第1章 基于 AXI DMA 的多通道数据采集 TCP 传输系统

AD7606C-18 ADC

章节导读

本章主要以 AC880 为主体，讲解如何通过 PS 侧以太网控制器，实现基于 AD7606C-18 的八通道同步采集 TCP 传输系统。文章同样适配小梅哥旗下 ZYNQ 系列开发板/评估板，实际学习时，用户需要根据实际板卡型号，调整器件、以太网管脚路由、DDR 配置以及管脚约束。

设计将会使用 TCP 作为通信渠道，通过以太网上位机用户可以下发指令，对 ADC 内部寄存器进行配置，也可以通过指令设置采样通道、采样速率、采样数量以及使能采样。ADC 采样后的 18 位结果，会通过 DMA，完整地写入到 PS 侧的 DDR 中，最终，在采样完成后，通过 PL 以太网接口统一传输给 PC 上位机，交由用户做进一步处理。

1.1 ACM7606C 模块简介

设计使用 ACM7606C 模块实现数据采集，该模块实物如下图 1-1 所示，适用于小梅哥全系列 FPGA、SoC、Zynq 及 MPSoC 开发板。模块支持 ADI 公司两种不同型号的模数转换芯片，分别为 AD7606C-16 和 AD7606C-18。这两款芯片同属于 AD7606C 系列 8 通道同步采样 SAR ADC，二者引脚兼容、硬件电路通用，均支持每个通道最高 1MSPS 吞吐，仅在分辨率、噪声动态指标、线性度上存在差异。



图 1-1 ACM7606C 模块图

本次设计使用的是 AD7606C-18 型号芯片，因此下文将着重介绍该芯片内容。

1.1.1 AD7606C-18 芯片介绍

AD7606C-18 是一款高性能、单电源、18 位、8 通道双极性同步采样模数转换数据采集系统（DAS），其功能结构如下图 1-2 所示。作为一款高度集成的数据采集系统，AD7606C-18 内部集成了 8 个独立通道的模拟输入箝位保护、二阶抗混叠积分滤波器、跟踪保持放大器、18 位逐次逼近型（SAR）ADC 内核以及数字滤波器，非常适合对多通道信号有严格同步和高分辨率要求的工业物联网、电力线监测及电机程控系统。

