深度优化。很明显我们必须对训练模块进行优化，否则该模块就不能使用。这次的优化涉及到对很多代码的改写，主要的优化策略在于：尽可能减小每个 always 模块的大小，尽可能减小 if 嵌套的深度，尽可能避免重复条件，尽可能减小单变量的依赖变量。以下是一个例子：

```

/* always @(posedge clk) begin    //几条信号线
    if(refresh_begin == 1) begin
        if(cnt >= train_delay)
            refresh_out <= 1;
        else
            refresh_out <= 0;
        if (cnt == frame_width + train_delay)
            this_step_end <= 1;
        else
            this_step_end <= 0;
    end
end

```

```

        end
    else begin
        refresh_out <=0;
        this_step_end <=0;
    end
end
*/
always @(posedge clk) begin    //refresh_out 信号模块
    if(refresh_begin ==1 && cnt >=train_delay)
        refresh_out <=1;
    else
        refresh_out <=0;
end
always @(posedge clk) begin    //this_step_end 信号模块
    if(refresh_begin ==1 && cnt ==frame_width+train_delay)
        this_step_end <=1;
    else
        this_step_end <=0;
end

```

这是一段摘取的代码，被注释掉的那一段是以前的代码，该代码控制两条信号线。下面两段代码是优化后的代码，优化的方案就是将每条信号线单独写入一个 `always` 模块，这样每个 `always` 模块都很小，占用的资源就会很少。尽管逻辑功能上两者是一样的，但是 ISE 明显对它的综合结果是不一样的。如此一番将代码拆散重新改写之后，我们再次进行综合，得到如下的结果：

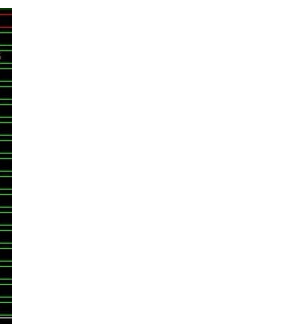
Device Utilization Summary (estimated values)				[1]
Logic Utilization	Used	Available	Utilization	
Number of Slice Registers	8499	106400	7%	
Number of Slice LUTs	4223	53200	7%	
Number of fully used LUT-FF pairs	3634	9088	39%	
Number of bonded IOBs	1060	200	530%	
Number of Block RAM/FFIO	2	140	1%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number of DSP48E1s	9	220	4%	

图 3-12 32 阶 GAL 算法深度优化后资源占用情况

LUT 的使用居然降到了 7%！这是连我们自己都没有想到的，优化居然可以这么成功。随着我们的代码越来越长，资源的消耗却越来越少，LUT 和 DSP48E1 都尚有富余。于是我们情不自禁思考一个问题：zynq-7020 的片上资源用来做 GAL 算法的极限在哪里？

GAL 算法由训练模块和更新模块构成：训练模块只有一个，消耗 9 个 DSP48E1；更新模块每阶消耗 6 个 DSP48E1。zynq-7020 总共只有 220 个 DSP48E1。简单一算，就知道该算法的极限在 35 阶，于是 35 阶 GAL 算法就是我们的目标。

因为已经写过 32 阶的 GAL 算法，我们重复了复制粘贴的机械过程，很快得到了 35 阶的 GAL。综合一下，结果如下：



行一次性读写。上面两种均采用内存映射控制方式，即 PS 端将用户自定义 IP 编入其某一内存地址进行访问，读写时就像在读写自己的片内 RAM，编程也很方便，开发难度较小，但代价是资源占用过多，需要额外的读地址线、写地址线、读数据线、写数据线、写应答线，而且传输速度受限（主要是因为采用 AXI-GP 物理接口，带宽很低）。

第三种 AXI 接口就是 AXI-Stream，这是一种连续传输的接口技术，速度能达到非常高（150MHz 数据时钟，32/64bit 数据位宽，最高可达 9.6Gbps 通信带宽），而且不需要地址线（有点像 FIFO，主机可以一直读或一直写）。这类 IP 不能通过上面的内存映射方式控制，必须有一个转换装置，例如 AXI-DMA 模块就能实现内存映射到流式接口的转换，但编程较复杂，调试起来没有内存映射方式直观，必须要通过芯片内部调试接口（Chipscope）来观察。AXI-Stream 适用的场合有很多：视频流处理；通信协议转换；数字信号处理；无线通信等。其本质都是针对数值流构建的数据通路，从信源（例如 PS 内存、DMA、无线接收前端等）到信宿（例如 HDMI 显示器、音频输出等）构建起连续的数据流，这种接口适合做实时信号处理^[11]。当然，实际信号处理中也有分块和不分块的情况，典型分块情况就是计算 FFT，而典型不分块的情况就是计算 FIR 滤波输出。

后面两种接口（PLB，FSL）貌似在 ZED 上面用处不大，都是 Microblaze 的接口。不过应该也有桥接 IP，实现协议转换，但本文对此没有研究。

本项目属于典型的流式数据，从射频前端、ADC 采集到信号传输到 DDR2 内存，组织为时分复用或并行通路来传输数值数据到自定义 IP，并携带额外信息（均衡器的系数），利用控制流通道传输过去。处理结果仍传回 DDR2 中，交给主机显示或存储为文件。与 DDR2 的通信需要借助 AXI-HP 物理接口，PL 部分为 master，负责数据搬移。

通过以上论述，应该比较清楚的看到整个数据走向了，具体实施细节还需要进一步研究。

3.2.2 AXI-Stream 接口说明

AXI4-Stream 协议是一种用来连接需要交换数据的两个部件的标准接口，它可以用于连接一个产生数据的主机和一个接受数据的从机。当然它也可以用于连接多个主机和从机。该协议支持多种数据流使用相同共享总线集合，允许构建类似于路由、宽窄总线、窄宽总线等更为普遍的互联。AXI4-Stream 接口的信号线定义如图 3-15 所示^[13]。比较重要的信号线有：

ACLK 为时钟线，所有信号都在 ACLK 上升沿被采样；

ARESETn 为复位线，低电平有效；

TVALID 为主机数据同步线，为高表示主机准备好发送数据；

TREADY 为从机数据同步线，为高表示从机准备好接收数据；这两根线完成了主机与从机的握手信号，一旦二者都变高有效，数据传输开始。

TDATA 为数据线，主机发送，从机接收。

TKEEP 为主机数据有效指示，为高代表对应的字节为有效字节，否则表示发送的为空字节。

TLAST 为主机最后一个字指示，下一 clk 数据将无效，TVALID 将变低。

TID，TDEST，TUSER 均为多机通信时的信号，这里不涉及，不予考虑。

Signal	Source	Description
ACLK	Clock source	The global clock signal. All signals are sampled on the rising edge of ACLK .
ARESETn	Reset source	The global reset signal. ARESETn is active-LOW.
TVALID	Master	TVALID indicates that the master is driving a valid transfer. A transfer takes place when both TVALID and TREADY are asserted.
TREADY	Slave	TREADY indicates that the slave can accept a transfer in the current cycle.
TDATA[(8n-1):0]	Master	TDATA is the primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.
TSTRB[(n-1):0]	Master	TSTRB is the byte qualifier that indicates whether the content of the associated byte of TDATA is processed as a data byte or a position byte.
TKEEP[(n-1):0]	Master	TKEEP is the byte qualifier that indicates whether the content of the associated byte of TDATA is processed as part of the data stream. Associated bytes that have the TKEEP byte qualifier deasserted are null bytes and can be removed from the data stream.
TLAST	Master	TLAST indicates the boundary of a packet.
TID[(i-1):0]	Master	TID is the data stream identifier that indicates different streams of data.
TDEST[(d-1):0]	Master	TDEST provides routing information for the data stream.
TUSER[(u-1):0]	Master	TUSER is user defined sideband information that can be transmitted alongside the data stream.

图 3-15 AXI-Stream 总线信号

看到这里，可能大家都还对 Stream 没有一个直观的认识。其实 Stream 并不陌生，在我们学 c++编程时，一定会包含<iostream>，这样就可以完成控制终端对程序的输入输出了。如果还是不够直观，想象一下连续不断的水流，永远向着一个方向以固定的速度输送的接口。以我们看视频为例，视频文件本来是保存在硬盘里的，怎么播放呢，不能一下子把整个文件都显示到屏幕上，而是以一定的速度，连续不断地输出到屏幕上（每秒 30~60 帧），这个过程就是流 Stream 接口完成的。

Xilinx 提供的流式 IP 核有很多用途，可以实现音频流、视频流、数据流到内存或者相反方向的传输。有人问了，内存是 PS 控制的，怎么能把 PS 里 DDR2 的内容以 Stream 形式发出去呢（例如以固定速度送往 DA，完成信号发生器的设计）？答案就是利用 AXI 总线做转换。ZYNQ 的 PS 部分是 ARM Cortex A9 系列，支持 AXI4，AXI-Lite 总线。PL 部分也有相应 AXI 总线接口，这样就能完成 PS 到 PL 的互联。仅仅这样还不够，需要 PL 部分实现流式转换，即 AXI-Stream 接口实现。Xilinx 提供的从 AXI 到 AXI-Stream 转换的 IP 核有：AXI-DMA，AXI-Datamover，AXI-FIFO-MM2S 以及 AXI-VDMA 等。这些 IP 核可以在 XPS 的 IP Catalog 窗口中看到。

AXI-DMA：实现从 PS 内存到 PL 高速传输高速通道 AXI-HP 到 AXI-Stream 的转换；

AXI-FIFO-MM2S：实现从 PS 内存到 PL 通用传输通道 AXI-GP 到 AXI-Stream 的转换；

AXI-Datamover：实现从 PS 内存到 PL 高速传输高速通道 AXI-HP 到 AXI-Stream 的转换，只不过这次是完全由 PL 控制的，PS 是完全被动的；

AXI-VDMA：实现从 PS 内存到 PL 高速传输高速通道 AXI-HP 到 AXI-Stream 的转换，只不过是专门针对视频、图像等二维数据的。

除了上面的还有一个 AXI-CDMA IP 核，这个是由 PL 完成的将数据从内存的一个位置搬到另一个位置，无需 CPU 来插手。这个和我们这里用的 Stream 没有关系，所以不表。

这里要和大家说明白一点，就是 AXI 总线和接口的区别。总线是一种标准化接口，由数

据线、地址线、控制线等构成，具有一定的强制性。接口是其物理实现，即在硬件上的分配。在 ZYNQ 中，支持 AXI-Lite, AXI4 和 AXI-Stream 三种总线，但 PS 与 PL 之间的接口却只支持前两种，AXI-Stream 只能在 PL 中实现，不能直接和 PS 相连，必须通过 AXI-Lite 或 AXI4 转接。PS 与 PL 之间的物理接口有 9 个，包括 4 个 AXI-GP 接口和 4 个 AXI-HP 接口、1 个 AXI-ACP 接口，均为内存映射型 AXI 接口。

上面的 IP 是完成总线协议转换，如果需要做某些处理（如变换、迭代、训练……），则需要生成一个自定义 Stream 类型 IP，与上面的 Stream 接口连接起来，实现数据输入输出。用户的功能在自定义 Stream 类型 IP 中实现。

3.2.3 my_stream_ip 设计实例

下面讲一个例子，来加深对上面介绍内容的理解。笔者使用的软件版本为 ISE 14.2。

1. 建立 PlanAhead 工程，一直到进入 XPS，具体流程见官方文档 CTT^[12]。
2. 在 XPS 中，添加一个 AXI-DMA 模块，配置界面如图 3-16 所示。