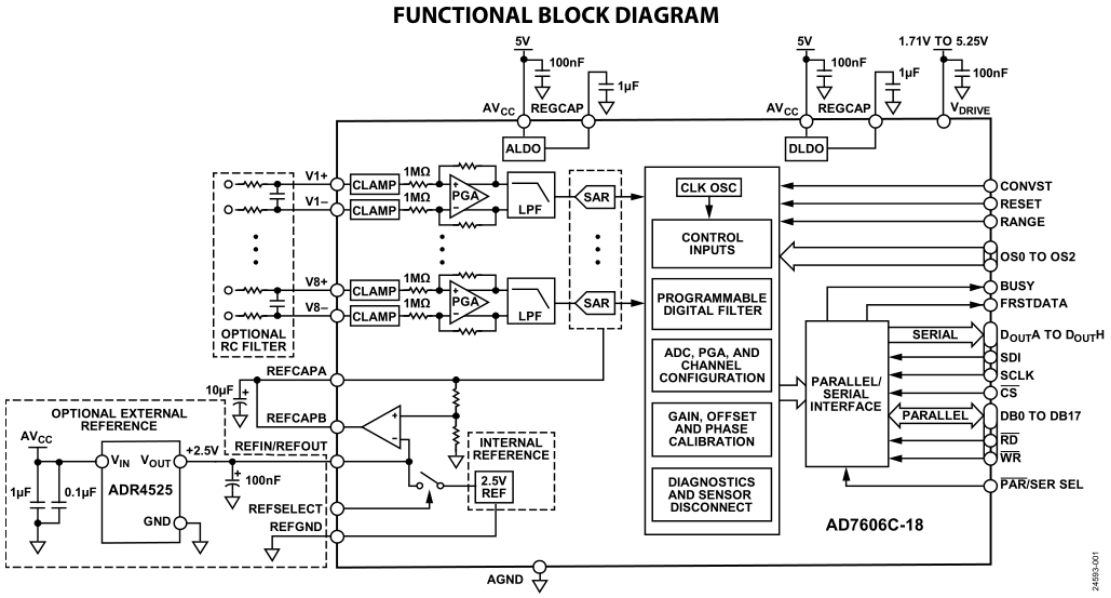

图 1-2 AD7606C-18 功能框图

AD7606C-18 采用 5V 单电源供电，支持多种双极性和单极性单端输入范围，能够满足不同类型传感器及前端信号调理电路的接入需求。芯片所有通道均可实现高达 1MSPS 的同步采样速率，且输入箝位保护电路可耐受最高 21V 的过压冲击。同时，芯片模拟输入阻抗较高，在多数应用场景下无需额外配置驱动运放或外部双极性电源即可工作，这为系统设计提供了较好的灵活性与集成度。

此外，AD7606C-18 还提供两种带宽模式：低带宽模式下，抗混叠滤波器的 3dB 截止频率为 25kHz；高带宽模式下，截止频率可达 220kHz。该带宽模式可通过软件模式下的寄存器进行配置，从而使芯片能够适应不同信号频段的采集需求。芯片还支持过采样功能，可根据应用场景选择不同的过采样倍率，以改善采样噪声和提高有效分辨率。

1.1.2 AD7606C-18 引脚/信号说明

芯片对外提供多个信号用于控制/数据读写（见图 1-2 右侧），一些常用引脚/信号见下表 1-1：

表 1-1 AD7606C-18 部分引脚/信号说明

引脚/信号	功能
OS0 to OS2	过采样模式引脚。硬件模式下用于选择过采样模式，软件模式下需全部置 1 以启用软件模式。
$\overline{PAR}$/SER SEL	并行/串口接口选择引脚。 逻辑低电平：并行接口。 逻辑高电平：串行接口
RANGE	模拟输入范围选择（仅硬件模式有效）。 逻辑高电平：±10V 单端 逻辑低电平：±5V 单端
REF SELECT	参考电压选择。 逻辑低电平：使用外部基准电压 逻辑高电平：使用内部基准电压

CONVST	转换启动。上升沿会触发采样保持与 ADC 转换，在软件模式下，可配置为外部过采样时钟。
$\overline{WR}$	并行写控制。用于软件模式下的寄存器写入控制，低电平有效。
RESET	复位。高电平有效，根据复位脉冲宽度，决定复位类型。 55ns < 脉宽 < 2us，部分复位，释放复位 50ns 后才可进行 ADC 转换 3.2us < 脉宽，完全复位，释放复位 274us 后才可进行 ADC 转换
$\overline{RD}$/SCLK	并行读控制/串行时钟输入。并行接口时，作为读控制信号，低电平有效。串行接口时，作为串行时钟输入脚。
$\overline{CS}$	片选。低电平有效。
BUSY	忙。在转换期间，Busy 持续输出高电平，直至转换完成。
FRSTDATA	首通道输出指示引脚。标记当前正在输出 V1（第 1 通道）采样数据，区分 8 通道读取时序，防止通道序号错乱
DB2 to DB4	并行输入输出数据（db[2:4]）
DB5 to DB8 / $D_{out}E$ to $D_{out}H$	并行输入输出数据（db[5:8]）/串行输出（$D_{out}E \sim D_{out}H$）
DB9 to DB12 / $D_{out}A$ to $D_{out}D$	并行输入输出数据（db[9:12]）/串行输出（$D_{out}A \sim D_{out}D$）
DB13 to DB15	并行输入输出数据（db[13:15]）
DB16 to DB17 / DB0 to DB1	并行输入输出数据（db[16:17]/db[0:1]）

其中，$\overline{PAR}$/SER SEL、RANGE、REF SELECT 分别用来控制 ADC 的接口类型、硬件模式下的模拟输入范围、参考电压源。而在 ACM7606C 模块上，这几个引脚则是通过跳线帽进行上下拉设置，如下图 1-3 所示：