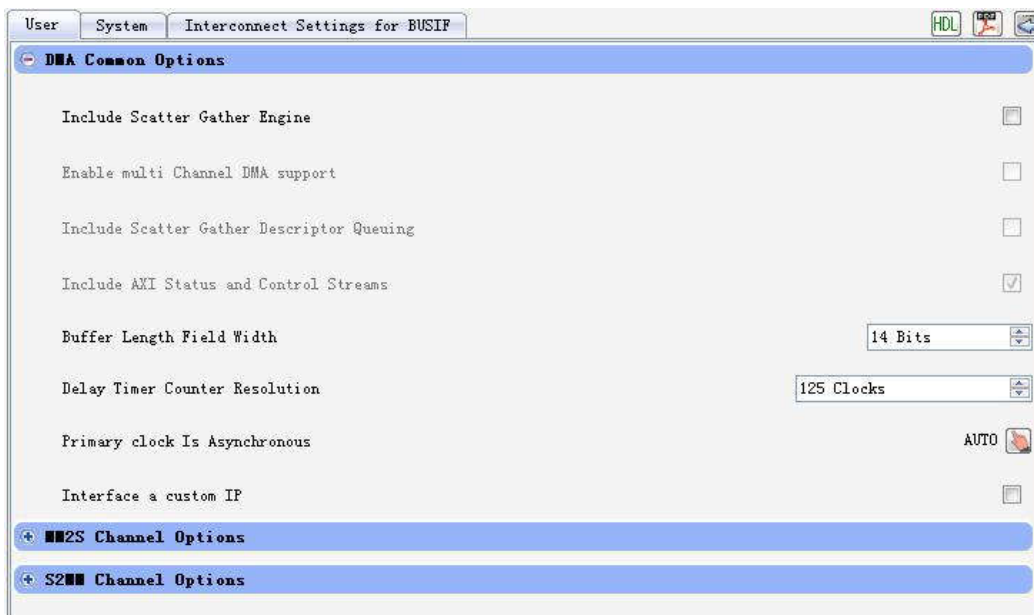


图 3-16 AXI-DMA 模块配置

其余参数默认。SG 模块如果选上，那么后面软件控制会相对复杂一些。这里不选，采用 Simple 模式，实现较为简单的传输。

3. 选菜单 Hardware->Create or Import Peripheral..., 设计自定义 IP。名称起为 my_stream_ip，自动版本为 1.00a。遇到 Bus Interface 选择 AXI4-Stream 类型，一直点下一步到最后结束。该类型 IP 的生成过程比 AXI4-Lite 和 AXI4 都要简单。

4. 添加一个 my_stream_ip 到系统中，连接见图 3-17。

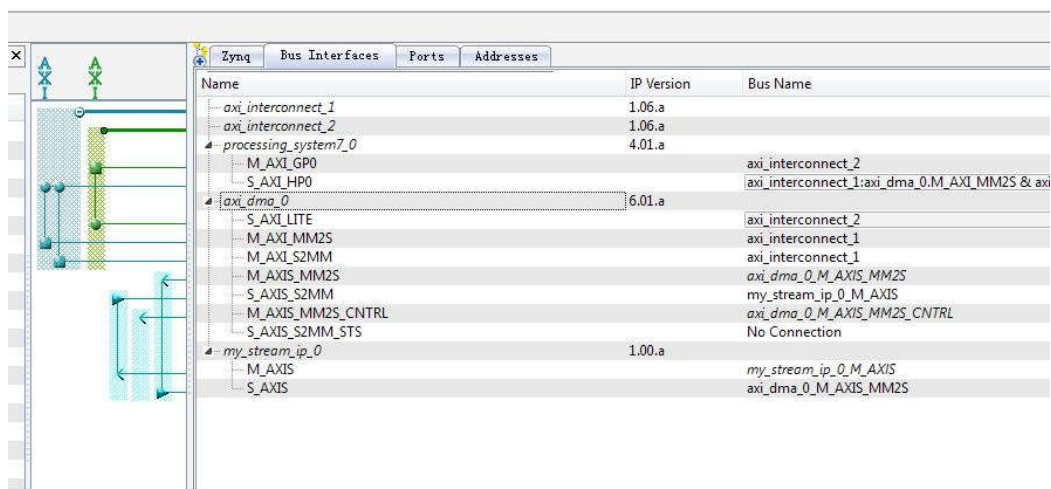


图 3-17 AXI Stream IP 硬件连接

由 XPS 自动生成的 my_stream_ip 实现了先接收 8 个 32bit 字，然后求和，再将结果发送回去（连续发送 8 次）。上图连接方式说明是 AXI-DMA 模块发送数据给 my_stream_ip，然后 my_stream_ip 又将结果发回 AXI-DMA。同时看到 AXI-DMA 和 PS 的数据流连接是通过 HP0 传输，而控制流通过 GP0 传输。

5. 上面连接在不做任何改动的情况下有问题（主要是 XPS 的 bug），需要一项项手动修改。首先是 HP0 的地址区间报错，可以先点 Zynq 标签，然后单击 HP0 绿线，在弹出的配置对话框中将 HP0 的地址区间改为我们 ZED Board 上 DDR2 区间 0x00000000~0x1FFFFFFF，像图 3-18 一样。

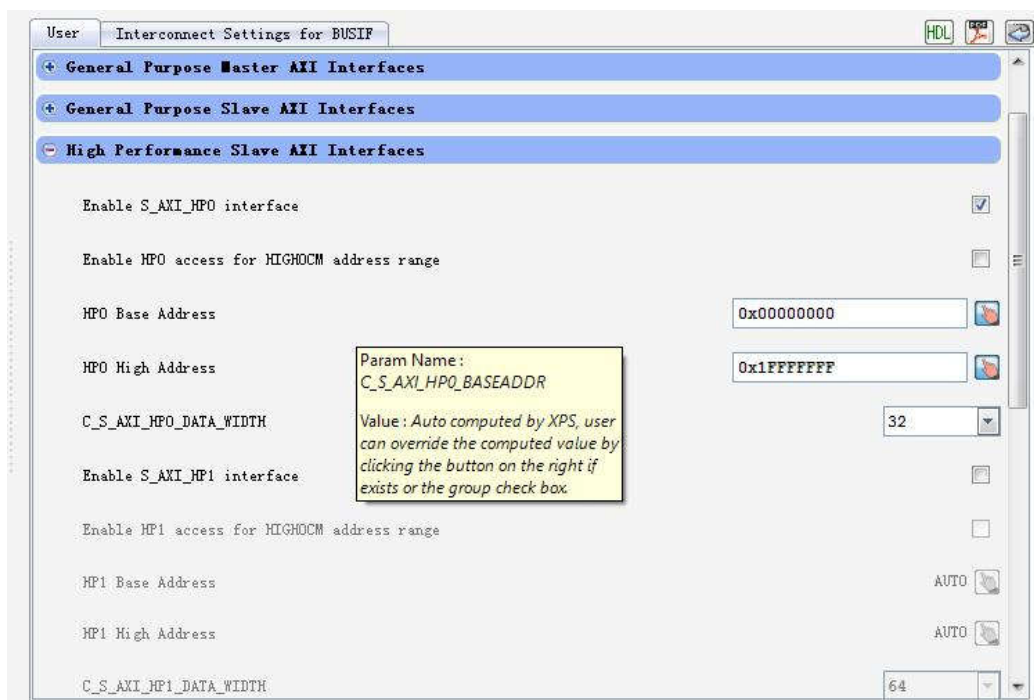


图 3-18 修正 bug1

在较高版本软件 ISE14.5 中，这个 bug 已经修复，不需要改。

第二个 bug 就是 AXI-DMA 和 my_stream_ip 的连线问题。本来都是 Stream 接口，按理说是标准接口，不应该有差异。但事实就是这样，XPS 界面掩饰的问题层出不穷。我们右击

my_stream_ip, 选择 View MPD, 将内容改为:

```
BEGIN my_stream_ip
## Peripheral Options
OPTION IPTYPE = PERIPHERAL
OPTION IMP_NETLIST = TRUE
OPTION HDL = VERILOG
## Bus Interfaces
BUS_INTERFACE BUS=M_AXIS, BUS_STD=AXIS, BUS_TYPE=INITIATOR
BUS_INTERFACE BUS=S_AXIS, BUS_STD=AXIS, BUS_TYPE=TARGET
## Parameters
PARAMETER C_S_AXIS_PROTOCOL = GENERIC, DT = string, TYPE = NON_HDL,
ASSIGNMENT = CONSTANT, BUS = S_AXIS
PARAMETER C_S_AXIS_TDATA_WIDTH = 32, DT = integer, TYPE = NON_HDL,
ASSIGNMENT = CONSTANT, BUS = S_AXIS
PARAMETER C_M_AXIS_PROTOCOL = GENERIC, DT = string, TYPE = NON_HDL,
ASSIGNMENT = CONSTANT, BUS = M_AXIS
PARAMETER C_M_AXIS_TDATA_WIDTH = 32, DT = integer, TYPE = NON_HDL,
ASSIGNMENT = CONSTANT, BUS = M_AXIS
## Peripheral ports
PORT ACLK = "", DIR=I, SIGIS=CLK, BUS=M_AXIS:S_AXIS
PORT ARESETN = ARESETN, DIR=I, INITIALVAL = VCC
PORT S_AXIS_TREADY = TREADY, DIR=O, BUS=S_AXIS
PORT S_AXIS_TDATA = TDATA, DIR=I, VEC=[31:0], BUS=S_AXIS
PORT S_AXIS_TLAST = TLAST, DIR=I, BUS=S_AXIS
PORT S_AXIS_TVALID = TVALID, DIR=I, BUS=S_AXIS
PORT M_AXIS_TVALID = TVALID, DIR=O, BUS=M_AXIS
PORT M_AXIS_TDATA = TDATA, DIR=O, VEC=[31:0], BUS=M_AXIS
PORT M_AXIS_TLAST = TLAST, DIR=O, BUS=M_AXIS
PORT M_AXIS_TREADY = TREADY, DIR=I, BUS=M_AXIS
PORT M_AXIS_TKEEP = TKEEP, DIR=O, VEC=[3:0], BUS=M_AXIS
END
```

这里存在两个问题: 一个是 ARESETN, 在连接时 AXI-DMA 上没有合适的引脚与之相连, 默认接地。这里显式声明接 VCC。另一个问题是 TKEEP 信号, 在我的博客文章《AXI-Stream 调试日记 (三)》里说过了, 这里加上这个引脚, 才能准确地将数据发回 AXI-DMA。

保存 MPD 文件, 关闭。再次右击 my_stream_ip, 选择 Browse HDL Sources, 打开 my_stream_ip.v (或 my_stream_ip.vhd), 添加 TKEEP 信号并设置 TLAST 信号。

```
module my_stream_ip
(
    ACLK,
    ARESETN,
    S_AXIS_TREADY,
    S_AXIS_TDATA,
    S_AXIS_TLAST,
    S_AXIS_TVALID,
```

```

M_AXIS_TVALID,
M_AXIS_TDATA,
M_AXIS_TLAST,
M_AXIS_TREADY,
M_AXIS_TKEEP
);
input          ACLK;
input          ARESETN;
output         S_AXIS_TREADY;
input  [31 : 0] S_AXIS_TDATA;
input          S_AXIS_TLAST;
input          S_AXIS_TVALID;
output         M_AXIS_TVALID;
output  [31 : 0] M_AXIS_TDATA;
output         M_AXIS_TLAST;
input          M_AXIS_TREADY;
output  [3:0]   M_AXIS_TKEEP;
localparam NUMBER_OF_INPUT_WORDS  = 8;
localparam NUMBER_OF_OUTPUT_WORDS = 8;
localparam Idle   = 3'b100;
localparam Read_Inputs = 3'b010;
localparam Write_Outputs = 3'b001;
reg [2:0] state;
reg [31:0] sum;
reg [NUMBER_OF_INPUT_WORDS - 1:0] nr_of_reads;
reg [NUMBER_OF_OUTPUT_WORDS - 1:0] nr_of_writes;
assign S_AXIS_TREADY = (state == Read_Inputs);
assign M_AXIS_TVALID = (state == Write_Outputs);
assign M_AXIS_TDATA = sum;
assign M_AXIS_TLAST = (nr_of_writes == 1);
assign M_AXIS_TKEEP = 4'b1111;
always @(posedge ACLK)
begin // process The_SW_accelerator
    if (!ARESETN) // Synchronous reset (active low)
    begin
        state      <= Idle;
        nr_of_reads <= 0;
        nr_of_writes <= 0;
        sum         <= 0;
    end
    else
    case (state)
        Idle:
            if (S_AXIS_TVALID == 1)

```

```

begin
  state    <= Read_Inputs;
  nr_of_reads <= NUMBER_OF_INPUT_WORDS - 1;
  sum      <= 0;
end
Read_Inputs:
if (S_AXIS_TVALID == 1)
begin
  sum      <= sum + S_AXIS_TDATA;
  if (nr_of_reads == 0)
  begin
    state    <= Write_Outputs;
    nr_of_writes <= NUMBER_OF_OUTPUT_WORDS - 1;
  end
else
  nr_of_reads <= nr_of_reads - 1;
end
Write_Outputs:
if (M_AXIS_TREADY == 1)
begin
  if (nr_of_writes == 0)
    state <= Idle;
  else
    nr_of_writes <= nr_of_writes - 1;
  end
endcase
end
endmodule