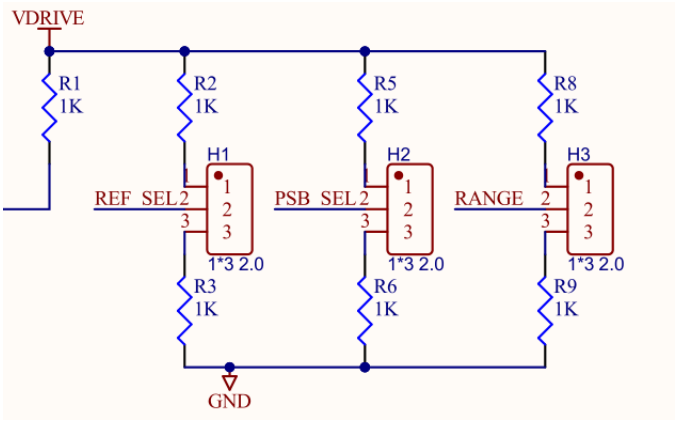

图 1-3 ACM7606C 跳线帽设置

对应地，跳线帽上下拉功能如下表 1-2 所示：

表 1-2 ACM7606C 跳线帽设置

信号名	上拉	下拉
REF SEL(H1)	内部基准电压	外部基准电压
PSB SEL(H2)	串行接口	并行接口
RANGE(H3)	模拟输入范围±10V 单端	模拟输入范围±5V 单端

ADC 的复位由 RESET 脚控制，根据复位高脉冲的脉宽，AD7606C-18 共有部分复位和全部复位两种复位策略。部分复位会清理掉数字逻辑内容，但不会清除用户配置；全部复位则会将芯片恢复到默认配置，等同于重新上电。

通过输出 CONVST 脉冲，主控设备可以控制 ADC 的转换启动，转换过程中 BUSY 引脚会一直保持高电平，直至转换完成。因此，在实际使用时，主控设备可以通过 BUSY 信号，来判断何时可以读取转换后的数据。

为了防止通道序号错乱，AD7606C-18 会通过 FRSTDATA 引脚对通道 1 输出的数据进行标识，当 FRSTDATA 为高电平时，代表此时正在输出通道 1 转换后的数据。

数据的获取共有两种方式，一种是通过串行接口进行访问，另一种则是通过并行接口进行访问。接口模式通过$\overline{PAR}$/SER SEL 引脚控制，低电平代表使用并行接口，高电平则代表使用串行接口。

使用串行接口时，$\overline{RD}$将作为串行时钟输入脚，数据的读写通过$D_{out}A$~$D_{out}H$完成。用户可以通过在软件模式下对 0x02 寄存器进行配置，选择输出通道数。芯片支持最多 8 路输出（对应$D_{out}A$~$D_{out}H$），相关寄存器配置如下图 1-4 所示：

DOUTx Format	Address 0x02, Bit 4	Address 0x02, Bit 3
1 DOUTx	0	0
2 DOUTx	0	1
4 DOUTx	1	0
8 DOUTx	1	1

图 1-4 串行接口输出格式配置

使用并行接口时，$\overline{WR}$将作为写控制信号，$\overline{RD}$将作为读控制信号，数据的读写通过 16 位的 DB 总线实现。由于 AD7606C-18 为 18 位高分辨率 ADC，因此，需要两次读脉冲才能完整获取单个通道转换的数据，如下图 1-5 所示：

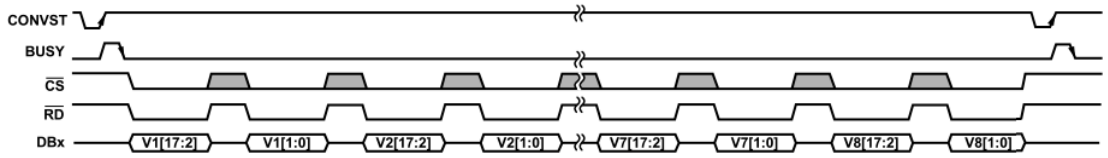

图 1-5 AD7606C-18 并行接口读取 18 位转换结果时序

第一个读脉冲，DB 会输出转换数据的高 16 位，第二个读脉冲则通过 DB[17:16]，输出转换数据的低两位。通过将两次读脉冲 DB 输出的数据进行拼接，主控设备便可以得到完整的 18 位数据。