```

到这里修正了已知的所有 bug。VHDL 代码见我的博客文章 <http://www.eeforum.com/> 附件，或通过邮件联系我获取。完成上述更改后，点 XPS 菜单 Project→Rescan User Repositories，实现用户配置更新。

6. 点 Port 标签，引脚连接。这里重点是将所有带 CLK 字样的都连接到 PS7_FCLK_CLK0。如图 3-19 所示。

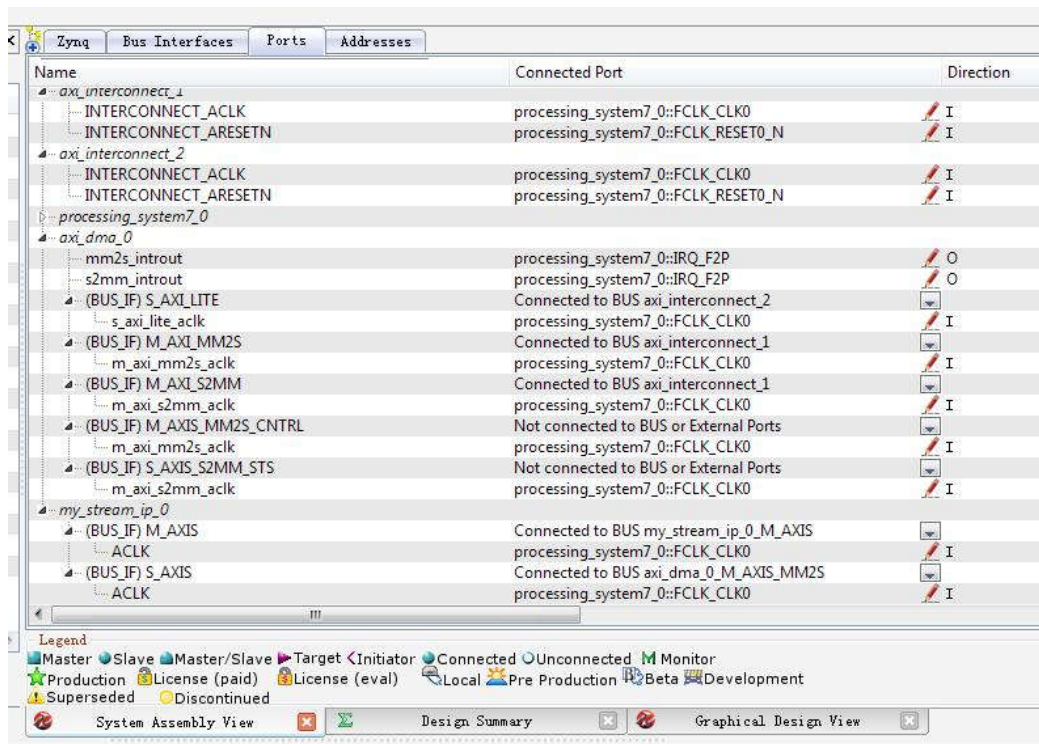


图 3-19 PORT 标签信号线连接

7. 点击 Addresses 标签，看看 AXI-DMA 是否分配了控制端口地址

Instance	Base Name	Base Address	High Address	Size	Bus Interface(s)	Lock	Bus Name
processing_system7_0's Addr...							
processing_system7_0	C_DDR_RAM_B...	0x00000000	0x1FFFFFFF	512M			
axi_dma_0	C_BASEADDR	0x40400000	0x4040FFFF	64K	S_AXI_LITE		axi_interconnect_
processing_system7_0	C_UART1_BASE...	0x20001000	0x20001FFF	4K			
processing_system7_0	C_GPIO_BASE...	0x2000A000	0x2000AFFF	4K			
processing_system7_0	C_ENET0_BASE...	0x2000B000	0x2000BFFF	4K			
processing_system7_0	C_SDIO0_BASE...	0x20100000	0x20100FFF	4K			
processing_system7_0	C_USB0_BASE...	0x20102000	0x20102FFF	4K			
processing_system7_0	C_TTC0_BASE...	0x20104000	0x20104FFF	4K			

图 3-20 地址分配

注意，如果你的 axi_dma_0 的地址和图中不一样，那么在后面软件编写时一定要修改成你的地址。

8. 点 Project->Design Rule Check; 没错时，点 Hardware->Generate Netlist，完成后关闭 XPS。

9. 在 PlanAhead 中完成综合、实现、生成 Bit 等步骤^[12]。其实上一步已经完成了综合，所以这一步速度就会非常快。

10 导出 SDK 工程。建立 Helloworld 工程。将 Helloworld.c 里面的内容改为如下代码。

```
#include <stdio.h>
#include <stdlib.h>
#include "platform.h"
#include "xil_cache.h" //必须包含该头文件，实现cache操作
#define sendram ((int *)0x10000000) //发送缓冲区
#define recvram ((int *)0x10001000) //接收缓冲区
#define sizeofbuffer 32
```

```

void print(char *str);
#define WITH_SG 0
#define AXI_DMA_BASE 0x40400000
#define MM2S_DMCR 0
#define MM2S_DMASR 1
#if WITH_SG
#define MM2S_CURDESC 2
#define MM2S_TAILDESC 4
#else
#define MM2S_SA 6
#define MM2S_LENGTH 10
#endif
#define S2MM_DMCR 12
#define S2MM_DMASR 13
#if WITH_SG
#define S2MM_CURDESC 14
#define S2MM_TAILDESC 16
#else
#define S2MM_DA 18
#define S2MM_LENGTH 22
#endif
void debug_axi_dma_register(unsigned int * p)
{
    printf("MM2S_DMCR = 0x%x\n",*(p+MM2S_DMCR));
    printf("MM2S_DMASR = 0x%x\n",*(p+MM2S_DMASR));
    #if WITH_SG
    printf("MM2S_CURDESC = 0x%x\n",*(p+MM2S_CURDESC));
    printf("MM2S_TAILDESC = 0x%x\n",*(p+MM2S_TAILDESC));
    #else
    printf("MM2S_SA = 0x%x\n",*(p+MM2S_SA));
    printf("MM2S_LENGTH = 0x%x\n",*(p+MM2S_LENGTH));
    #endif
    printf("S2MM_DMCR = 0x%x\n",*(p+S2MM_DMCR));
    printf("S2MM_DMASR = 0x%x\n",*(p+S2MM_DMASR));
    #if WITH_SG
    printf("S2MM_CURDESC = 0x%x\n",*(p+S2MM_CURDESC));
    printf("S2MM_TAILDESC = 0x%x\n",*(p+S2MM_TAILDESC));
    #else
    printf("S2MM_DA = 0x%x\n",*(p+S2MM_DA));
    printf("S2MM_LENGTH = 0x%x\n",*(p+S2MM_LENGTH));
    #endif
}
void init_axi_dma_simple(unsigned int * p)
{

```

```

*(p+MM2S_DMCCR) = 0x04; //reset send axi dma
while(*(p+MM2S_DMCCR)&0x04);
*(p+S2MM_DMCCR) = 0x04; //reset send axi dma
while(*(p+S2MM_DMCCR)&0x04);
*(p+MM2S_DMCCR)=1;
while((*(p+MM2S_DMASR)&0x01));
*(p+S2MM_DMCCR)=1;
while((*(p+S2MM_DMASR)&0x01));
*(p+MM2S_SA) = (unsigned int )sendram;
*(p+S2MM_DA) = (unsigned int )recvram;
Xil_DCacheFlushRange((u32)sendram,sizeofbuffer); //将 cache 内容同步到 DDR2
*(p+S2MM_LENGTH) = sizeofbuffer;//sizeof(recvram);
*(p+MM2S_LENGTH) = sizeofbuffer;//sizeof(sendram);
while(!(*(p+MM2S_DMASR)&0x1000)); //wait for send ok
}
void init_sendbuffer()
{
    int i;
    for(i=0;i< sizeofbuffer/4;i++)
    {
        sendram[i]=i*2;
    }
}
void show_recvbuffer()
{
    int i;
    printf("Recv contents are:\n");
    for(i=0;i< sizeofbuffer/4;i++)
    {
        printf("%d\t",recvram[i]);
    }
    printf("\r\n");
}
void show_sendbuffer()
{
    int i;
    printf("Send contents are:\n");
    for(i=0;i< sizeofbuffer/4;i++)
    {
        printf("%d\t",sendram[i]);
    }
    printf("\r\n");
}
int main()

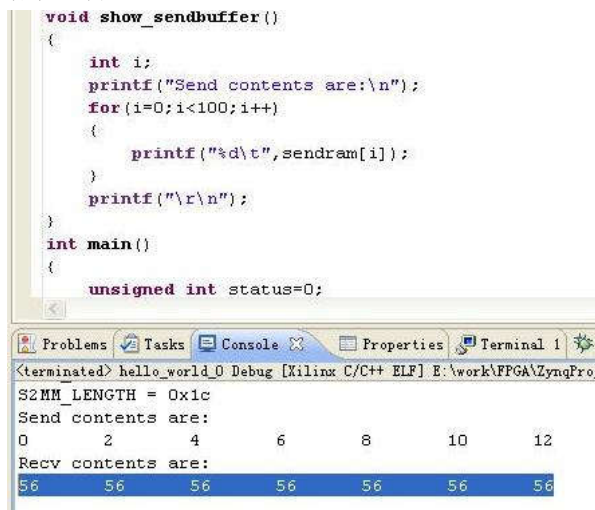
```

```

{
unsigned int status=0;
int rxlen;
init_platform();
init_sendbuffer();
init_axi_dma_simple((unsigned int *)AXI_DMA_BASE);
printf("Hello World\n\nPlease input data:");
while(1)
{
scanf("%x",&status);
printf("Got 0x%x\n",status);
debug_axi_dma_register((unsigned int *)AXI_DMA_BASE);
if(status==0)
{
break;
}
}
show_sendbuffer();
Xil_DCacheInvalidateRange((u32)recvram,sizeofbuffer);    //将 DDR2 内容同步到 cache
show_recvbuffer();
cleanup_platform();
return 0;
}

```

保存，等待生成 elf。然后连接板子，下载 bit 文件，Run App，打开串口终端，等待输出。由图 3-21 可见结果正确。



```

void show_sendbuffer()
{
    int i;
    printf("Send contents are:\n");
    for(i=0;i<100;i++)
    {
        printf("%d\t",sendram[i]);
    }
    printf("\r\n");
}

int main()
{
    unsigned int status=0;

```

Terminal Output:

```

<terminated> hello_world_0 Debug [Xilinx C/C++ ELF] E:\work\FPGA\ZynqPro:
S2MM_LENGTH = 0x1c
Send contents are:
0      2      4      6      8      10     12
Recv contents are:
56     56     56     56     56     56     56

```

图 3-21 程序输出

最终实现的 my_stream_ip 对外接口如下图所示。其中“M_AXIS”开头的信号线表示为 AXI_Stream 主机信号线，而“S_AXIS”开头的信号线表示为 AXI_Stream 从机信号线。自动生成的代码中没有 M_AXIS_TKEEP 信号，根据 AXI4_Stream 协议，这会导致该模块作为主机时发送的数据一直处于无效状态，影响数据传输。我们在 my_stream_ip 中添加了该信号，并使之有效，从而能够获得正确的处理数据。

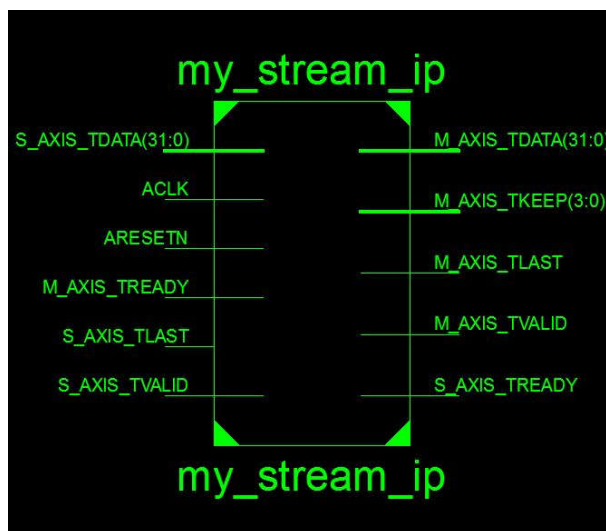


图 3-22 my_stream_ip 对外接口

其中 Xil_DCacheFlushRange()和 Xil_DCacheInvalidateRange()两个函数均在"xil_cache.h"中声明，用于将 cache 内容同步到 DDR2 或相反的操作。之前由于不了解 cache，导致程序一直得不到正确的结果，总是怀疑硬件问题，后来通过 forums.xilinx.com 看到了相关的帖子才明白这一点，在此感谢论坛上国内外的技术大牛为社区提供的支持。

3.2.4 总线与算法模块整合

有了上面 AXI_Stream IP 的开发基础，我们就可以对 3.1 节的 GAL 算法进行封装。这里主要完成两项任务：

1. GAL 模块实例化；
2. AXI-Stream 总线收发实现；

GAL 实例化比较简单，直接调用前面的模块，用线网将其对外接口与 my_stream_ip 的内部信号相连。遇到问题是更新模块第一阶的输入只能为寄存器型，不能为线网型（否则实现时会有一堆时序约束警告）。这样相当于数据输入多了一个 clk 延时，不影响信号连续处理过程。

总线收发部分代码采用状态机设计，my_stream_ip 在复位后进入 Idle 状态，开始监测 AXI_Stream 从机接口的 S_AXIS_TVALID 信号，一旦该信号有效，则进入 Read_Inputs 状态，接收 PS 发过来的输入数据，同时启动 35 阶 GAL 计算模块对输入数据和前一次的结果进行并行处理，同时将计算结果缓存至内部 BRAM；当读数据计数达到上限时（这里为了实现流水线处理，规定读计数上限为 7 个字），则使能 AXI_Stream 主机接口的 M_AXIS_TVALID 信号，将处理结果传回 PS 内的 DDR2 存储器，写计数上限为 35*7=245 个字，写入最后一个字的同时使能 M_AXIS_TLAST 信号，告知从机将结束数据传输，状态回到 Idle，并停止 35 阶 GAL 计算模块，等待下一批数据的到来。经过以上处理，实现了 AXI_Stream 总线的分时复用，持续不断更新输入数据并将计算结果传回 PS。该模块的状态流图见图 3-22 所示。

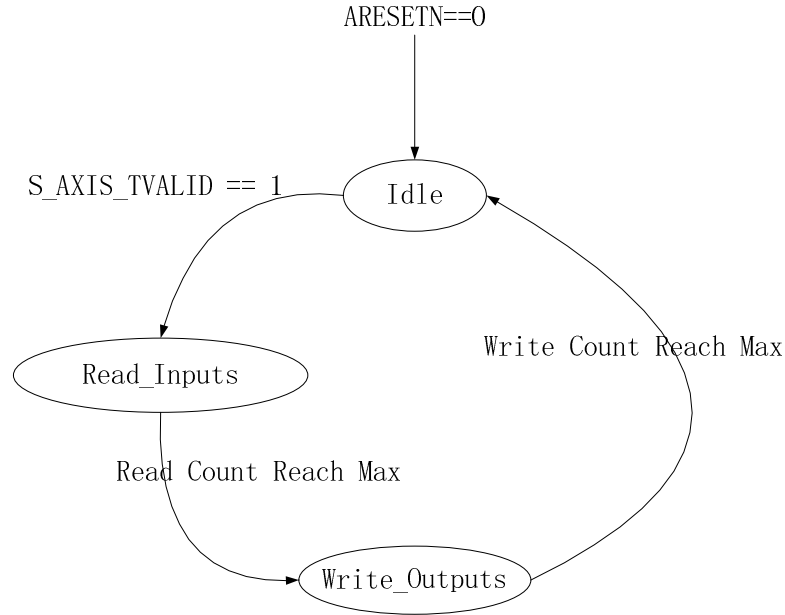


图 3-23 my_stream_ip 状态流图

根据 AXI-Stream 协议，本项目设计的 IP 包含主机与从机两个接口，首先通过从机接口获得 PS 部分传过来的输入数据，然后进行 GAL 处理，GAL 算法每输入一个数据点，各阶次会同时有输出点，35 阶 GAL 算法将会同时输出 35 个点，这样需要先将结果暂存，然后按照 Stream 总线协议依次将 35 个点通过主机接口传回 PS 部分的 DDR2 存储空间中。

为了便于流水线操作，各阶次对输入做了 7clk 的 FIFO，每次计算时从 FIFO 末端取数据进行计算。为了使硬件达到最高吞吐率，输入数据点组织为每 7 个点一组，依次输入，当最后一个点进入 FIFO 时，第一个点的计算结果已在输出端有效，可以取出缓存，最后通过总线一起传回 PS。这里遇到的问题是每块输入数据结束时，由于第一阶输入采用寄存器类型，如果没有恰当的措施控制，那么 GAL 模块将一直运行，对该输入寄存器中存储的最后一个样值连续重复采样，无形中相当于增加了输入数据长度，破坏了输入数据的连续性，导致后面一组输入数据到来时，前面的错误计算结果会引入后一组计算中，导致误差传播，以致结果完全错误。解决方法是在每次从机读取数据时，进行流水线更新操作；而空闲时，流水线处于静默状态，并停止其 clk 信号。这样做的另一个好处是 DSP48E1 模块在数据到来之前不运行，降低了系统功耗。

在硬件开发过程中，前期仿真（又称功能仿真）是非常关键且必要的的一个步骤，无论对算法调试还是总线调试都非常有帮助。通过 ISE 自带的 ISim 可以对工程中的任何一个模块进行功能验证。本项目开发流程为自底向上，首先项目组成员分别单独开发每个模块，并通过仿真验证其正确性，之后逐渐集成，生成一个个临时的顶层模块，然后通过 ISim 分别进行逐级仿真测试，及时发现 bug，并尽可能在早期改正，避免对后面系统造成更大的错误。算法调试需要有数据源，项目组成员为此专门利用 Matlab 软件编写了与 ISim 仿真配套的支持软件，用于数据产生、十进制-十六进制格式转换、仿真结果校验等，大大提升了算法调试效率。这里对 my_stream_ip 进行仿真的测试脚本如下：

```

module test_stream_ip;
    reg ACLK;
    reg ARESETN;
    reg [31:0] S_AXIS_TDATA;

```

```

reg S_AXIS_TLAST;
reg S_AXIS_TVALID;
reg M_AXIS_TREADY;
wire S_AXIS_TREADY;
wire M_AXIS_TVALID;
wire [31:0] M_AXIS_TDATA;
wire M_AXIS_TLAST;
wire [3:0] M_AXIS_TKEEP;
reg [15:0] data_mem[1:595*2];    //输入数据寄存
integer input_cnt,output_cnt;
integer fd;
// Instantiate the Unit Under Test (UUT)
my_stream_ip uut (
    .ACLK(ACLK),
    .ARESETN(ARESETN),
    .S_AXIS_TREADY(S_AXIS_TREADY),
    .S_AXIS_TDATA(S_AXIS_TDATA),
    .S_AXIS_TLAST(S_AXIS_TLAST),
    .S_AXIS_TVALID(S_AXIS_TVALID),
    .M_AXIS_TVALID(M_AXIS_TVALID),
    .M_AXIS_TDATA(M_AXIS_TDATA),
    .M_AXIS_TLAST(M_AXIS_TLAST),
    .M_AXIS_TREADY(M_AXIS_TREADY),
    .M_AXIS_TKEEP(M_AXIS_TKEEP)
);

initial begin
    // Initialize Inputs
    ACLK = 0;
    ARESETN = 0;
    S_AXIS_TDATA = 0;
    S_AXIS_TLAST = 0;
    S_AXIS_TVALID = 1;
    M_AXIS_TREADY = 1;
    input_cnt = 0;
    output_cnt = 0;
    $readmemh("xn.txt",data_mem); // 读入输入文件
    fd = $fopen("bn_1_to_35_fpga.txt");
    // Wait 100 ns for global reset to finish
    #105;
    ARESETN=1;
    // Add stimulus here

end

```

```

always #10 ACLK = ~ACLK;
always@(negedge ACLK)
begin
    if(S_AXIS_TREADY)
    begin
        S_AXIS_TDATA <= {data_mem[2*input_cnt+1],data_mem[2*input_cnt+2]};
        if(input_cnt<595)
        begin
            input_cnt = input_cnt+1;
        end
    end
end
always@(negedge ACLK)
begin
    if(M_AXIS_TVALID)
    begin
        if(output_cnt<595)
        begin
            $fdisplay(fd,"%8h",M_AXIS_TDATA);
            output_cnt = output_cnt+1;
        end
        else if(output_cnt==595)
        begin
            $fclose(fd);
            output_cnt = output_cnt+1;
        end
        else
        begin
            end
        end
    end
end
endmodule

```



图 3-24 my_stream_ip 功能仿真波形图

从上图可以明显看到在 S_AXIS_TREADY 有效时, my_stream_ip 在 ACLK 上升沿采样输入数据, 我们之前设计的测试脚本也是在 ACLK 上升沿送出数据, 这样会导致数字电路的“竞争——冒险”现象, 经过实现后与仿真结果有很大差异, 后来经过认真分析, 认为测试脚本输出数据在下降沿采样, 这样可以使输入数据稳定后再由 my_stream_ip 采样, 提高了系统稳定性, 经过硬件实现, 与仿真结果相同, 表明了上述测试脚本的有效性, 该脚本对 AXI_Stream 系统的模拟已经非常接近实际硬件。

在集成了 GAL 算法、AXI_Stream 总线控制模块后, 设计占用资源报告如下:

资源类型	资源明细	使用量	总量	使用百分比
Slice Registers	Total	35,152	106,400	33%
	Flip Flops	35,100	---	---
	AND/OR logics	52	---	---
Slice LUTs	Total	23,608	53,200	44%
	logic	16,257	53,200	30%
	Memory	4,720	17,400	27%
DSP48E1s		207	220	94%
RAMB36E1/FIFO36E1s		11	140	7%
RAMB18E1/FIFO18E1s:		9	280	3%

可见, 除了 DSP48E1 硬件乘法器资源占用接近 100% 以外, 其他资源均有较大剩余, 可以在此基础上进一步增加硬件模块, 扩展该项目硬件的功能 (如增加实际射频采集硬件, 直接实时处理射频信号, 这样有更强的实用性)。时间关系, 本项目暂时不做这方面研究, 而是处理存储在 SD 卡上的实测数据。

4. PS 上的软件设计

本章主要介绍 PS 上的软件设计, 具体分为基于 Linux 的驱动程序设计和基于 QT 的应用程序设计、NLMS 算法模块这三部分。

4.1 基于 Linux 的驱动程序设计

第三章中的硬件部分需要通过本节的驱动程序来完成初始化和数据输入、输出操作。驱动程序将硬件抽象为文件, 应用程序需要访问时, 只考虑文件的打开、关闭、读写等操作, 并不关心具体实现, 这样便于操作系统管理设备, 加载驱动与卸载驱动可以动态进行而不需要重新启动内核。驱动程序对本项目的整体性能具有至关重要的作用, 如果硬件性能很高, 而驱动程序效率很低, 势必会影响整机性能。

4.1.1 AXI_DMA 模块接口

本项目中的驱动程序主要用于控制并同 AXI_DMA 模块通信, 将上层应用程序发送的数据交给硬件处理, 并将硬件处理的结果返回给应用程序。编写驱动程序需要对硬件有较为深入的了解, 因此我们项目组成员专门搜集了官方硬件资料^[10]。该模块同时具有 AXI4_Lite、AXI4、AXI4_Stream 三种总线接口, 复杂度非常高, 所以对该模块的了解可以使我们更加深入地理解 AXI4 协议族, 对总线接口的选择更有把握。

AXI_DMA 的结构图如图 4-1 所示。其内部包含两个 DataMover 模块，负责将 AXI4 端的内存数据转换为 Stream 串行输出，或者将 Stream 串行输入放回 AXI4 端的内存中。其内部还有 Scatter/Gather 模块，可以支持分散传输，将大块数据分成小块进行传输，充分利用总线带宽。