1.1.3 AD7606C-18 工作模式

AD7606C-18 共有两种工作模式，分别是硬件模式和软件模式。软硬件模式由过采样引脚 OS[2:0]决定，具体如下表 1-3：

表 1-3 OS 编码

OS2	OS1	OS0	AD7606C-18
0	0	0	无过采样
0	0	1	2
0	1	0	4

0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	使能软件模式

硬件模式下，主控设备只能控制 ADC 对模拟信号进行采样，从而获取采样数据。而在软件模式下，又细分为了 ADC 模式与寄存器模式。ADC 模式与硬件模式类似，只能控制 ADC 进行模拟采样；而寄存器模式下，则可以通过配置 AD7606C-18 内部的寄存器，对 ADC 的功能以及各个通道进行设置。

但需要注意的是，配置 ADC 内部寄存器需要通过$\overline{WR}$实现，因此，设置软件模式前，需要先拔掉 ACM7606C 上 H5 的跳线帽， 取消$\overline{WR}$与 CONVST 的短接，这样主控设备才能单独对$\overline{WR}$引脚进行控制。如下图 1-6 所示：

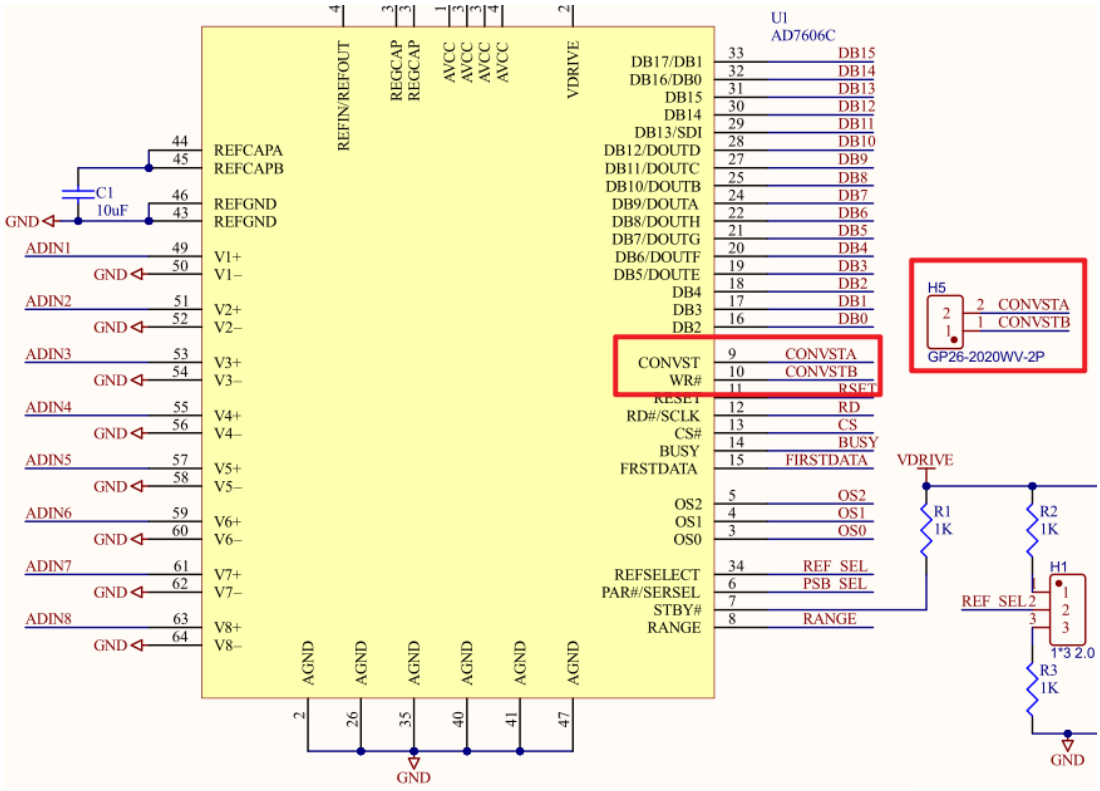

图 1-6 软件模式跳线帽设置

根据所选择接口类型的不同，硬件/软件模式下对于 DB 总线的复用也有所不同，具体如下图 1-7 所示：

Pin Mnemonic	Pin No.	Parallel Interface			Serial Interface		
			Software Mode			Software Mode	
		Hardware Mode	ADC Mode	Register Mode	Hardware Mode	ADC Mode	Register Mode
DB2 to DB4	16 to 18	DB2 to DB4		Register data	N/A[1]	N/A	
DB5/D_{OUT}E	19	DB5		Register data	N/A	$D_{OUT}E$[2]	Unused