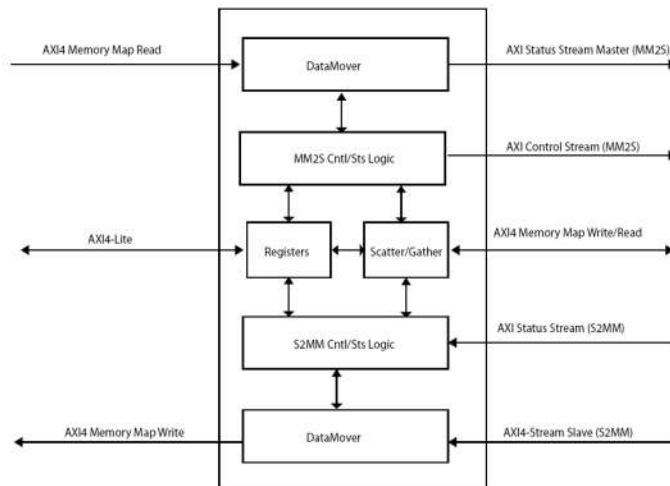


图 4-1 AXI_DMA 内部结构图

由上图可以看出，AXI_DMA 内部有配置寄存器，PS 端可以通过 AXI4_Lite 接口访问。在图 3-20 中看到，其内存映射地址为 0x4040000，我们将该地址作为同硬件通信的基地址。同两个 DataMover 相连的 AXI4 MM Read、MM Write 总线均连接到 PS 端的 AXI_HP0 接口，见图 3-17 所示，该接口可以访问 PS 内部 SRAM 和外部 DDR2 存储器，而且 AXI_DMA 模块做主机，有更高的访问权限。当有数据需要从 PS 传给 PL 时，CPU 将输入数据放至 DDR2 某一地址开始的一段空间，并通过配置 AXI_DMA 模块相关寄存器，告知起始地址和数据块大小，然后启动传输，AXI_DMA 自动完成数据搬移，CPU 可以通过中断方式或查询方式得到数据搬移结束信号，然后等待硬件处理结果。这时 CPU 也配置好了 AXI_DMA 相应寄存器，告知传回数据的起始地址和数据块大小，当 PL 部分产生计算结果需要传回 PS 时，直接按照配置进行传输，无需 CPU 干预，完成后也会产生中断或 CPU 直接查询相应标志位。

AXI_DMA 的内部配置寄存器如图 4-2 所示^[10]。

Address Space Offset ⁽¹⁾	Name	Description
00h	MM2S_DMOCR	MM2S DMA Control Register
04h	MM2S_DMASR	MM2S DMA Status Register
08h - 14h	Reserved	N/A
18h	MM2S_SA	MM2S Source Address
1Ch - 24h	Reserved	N/A
28h	MM2S_LENGTH	MM2S Transfer Length (Bytes)
30h	S2MM_DMOCR	S2MM DMA Control Register
34h	S2MM_DMASR	S2MM DMA Status Register
38h - 44h	Reserved	N/A
48h	S2MM_DA	S2MM Destination Address
4Ch - 54h	Reserved	N/A
58h	S2MM_LENGTH	S2MM Buffer Length (Bytes)

图 4-2 AXI_DMA 内部寄存器映射

根据以上寄存器地址，我们可以很方便地通过 CPU 访问 AXI_DMA 模块，对其进行相应配置从而实现我们项目需要的功能。为了便于访问，在驱动程序中首先声明了寄存器相关的宏定义如下所示。

```
#define WITH_SG 0 //我们没有使能 SG 模式，因此为 0
#define AXI_DMA_BASE 0x40400000 //基地址
#define MM2S_DMACR 0
#define MM2S_DMASR 1
#if WITH_SG
#define MM2S_CURDESC 2
#define MM2S_TAILDESC 4
#else
#define MM2S_SA 6
#define MM2S_LENGTH 10
#endif
#define S2MM_DMACR 12
#define S2MM_DMASR 13
#if WITH_SG
#define S2MM_CURDESC 14
#define S2MM_TAILDESC 16
#else
#define S2MM_DA 18
#define S2MM_LENGTH 22
#endif
```

有了以上宏定义，我们就可以通过硬件地址访问并读写 AXI_DMA 模块了。

4.1.2 驱动程序模块入口与出口

Linux 驱动程序也是以模块的方式组织的，模块的初始化和退出由下面代码实现。在终端下执行 insmod 时自动调用模块初始化函数，而执行 rmmod 时自动调用模块退出函数。

```
static void __iomem* Regs;
int *sendbuffer;
int *recvbuffer;
static struct miscdevice axi_dma_dev =
{
    .minor = MISC_DYNAMIC_MINOR,
    .name = DEVICE_NAME,
    .fops = &axi_dma_fops,
};
static int __init axi_dma_init(void)
{
    int ret;
    Regs = ioremap(AXI_DMA_BASE, 23); //将硬件地址转换为虚拟地址，23 个寄存器
    printk("GalEqlz:Access address to device is :0x%x\n", (unsigned int)Regs);
    int val;
```

```

iowrite32(0x04, Regs+4*MM2S_DMACR); //reset send axi dma
do
{
    val = ioread32(Regs+4*MM2S_DMACR);
}while(val&0x04);
iowrite32(0x04, Regs+4*S2MM_DMACR); //reset recv axi dma
do
{
    val = ioread32(Regs+4*S2MM_DMACR);
}while(val&0x04);
iowrite32(0x01, Regs+4*MM2S_DMACR); //enable send axi dma
do
{
    val = ioread32(Regs+4*MM2S_DMASR);
}while(val&0x01);
iowrite32(0x01, Regs+4*S2MM_DMACR); //enable recv axi dma
do
{
    val = ioread32(Regs+4*S2MM_DMASR);
}while(val&0x01);
ret = misc_register(&axi_dma_dev); //注册设备
return ret;
}
static void __exit axi_dma_exit(void)
{
    iounmap(Regs); //取消虚拟地址与硬件地址的映射关系
    misc_deregister(&axi_dma_dev); //注销设备
    printk("GalEqlz module exit.\n");
}
module_init(axi_dma_init); //将 axi_dma_init 做为初始化函数
module_exit(axi_dma_exit); //将 axi_dma_exit 做为初始化函数

```

4.1.3 驱动程序文件接口

只有上面这两个函数，只能完成硬件初始化操作，其他的还是什么都做不了。我们需要为上层应用程序提供文件接口，所以需要完成文件读写接口函数的编写。文件相关的函数声明代码见下面所示。

```

static int axi_dma_open(struct inode*inode, struct file*filp);
static int axi_dma_release(struct inode*inode, struct file *filp);
static int axi_dma_read(struct file*filp, char *buffer, size_t length, loff_t *
offset);
static int axi_dma_write(struct file*filp, char *buffer, size_t length, loff_t *
offset);
static struct file_operations axi_dma_fops={ //文件操作结构体

```



```
.owner = THIS_MODULE,
.open = axi_dma_open,          //文件打开回调函数
.release = axi_dma_release,    //文件关闭回调函数
.read = axi_dma_read,         //文件读取回调函数
.write = axi_dma_write,       //文件写入回调函数
};
```

上面声明了和文件操作相关的 5 个接口函数，基本涵盖了用户应用的所有操作（打开文件，读取文件，写入文件，关闭文件等），所以下面问题就是如何实现这 5 个函数。

文件打开和关闭比较简单，打开时分配资源，关闭时释放资源，分别调用相应的内核函数即可，通过下面代码可以实现这两个功能。

```
static int axi_dma_open(struct inode*inode,struct file*filp)
{
    int i,j;
    sendbuffer=kmalloc(MAX_LEN*4,GFP_KERNEL);    //分配发送缓冲区
    if(!sendbuffer)
    {
        printk("Open Gal-->Send buffer NULL!\n");
        return -1;
    }
    recvbuffer=kmalloc(MAX_LEN*GAL_M*4+10,GFP_KERNEL); //分配接收缓冲区
    if(!recvbuffer)
    {
        printk("Open Gal-->Recv buffer NULL!\n");
        return -1;
    }
    return 0;
}

static int axi_dma_release(struct inode*inode,struct file *filp)
{
    if(sendbuffer)
    {
        kfree(sendbuffer);    //释放发送缓冲区
        sendbuffer = NULL;
    }
    if(recvbuffer)
    {
        kfree(recvbuffer);    //释放接收缓冲区
        recvbuffer = NULL;
    }
    return 0;
}
```

本项目中使用硬件时，应该每次先写入一组输入数据，然后调用 35 阶 GAL 算法模块进行计算，得到 35 组输出数据一次性读出。写入数据的函数如下所示。

```
#define MAX_LEN 7
```

```

#define GAL_M 35
static int axi_dma_write(struct file*filp, char *buffer, size_t length, loff_t *
offset)
{
    int bytes_write = 0, i;
    int val;
    for(i = 0; i < MAX_LEN; i++)
    {
        sendbuffer[i] = 0; //初始化缓冲区
    }
    if(length > 0) //如果应用程序写入数据长度有效
    {
        char * p = (char*)sendbuffer;
        bytes_write = (length > (MAX_LEN*4)) ? (MAX_LEN*4) : (length);
        for(i=0; i < bytes_write; i++)
        {
            *p++ = *(buffer+i); //将用户数据放入发送缓冲区
        }
    }
    dma_addr_t send_mapping;
    send_mapping = dma_map_single( &axi_dma_dev, sendbuffer, MAX_LEN*4,
DMA_TO_DEVICE); //获得 sendbuffer 的硬件地址
    iowrite32((unsigned int)send_mapping, Regs+4*MM2S_SA); //配置 AXI_DMA
发送通道的起始地址寄存器
    dma_addr_t recv_mapping;
    recv_mapping = dma_map_single(&axi_dma_dev, recvbuffer, MAX_LEN*GAL_M*4,
DMA_FROM_DEVICE ); //获得 recvbuffer 的硬件地址
    iowrite32((unsigned int)recv_mapping, Regs+4*S2MM_DA); //配置 AXI_DMA
接收通道的起始地址寄存器
    dma_sync_single_for_device(&axi_dma_dev, send_mapping, MAX_LEN*4,
DMA_TO_DEVICE ); //清空 CPU cache, 写同步
    iowrite32(MAX_LEN*GAL_M*4, Regs+4*S2MM_LENGTH); //配置接收长度并开启接收
    iowrite32(MAX_LEN*4, Regs+4*MM2S_LENGTH); //配置发送长度并开启发送
    do
    {
        val = ioread32(Regs+4*MM2S_DMASR);
    } while(!(val & 0x1000)); //等待发送完成
    return bytes_write;
}

```

在编写这部分代码时遇到了 Cache 问题, 纠结了很长时间。PS 运行 Linux 操作系统, 为了提高 CPU 性能, 使能了 Cache 功能, 所以当我们向 DDR2 某一地址写入内容时, CPU 运行结束后, DDR2 中相应地址可能还没有更新, 因为数据可能仍然在 Cache 中。这时需要强制将 Cache 中的内容刷新到 DDR2 中完成数据写同步, 通过调用 `dma_sync_single_for_device(&axi_dma_dev, send_mapping, MAX_LEN*4, DMA_TO_DEVICE)`

函数实现，第一个参数为设备结构体指针，第二个参数为需要同步的数据物理基地址，第三个参数为需要同步的数据长度，第四个参数 DMA_TO_DEVICE 表示数据同步方向为 Cache 到 DDR2，而 DMA_FROM_DEVICE 表示数据同步方向为 DDR2 到 Cache，这两个宏均在头文件 <linux/slab.h>中声明。

剩下最后一个文件读取回调函数了，这个在前面基础上实现起来就很方便快捷了。时间关系，下面的代码不再添加注释，相应的函数介绍见前面函数相同的部分。

```
static int axi_dma_read(struct file*filp,char *buffer,size_t length,loff_t *
offset)
{
    int bytes_read = 0;
    int i = 0;
    int val;
    if(filp->f_flags&O_NONBLOCK)
    {
        return -EAGAIN;
    }
    dma_addr_t recv_mapping;
    recv_mapping =
dma_map_single(&axi_dma_dev,recvbuffer,MAX_LEN*GAL_M*4,DMA_FROM_DEVICE);
    do
    {
        val = ioread32(Regs+4*S2MM_DMASR);
    }while(!(val&0x1000));
    val = ioread32(Regs+4*S2MM_LENGTH);
    dma_sync_single_for_cpu(&axi_dma_dev,recv_mapping,MAX_LEN*GAL_M*4,DMA_F
ROM_DEVICE);
    if(length>0)
    {
        if(length<=val)
        {
            char * p = (char*)recvbuffer;
            for(i=0;i<length;i++)
            {
                *(buffer+i)=*p++;
            }
            bytes_read=i;
        }
        else
        {
            char * p = (char*)recvbuffer;
            for(i=0;i<val;i++)
            {
                *(buffer+i)=*p++;
            }
        }
    }
}
```

```

        bytes_read=i;
    }
}
return bytes_read;
}

```

4.1.4 驱动程序小结

通过对硬件的深入了解与基础模块的编写，基于 Linux 系统的驱动程序终于告一段落，实现了应用程序读写设备文件所需的全部接口，通过 AXI_DMA 实现了同用户自定义 IP 的高速数据通信，为后面应用程序的设计提供了重要的支撑。

由于以前有过 Linux 驱动程序开发经验，所以感觉这次比赛驱动开发还是相对轻松的，唯一遇到比较有挑战性的问题就是 Cache 问题。之前做过比较简单的字符设备驱动，没有遇到过 Cache 这类问题，而通过这次驱动程序编写，涉及到 DMA 操作时发现必须解决 Cache 的同步问题。一开始的解决思路是从裸机代码入手，查看了 Xilinx SDK 中 xil_cache.h 中的 Xil_DCacheFlushRange() 函数，试图直接通过改写裸机的 Cache 同步代码为驱动程序，但很失败，每次运行改写后的函数时都会出现“Segment Error”这类错误。一次次打击后，放弃了这种做法，通过网络发现在 Linux 下不能保留裸机开发的思路，因为裸机代码是对硬件的直接操作，不用考虑其他线程访问问题，但在 Linux 下就不能这么做，操作系统会有一些保护措施。后来仔细读了一下 linux 和 DMA、Cache 有关的源代码，又做了很多次尝试，终于找到了较可靠的解决办法，即 dma_map_single 函数。

从这次经历我收获很大，至少对于 Linux 驱动程序编写有了很多新的认识，包括驱动与内核的关系，驱动与裸机代码的区别，驱动与上层应用程序的任务划分等。相信在以后遇到这类问题时，我都能够从容应对。感谢 Open HW 社区提供的这次开发实际项目的机会，我获益匪浅。

4.2 基于 Qt 的应用程序设计

Qt 作为 Linux 下图形用户界面的强大编程工具，能给用户提供更精美的图形界面所需的所有元素，已经得到了越来越广泛的应用，并且当前多数高端嵌入式设备生产商都选择了 Qt 作为开发工具^[14]。Qt 同时是一个移植性很强的 C++ 库，可运行在 Linux、Windows、MacOS 等多种平台，本项目组开发采用 Qt Creator 工具，在 Windows 下设计主界面，调试基本单元，成功后在 Linux 下进行交叉编译，使之能在 ZED-Board 上运行，大大加速了开发进度。

基于 ZED-Board 实现的宽带实时自适应均衡器项目应用程序主要完成如下几个功能：

1. 读取实测数据；
2. 显示实测数据波形；
3. 设置 GAL_NLMS 算法的基本参数；
4. 控制 PL 硬件部分的打开、关闭、读写数据；
5. 利用软硬件协同完成 GAL_NLMS 算法；
6. 显示处理结果（时域波形和星座图）；
7. 显示处理参数（均衡器权值，反射系数值）；
8. 显示算法处理耗时；

项目组为该应用程序命名为 ZEDBOARD Control Center，简称 ZCC，当前版本为 1.0。

该软件的主界面如图 4-3 所示。

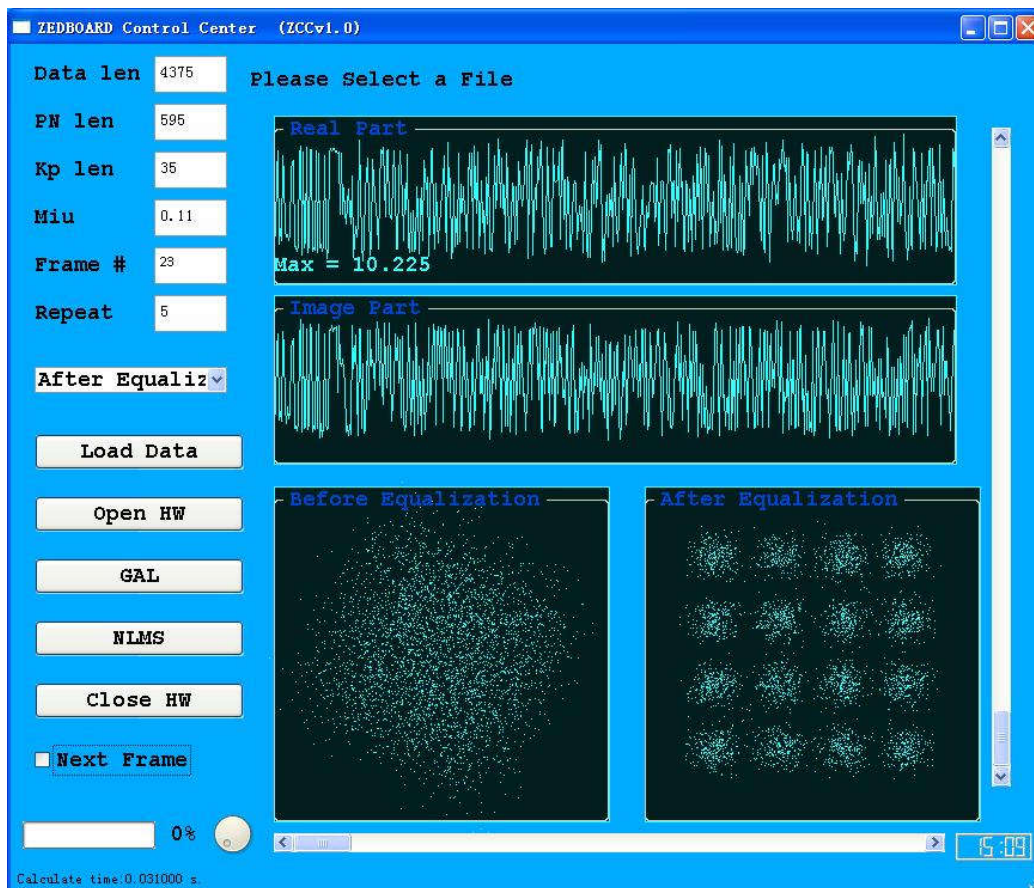


图 4-3 ZCC 主界面

软件主要分为 5 个区域：参数设置区（左上方）、控制区（左下方）、时域波形显示区（右上方）、星座图显示区（右下方）、时间显示区（最底行）。

参数设置区主要用于设置 GAL_NLMS 算法参数如阶次（Kp len）、步长因子（Miu）、反复次数（Repeat），以及输入数据格式参数如处理长度（Data len）、均衡器训练序列长度（PN len）和起始帧（Frame #）。只有参数符合实际数据才能正确读取输入文件。这里输入文件参数如图 4-3 所示，处理长度为 4375 个采样点，而训练序列占 595 个采样点，这个在软件进行均衡处理时是固定不变的。阶次由于第三章硬件实现的原因，只能做固定的 35 阶，所以该参数也是固定不变的。步长因子与反复次数可以根据需求手动设置，范围为 (0, 1)，步长因子越大则收敛越快，越小收敛越慢，需要增加 Repeat 次数使均衡器权值达到收敛。

参数设置区下方的下拉菜单可以选择当前时域信号显示区的信号类型，共有 4 种：均衡前信号，均衡后信号，均衡器权值，反射系数。当选择其中某一项时，右上侧的两个窗口会显示相应的波形（实部与虚部分开显示）。

再向下就是控制区按钮了，“Load Data”为载入数据按钮；“Open HW”为打开 PL 硬件按钮（与 OpenHW 社区名不谋而合 $0(\cap \cap)0^{\sim}$ ），只有打开成功时，才能进行后面的 GAL 算法；“GAL”开始 GAL 算法，结束后会在时间显示区显示 GAL 处理时间（以秒为单位）；“NLMS”开始对 GAL 输出结果进一步用 NLMS 算法处理，得到均衡后的信号，同时在时间显示区显示处理时间；“Close HW”为关闭硬件按钮，当处理完成后，关闭软件之前需要先关闭硬件，释放 CPU 与内存、硬件资源。

界面右上方两个窗口为时域波形显示区，显示信号的时域波形细节，可以通过两个滑动

条进行缩放显示。通过时域波形显示，可以对比均衡前后时域波形的差异，评估均衡性能。而界面右下方的两个方形窗口为星座图显示区，分别显示均衡前和均衡后的信号星座图，更加直观地评估均衡性能。图 4-3 中，均衡前几乎分辨不出星座的形状，说明信号失真非常严重，经过均衡后，信号星座图得到了旋转校正和失真补偿，能够清楚地看出其调制方式为 16QAM，说明均衡算法是有效的。

时间显示分为处理时间和实时时钟两部分，最底行左侧为处理时间显示，通过它可以记录下处理一批数据所需的时间（包括 GAL 部分和 NLMS 两部分），为系统性能评估提供重要的时间参考。实时时钟显示当前系统时间，由于板卡上电时系统时钟被设为 00:00，所以也可以作为软件持续运行时间记录。

软件操作流程如图 4-4 所示：

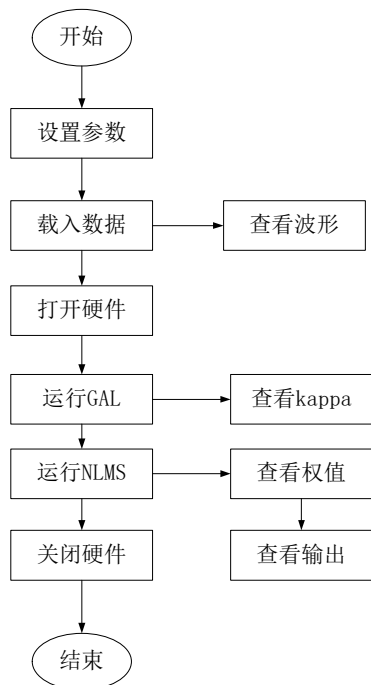


图 4-4 软件操作流程

由于时间关系，软件还有很多进一步优化的可能。首先是多线程，可以将 GAL 算法数据读写划分为单独的一个线程，这样操作起来会更流畅；其次是增加信号类型，当前只用了较为简单的 16QAM 信号，我们还可以针对当前系统研究高斯噪声信号、调频连续波信号、DSSS 扩频信号、OFDM 信号等作为输入信号，更全面、系统地评估算法的有效性与实用性；最后是增加网络功能，将输入数据和处理结果通过千兆以太网接口与远程工作站交互，而不是反复读写 SD 卡（SD 卡的读写速度制约了算法加速），这样也为今后多板卡联合组网处理提供支持。

4.3 NLMS 算法模块

NLMS 算法采用 C++ 语言实现。相比硬件描述语言，C++ 更适合复数处理，下面代码描述了复数计算的功能。

```

typedef struct
{
    double r;

```

```

        double i;
    }base_complex;
class CComplex
{
public:
    CComplex()
    {
        store.r=0;
        store.i=0;
    }
    CComplex(double r,double i)
    {
        store.r=r;
        store.i=i;
    }
    CComplex(double r)
    {
        store.r=r;
        store.i=0;
    }
    CComplex(base_complex p)
    {
        store=p;
    }
    ~CComplex() {}
    double getReal()
    {
        return store.r;
    }
    double getImag()
    {
        return store.i;
    }
    CComplex operator+(CComplex income)
    {
        CComplex p;
        p.store.r = store.r+income.store.r;
        p.store.i = store.i+income.store.i;
        return p;
    }
    CComplex operator+(double d)
    {
        return CComplex(store.r+d,store.i);
    }
}

```

```

CComplex operator-(CComplex income)
{
    CComplex p;
    p.store.r = store.r-income.store.r;
    p.store.i = store.i-income.store.i;
    return p;
}
CComplex operator-(double d)
{
    return CComplex(store.r-d,store.i);
}
CComplex operator*(CComplex income)
{
    CComplex p;p.store.r = store.r*income.store.r-store.i*income.store.i;
    p.store.i = store.i*income.store.r+store.r*income.store.i;
    return p;
}
CComplex operator*(double d)
{
    return CComplex(store.r*d,store.i*d);
}
CComplex operator/(CComplex income)
{
    CComplex p;
    p.store.r = (store.r * income.store.r + store.i * income.store.i) /
(income.store.r * income.store.r + income.store.i * income.store.i);
    p.store.i = (store.i * income.store.r - store.r * income.store.i) /
(income.store.r * income.store.r + income.store.i * income.store.i);
    return p;
}
CComplex operator/(double d)
{
    return CComplex(store.r/d,store.i/d);
}
CComplex abssquare()
{
    return CComplex(store.r*store.r+store.i*store.i);
}
CComplex conj()
{
    return CComplex(store.r,-store.i);
}
private:
    base_complex store;