DB6/D_{OUT}F	20	DB6		Register data	N/A	$D_{OUT}F$[2]	Unused
DB7/D_{OUT}G	21	DB7		Register data	N/A	$D_{OUT}G$[2]	Unused
DB8/D_{OUT}H	22	DB8		Register data	N/A	$D_{OUT}H$[2]	Unused
DB9/D_{OUT}A	24	DB9		Register data (MSB)	$D_{OUT}A$	$D_{OUT}A$	$D_{OUT}A$
DB10/D_{OUT}B	25	DB10		ADD0	$D_{OUT}B$	$D_{OUT}B$[3]	Unused
DB11/D_{OUT}C	27	DB11		ADD1	N/A	$D_{OUT}C$[4]	Unused
DB12/D_{OUT}D	28	DB12		ADD2	N/A	$D_{OUT}D$[4]	Unused
DB13/SDI	29	DB13		ADD3	N/A	Unused	SDI
DB14 to DB16	30	DB14		ADD4	N/A		N/A
DB15	31	DB15		ADD5	N/A		N/A
DB16/DB0	32	DB16/DB0[5]		ADD6	N/A		N/A
DB17/DB1	33	DB17/DB1[5]		R/$\overline{W}$	N/A		N/A

图 1-7 DB 与工作模式

以并行接口为例，在硬件模式以及 ADC 模式下时，DB 用来传输 ADC 转换后的数据。由于 DB 总线只有 16 位，所以传输会分为两拍，第一拍输出高 16 位，第二拍复用 DB[17:16]，输出采样数据的低 2 位。

而当 ADC 处于寄存器模式时，传输只需要一拍。DB17 作为读写标志位，DB[16:10]用来传输寄存器地址，DB[9:2]用来传输寄存器数据。

1.1.4 读写时序与模式转换

对于 AD7606C-18 而言，其经典的工作流程为：上电 -> 复位 -> 采样 -> 转换 -> 输出 -> 采样 -> 转换 -> 输出 -> .....。而与之对应的时序，则如下图 1-8 所示，相关时序参数如下图 1-9 所示：