```



```
};
```

经过 C++ 类的封装，我们可以很方便地对复数数据进行运算，实现丰富的算法。下面代码为基本 LMS 部分的程序：

```
void lms(CComplex *input, CComplex *refer, CComplex *w, CComplex*out, double
miu, int M, int LEN)
{
    int i, j;
    int offset = M/2;
    for(i = M; i < LEN; i++)
    {
        CComplex sum(0, 0), power(0, 0);
        CComplex err;
        CComplex newdata;
        for(j=0; j<M; j++)
        {
            newdata = input[i-M+j];
            sum = sum + newdata*w[j];
            power = power + newdata.abssquare();
        }
        out[i] = sum;
        err = refer[i-offset]-sum;
        for(j=0; j<M; j++)
        {
            newdata = input[i-M+j];
            w[j] = w[j] + newdata.conj()*err*miu/(power.getReal()+0.0001);
        }
    }
}
```

由于本项目用到的是 GAL-NLMS 算法，串行部分采用正交输入的 NLMS 算法，首先需要—个通信模块，完成于 GAL 部分的数据交换。这部分代码如下：

```
//GAL-related Process
//Begin
if(openhw_id < 0) //如果硬件没有打开
{
    QMessageBox msgBox;
    msgBox.setText("GAL Device not open!");
    msgBox.exec();
    ui->statusBar->showMessage("GAL Device not open!");
    return;
}
if(this->isLoading != 1) //如果数据没有载入
{
    QMessageBox msgBox;
    msgBox.setText("Data not loaded!");
```

```

        msgBox.exec();
        ui->statusBar->showMessage("Data not loaded!");
        return;
    }
    if(to_gal)
    {
        delete [] to_gal;
    }
    to_gal = new int[data_len];    //输入缓冲区
    if(from_gal)
    {
        delete [] from_gal;
    }
    from_gal = new int[data_len*kappa_len+kappa_len*kappa_len*BLK_LEN]; //输出缓冲区
    int myzero[BLK_LEN]={0};    //输入零缓冲区
    for(int i = 0;i<data_len;i++)    //转换为 3.1.1 中的数据格式，Q15_IQ 格式
    {
        short a1 = short( 0.5 + eq_input[i].getReal() / maxValue_input / 1.5
* 32767 );
        short a2 = short( 0.5 + eq_input[i].getImag() / maxValue_input / 1.5
* 32767 );
        int gendat = a1;
        gendat <= 16;
        gendat |= (a2&0x0000ffff);
        to_gal[i] = gendat;
    }
    QTime t;    //计时器
    t.start();    //计时开始
    for(int i=0;i<data_len/BLK_LEN;i++)
    {
        write(openhw_id, (void*)(to_gal+i*BLK_LEN), BLK_LEN*sizeof(int)); //向 GAL 模块写入 BLK_LEN 个采样点 (BLK_LEN=7)
        read(openhw_id, (void*)(from_gal+i*BLK_LEN*kappa_len), BLK_LEN *
kappa_len * sizeof(int));    //从 GAL 模块中读出 BLK_LEN*kappa_len (=7*35) 个
计算结果
    }
    for(int i=0;i<kappa_len;i++)    //将各级延迟的处理结果读回
    {
        write(openhw_id, (void*)(myzero), BLK_LEN*sizeof(int));    //写入 0
        read(openhw_id, (void*)( from_gal + data_len * kappa_len+ i * BLK_LEN
* kappa_len ), BLK_LEN * kappa_len * sizeof(int));    //读回延迟结果
    }
    int calc_time = t.elapsed();

```

```

QString str;
str.sprintf("GAL OK! Calculate time: %.3f s.", calc_time/1000.0f); //显示计算所用时间
ui->statusBar->showMessage(str);
/***** E **** N **** D *****/
在上面程序的基础上很快得到下面的算法:
void galnlms(int *input, CComplex *refer, CComplex *w, CComplex*out, double
miu, int M, int eq_len, int frame_len)
{
    int i, j;
    int offset = M/2;
    for(i = 0; i < frame_len; i++) //均衡点数索引
    {
        CComplex sum(0, 0), power(0, 0);
        CComplex err;
        CComplex newdata;
        for(j=0; j<M; j++) //阶次索引
        {
            int block_idx = (i+j+1)/BLK_LEN;
            int inner_idx = (i+j+1)%BLK_LEN;
            int total_idx =
            BLK_LEN*M*(block_idx+1+j)+(j*BLK_LEN+inner_idx+1)*(BLK_LEN*M);
            newdata =
            CComplex(input[total_idx]>>16, (input[total_idx]<<16)>>16);
            sum = sum + newdata*w[j];
            power = power + newdata.abssquare();
        }
        out[i] = sum;
        if(i<eq_len)
        {
            if(i>=offset)
            {
                err = refer[i-offset]-sum;
            }
            else
            {
                err = CComplex(0, 0)-sum;
            }
            for(j=0; j<M; j++)
            {
                int block_idx = (i+j+1)/BLK_LEN;
                int inner_idx = (i+j+1)%BLK_LEN;
                int total_idx =
                BLK_LEN*M*(block_idx+1+j)+(j*BLK_LEN+inner_idx+1)*(BLK_LEN*M);

```

```

newdata
=
CComplex(input[total_idx]>>16, (input[total_idx]<<16)>>16);
w[j] = w[j] + newdata.conj()*err*miu/(power.getReal()+0.0001);
}
}
}
}
}

```

其中计算 total_idx 的算法是非常关键的一步。前面 GAL 算法采用硬件实现，与 PS 接口导致其传输数据只能分组进行，每组输入 7 个点，每阶同时都有 7 个点输出，总共输出 $35 \times 7 = 245$ 个点。GAL 算法每级有 7clk 延迟，也就是说，得到 245 个输出点还不能直接进行 NLMS 计算，而需要用到其后 34 组输出点，这样就为 NLMS 算法带来了许多挑战，我们需要开辟较大的存储区缓存结果，等需要的数据到齐了才能进行 NLMS 计算。