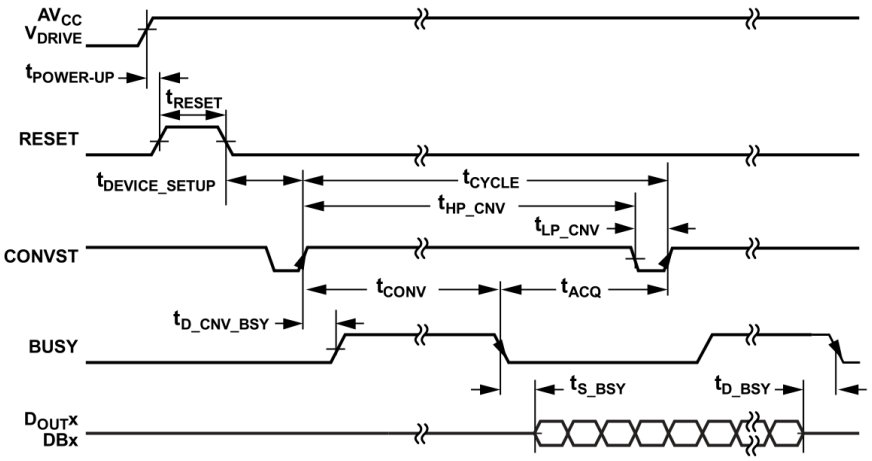

图 1-8 通用时序图

Parameter	Min	Typ	Max	Unit	Description
t_{CYCLE}	1			µs	Minimum time between consecutive CONVST rising edges (excluding oversampling modes)[1]
t_{LP_CNV}	10			ns	CONVST low pulse width
t_{HP_CNV}	10			ns	CONVST high pulse width
$t_{D_CNV_BSY}$			22	ns	CONVST high to BUSY high delay time
t_{S_BSY}	0			ns	Minimum time from BUSY falling edge to $\overline{RD}$ falling edge setup time (in parallel interface) or to MSB being available on the $D_{OUT}x$ line (in serial interface)
t_{D_BSY}			25	ns	Maximum time between last $\overline{RD}$ falling edge (in parallel interface) or last LSB being clocked out (serial interface) and the following BUSY falling edge, read during conversion
t_{ACQ}	0.35			µs	Acquisition time
t_{CONV}	0.5		0.65	µs	Conversion time, no oversampling
	1.7		1.75	µs	Oversampling by 2
	3.6		3.8	µs	Oversampling by 4
	7.6		7.85	µs	Oversampling by 8
	15.5		16	µs	Oversampling by 16
	31.0		32.5	µs	Oversampling by 32
	62.75		65.0	µs	Oversampling by 64
	126		130	µs	Oversampling by 128
	252		256	µs	Oversampling by 256
t_{RESET}					
Partial Reset	55		2000	ns	Partial RESET high pulse width
Full Reset	3200			ns	Full RESET high pulse width
t_{DEVICE_SETUP}				µs	Time between RESET falling edge and first CONVST rising edge
Partial Reset	50			ns	
Full Reset	274			µs	
$t_{WAKE-UP}$					Wake-up time after standby and shutdown mode (see Figure 86)
Standby	1			µs	
Shutdown	10			ms	
$t_{POWER-UP}$	10			ms	Time between stable AV_{CC} and V_{DRIVE} and assert of RESET

图 1-9 通用时序参数

ADC 从上电到稳定需要至少保持$t_{power-up}$，也就是 10ms。随后，用户才可以对 ADC 进行复位， 根据复位模式不同，需要保持 Reset 信号高低电平不同时间，直到复位完成且设备被唤醒。紧接着主控设备便可以通过 CONVST 信号产生低脉冲，控制 ADC 开始采样。ADC 会在 CONVST 信号的上升沿开始采集模拟信号，并进行转换，在采集并转换的过程中，BUSY 信号会持续拉高，直至转换完成，数据会通过 DB 总线输出。

单次转换的最小周期为 1us，对应每个通道的最大采样率 1MSPS，通过控制 CONVST 信号的频率，主控设备便能控制 ADC 的采样频率。至于 DB 总线上，数据的写入和读出，根据接口类型的不同，需要遵循不同的时序，这里我们分开进行说明。

1.1.4.1并行接口时序

通过设置 ACM7606C 模块的 H2 跳线帽为下拉，AD7606C-18 可以被配置为并行接口模式。此时，主控制器能够通过$\overline{RD}$信号对芯片内部转换结果进行读取（硬件模式和 ADC 模式）。其时序如下图 1-10 所示：

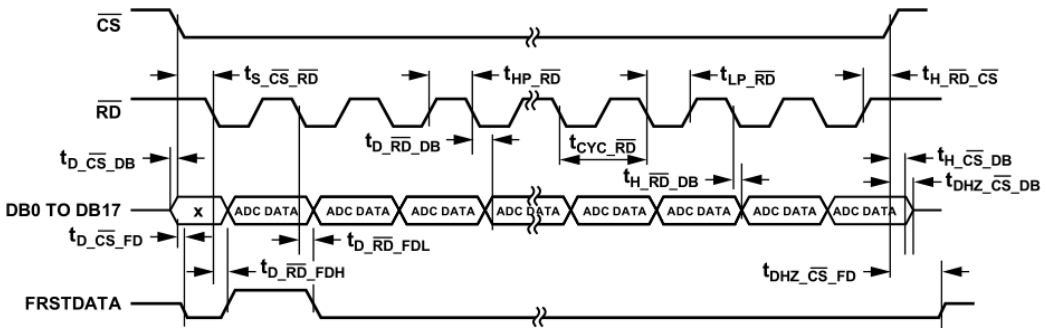

图 1-10 并行接口读数据时序

要想读取转换数据，主控制器需要拉低$\overline{CS}$信号，对 ADC 进行片选。转换后的数据会在$\overline{RD}$下降沿的$t_{D_\overline{RD}_DB}$后，更新到 DB 总线上，主控制器需要在数据稳定后，也就是$\overline{RD}$的上升沿将数据读出。由于 DB 总线只有 16 位，因此，单个通道的转换结果需要通过两次读操作才能完整获取。为了避免多通道采样过程中出现通道顺序错乱的问题，FRSTDATA 会在通道 1 数据输出期间持续保持高电平。实际采集时，可以根据 FRSTDATA 对通道进行定位。下图 1-11 给出了并行接口读数据时的时序参数：