5. 系统调试和测试

5.1 平台测试

1. BOOT 测试

测试步骤：将硬件设置为 SD 卡启动，将 SD 卡插入卡槽，连接串口，开机后打开 PC 上的超级终端，观察 UBoot 是否启动成功。

测试结果：正常启动。

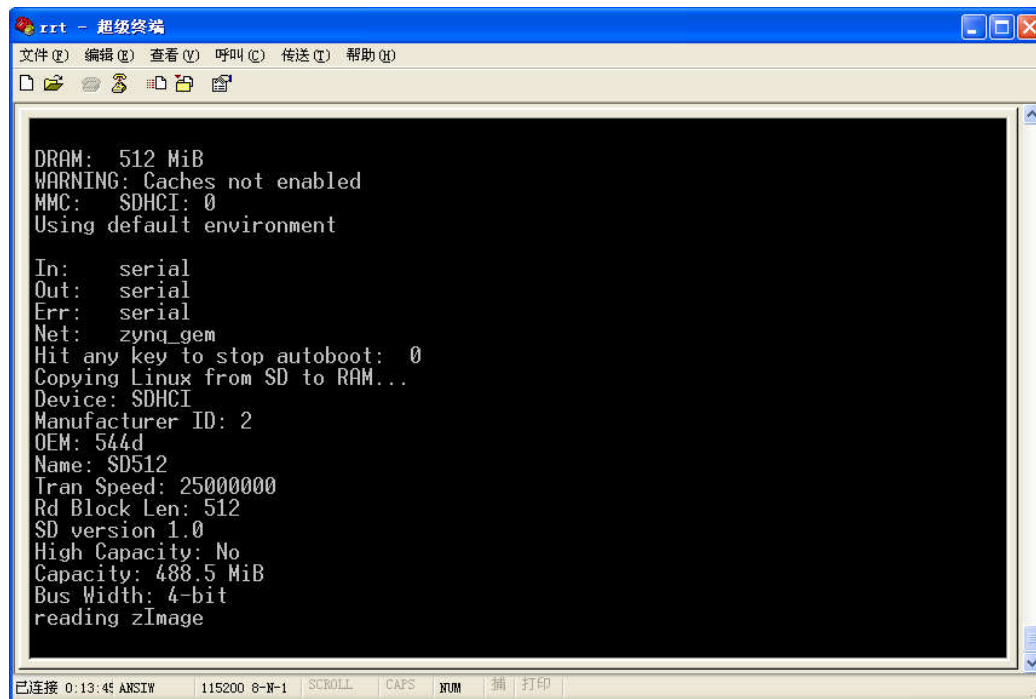


图 5-1 BOOT 测试结果

2. 硬件基本功能测试

测试步骤：Linux 启动成功后，OLED 是否显示 Digilent 图标。

测试结果：正常显示。

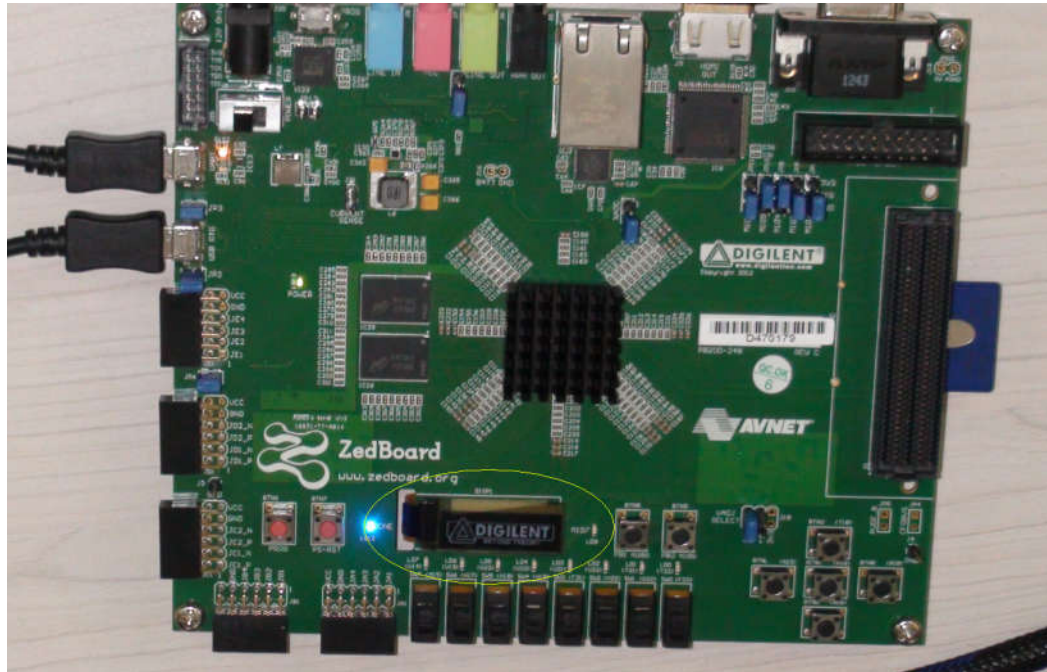


图 5-2 OLED 显示测试

3. Qt 应用程序测试

测试步骤：Linux 启动成功后自动运行 Qt 应用程序，观察是否显示完整，按照 4.2 节的操作观察各项功能是否正常。

测试结果：正常。

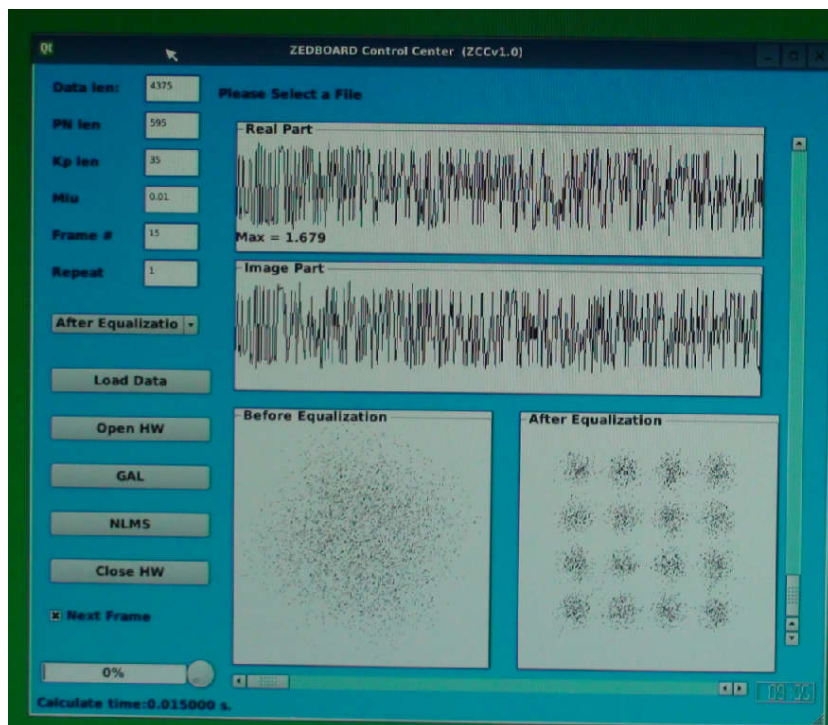


图 5-3 Qt 应用程序测试

4. 强度测试

测试步骤：Qt 应用程序 ZCC 连续运行 12 小时，观察是否会出现死机、算法出错、硬件故障等异常情况。

测试结果：正常工作。

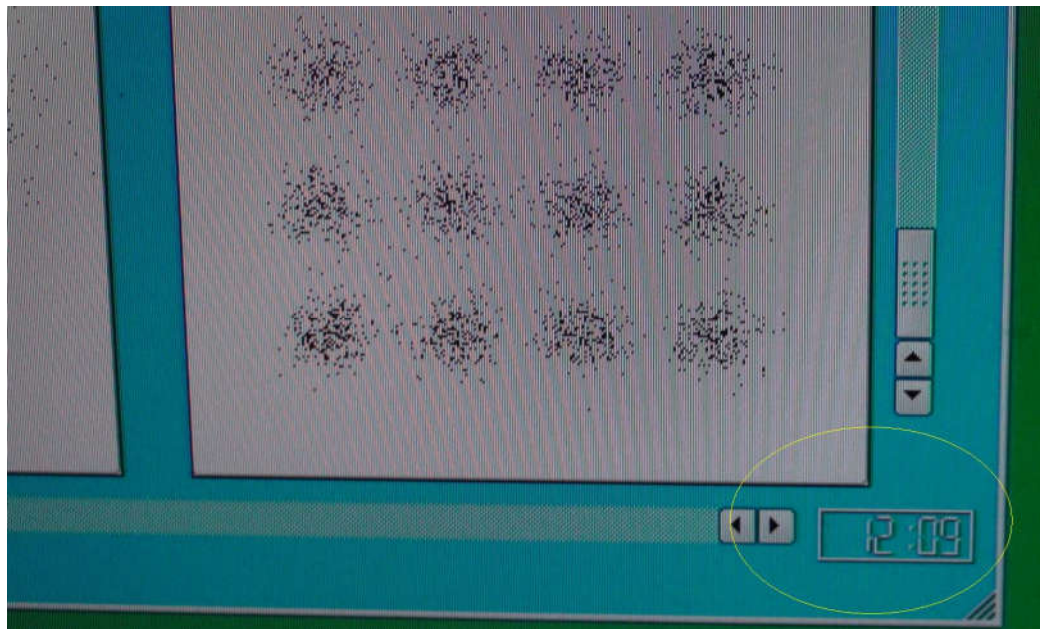


图 5-4 强度测试

5.2 GAL 算法调试

1. 功能测试

测试方法：将 ZCC 保存的中间结果（input_hex.txt 和 output_hex.txt）从 SD 卡拷贝到 PC 上，运行 matlab 脚本进行中间结果验证，观察是否与预期相同。脚本内容为：

```
clear;
clc;
close all;
M = 36; % filter length
miu = 0.11; % Step size
D = 18; % Number of delay samples
ntr= 595; % Number of iterations
send_sig = load_from_VC_dat_file('eq_refer.dat');
d = [zeros(1,D),send_sig(1:ntr-D)];
fid = fopen('input_hex.txt','r');
to_gal = fscanf(fid,'%04x\n');
fclose(fid);
to_gal_s = to_gal;
to_gal_s(to_gal_s>32767) = to_gal_s(to_gal_s>32767)-65536;
to_gal = to_gal_s;
xin = downsample(to_gal,2,0)+1j*downsample(to_gal,2,1);
xin = reshape(xin,1,[]);
x = xin(1:ntr);
```

```

fid = fopen('output_hex.txt','r');
from_gal = fscanf(fid,'%04x\n');
fclose(fid);
from_gal_s = from_gal;
from_gal_s(from_gal_s>32767) = from_gal_s(from_gal_s>32767)-65536;
from_gal = from_gal_s;
cin = downsample(from_gal,2,0)+1j*downsample(from_gal,2,1);
cin = reshape(cin,35*7,[]).';
cin = [cin(:,2:end),cin(:,1)];
for k = 1:35
    cout(k,:) = reshape(cin(:,(k-1)*7+1:k*7).',1,[]);
    cout(k,:) = [cout(k,1+k*8:end),cout(k,1:k*8)];
end
[y,e,w] = KKK_EqualizationByGal_0_3(x,d,M,miu,2);
load('bn.mat');
bn = bn(2:35,1:ntr);
cout = cout(1:34,1:ntr);
figure;mesh(abs(bn-cout));
title('GAL 算法误差曲面');
xlabel('采样点/samples');
ylabel('阶次/M');
zlabel('误差/量化单位');

```

得到误差曲面如下图所示：

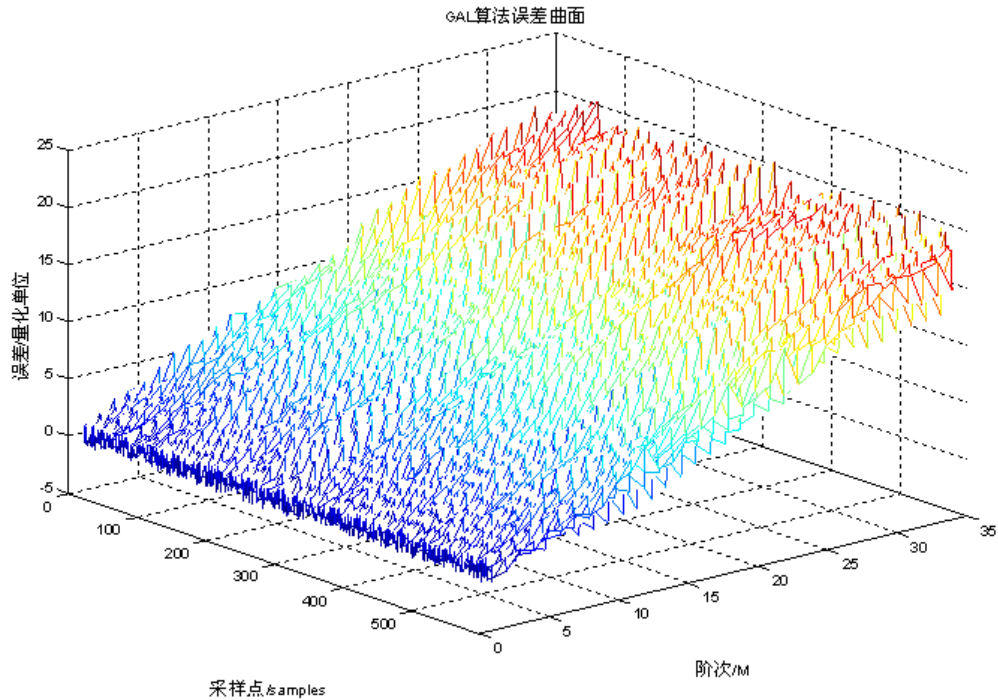


图 5-5 PL 计算结果与 matlab 计算结果误差曲面

从上图看到，GAL 算法在 PL 上实现后，与 matlab 计算结果误差绝对值小于 25，误差百

分比小于 3%，误差随着阶次增加而近似线性增长，而每一阶 GAL 的误差则随机变化。由此可得出结论，GAL 算法在有限字长的处理器上实现是可行的，计算误差处于可以接受范围内。

2. 速度测试

参数设置如下：

参数	值
数据长度	4375
训练序列长度	595
均衡阶次	35
步长因子	0.01
帧号	0~25
反复次数	10

按照 4.2 节操作，运行 GAL_NLMS 算法，记录运行时间，测试结果如下：

	GAL	NLMS	总计
耗时（s）	0.010	0.015	0.025

平均处理速度：每秒 40 帧数据。

5.3 目前性能指标

PS 与 PL 理论通信带宽： $150\text{MHz} \times 32\text{bit}/8 = 600 \text{ MB/s}$

PS 与 PL 实际通信带宽： $(4375/7+35) \times (1+35) \times 32\text{bit}/0.010\text{s} = 532.224 \text{ Mbps}$ （包含了系统内存拷贝、AXI_DMA 操作等开销）

最高输入数据率： $4375 \times 40 = 175 \text{ kHz}$

最高均衡器阶次：35

PL 峰值运算性能： $207 \times 166\text{MHz} = 34.362 \text{ GMACS}$

最大误差百分比： $<3\%$

6. 总结

本项目中实现的自适应均衡器是一类应用很广的信号处理模块。

从拿到板子到现在已经近 4 个月了，学习时遇到很多困难，经过队友间密切配合，全力攻关，最终一一克服。得到的经验和教训有：

1. 前期规划很重要

在项目初期，系统存在很多未知因素，队员对每个细节都不够了解，这时需要有一个统一的计划，将整个项目进行详细分工，规划好每一模块的功能、输入输出、精度要求，然后队员分头行动，全面而深入地对项目进行研究。

2. 善于查找资料

每个组员的方向定了之后，接下来就是广泛而细致地搜索资料。做硬件开发一定要先看最权威的官方 datasheet 和应用笔记，在做项目时遇到问题往往就是因为没注意到一些细节，如果之前看过 datasheet 一般就能避免出现一些低级错误。本项目用到了 AXI-DMA IP，

通过仔细阅读数据手册确定使用该 IP 而不是 AXI-CDMA IP，理由是前者实现了从内存映射到 Stream 模式转换，更适合本项目的信号处理流程，而后者没有这个功能。通过阅读相应的应用笔记还能参考别人成功的经验，减少出错的可能性。

3. 定期总结文档

为了便于组员交流，最好每个人都养成写文档的好习惯，将实验过程、预期结果、实测结果、问题分析记录下来，开会时统一进行讨论，这样可以大大提高效率，节省时间，而且写文档也是一种捋顺思路的过程，常常在写文档时发现自己程序存在的问题，并加以改正。

4. 遇到问题多求助网络

由于 Zynq 属于新事物，没有特别完善的开发资料可以查询，互联网上一些博客、教程、官方论坛等都是重要的消息来源和问题聚焦点，通过别人遇到的问题可以很大程度上给自己提供解决思路。同时，网络也是交流的工具，通过与其他组交流，可以发现共同问题，从而更容易得到解决。

5. 优化

算法优化需要对算法本身有深刻的了解，而且需要对硬件平台也有足够的认识，所以这部分工作开发难度较大，需要有一定经验。将作品进行优化需要很大的耐心，通过仔细研究得到线索。优化是一个循序渐进的过程，在改进的过程中会遇到种种困难，但如果前期准备做得好，这些困难都不是问题，一定能找到很好的解决办法。

7. 参考文献

- [1] Xilinx FPGA 开发实用教程，田耘等，清华大学出版社，北京，2008
- [2] 通信原理（第六版），樊昌信等，国防工业出版社，北京，2009
- [3] 嵌入式系统软硬件协同设计实战指南，陆佳华等，机械工业出版社，北京，2013
- [4] 无线通信原理与应用（第二版），Theodore S. Rappaport，电子工业出版社，北京，2006
- [5] Zynq-7000 EPP Technical Reference Manual, Xilinx, UG585(v1.2), 2012
- [6] VHDL 程序设计，曾繁泰等，清华大学出版社，北京，2000
- [7] 自适应滤波器原理（第四版），Simon Haykin，电子工业出版社，北京，2002
- [8] 数字电子技术基础（第五版），阎石，高等教育出版社，北京，2006
- [9] ZYNQ-7000 EPP Overview, Xilinx, DS190(v1.1.1), 2012
- [10] LogiCORE IP AXI DMA v6.03a, Xilinx, PG021, 2012
- [11] AXI Reference Guide, Xilinx, UG761(v14.3), 2012
- [12] ZedBoard:Zynq-7000 AP SoC Concepts, Tools, and Techniques, Xilinx, ZedBoard(v14.1), 2012
- [13] AMBA 4 AXI4-Stream Protocol v1.0, ARM, IHI0051A, 2010
- [14] Linux 窗口程序设计——Qt4 精彩实例分析，成洁等，清华大学出版社，2008