Parameter	Min	Typ	Max	Unit	Description
$t_{S_\overline{CS}_\overline{RD}}$	0			ns	$\overline{CS}$ falling edge to $\overline{RD}$ falling edge setup time
$t_{H_\overline{RD}_\overline{CS}}$	0			ns	$\overline{RD}$ rising edge to $\overline{CS}$ rising edge hold time
$t_{HP_\overline{RD}}$	10			ns	$\overline{RD}$ high pulse width
$t_{LP_\overline{RD}}$	10			ns	$\overline{RD}$ low pulse width
$t_{HP_\overline{CS}}$	10			ns	$\overline{CS}$ high pulse width
$t_{D_\overline{CS}_DB}$			35	ns	Delay from $\overline{CS}$ until DBx three-state disabled
$t_{D_\overline{RD}_DB}$					Data access time after falling edge of $\overline{RD}$
			30	ns	V_{DRIVE} > 2.7 V
			25	ns	V_{DRIVE} < 2.7 V
$t_{H_\overline{RD}_DB}$	12			ns	Data hold time after falling edge of $\overline{RD}$
$t_{DHZ_\overline{CS}_DB}$			40	ns	$\overline{CS}$ rising edge to DBx high impedance
$t_{CYC_\overline{RD}}$	30			ns	$\overline{RD}$ falling edge to next $\overline{RD}$ falling edge
$t_{D_\overline{CS}_FD}$			20	ns	Delay from $\overline{CS}$ falling edge until FRSTDATA three-state disabled
$t_{D_\overline{RD}_FDH}$			30	ns	Delay from $\overline{RD}$ falling edge until FRSTDATA high
$t_{D_\overline{RD}_FDL}$			30	ns	Delay from $\overline{RD}$ falling edge until FRSTDATA low
$t_{DHZ_\overline{CS}_FD}$			25	ns	Delay from $\overline{CS}$ rising edge until FRSTDATA three-state enabled
$t_{S_\overline{CS}_\overline{WR}}$	0			ns	$\overline{CS}$ to $\overline{WR}$ setup time
$t_{HP_\overline{WR}}$	2			ns	$\overline{WR}$ high pulse width
$t_{LP_\overline{WR}}$	35			ns	$\overline{WR}$ low pulse width
$t_{H_\overline{WR}_\overline{CS}}$	0			ns	$\overline{WR}$ hold time
$t_{S_DB_\overline{WR}}$	5			ns	Configuration data to $\overline{WR}$ setup time
$t_{H_\overline{WR}_DB}$	5			ns	Configuration data to $\overline{WR}$ hold time
$t_{CYC_\overline{WR}}$	180			ns	Configuration data settle time, $\overline{WR}$ rising edge to next $\overline{WR}$ rising edge

图 1-11 并行接口读写时序参数

除了读取 ADC 转换后的数据外，在寄存器模式下，AD7606C-18 还支持对内部寄存器进行配置与读取，从而实现对过采样倍率、输入范围、带宽模式以及通道相关参数的灵活设置。

要想进入寄存器模式，需要先将 OS[2:0]设置为 3'b111，让 AD7606C-18 进入到软件模式。此时，ADC 默认处于软件模式下的 ADC 模式。主控设备还需要通过$\overline{WR}$和$\overline{RD}$信号读取任意一个寄存器的值，这样 ADC 才能进入到寄存器模式。当寄存器读写完成后，主控设备只需在一个$\overline{WR}$周期内，控制 DB 总线为低电平，便能退出寄存器模式。寄存器模式下的读写时序如下图 1-12 所示：

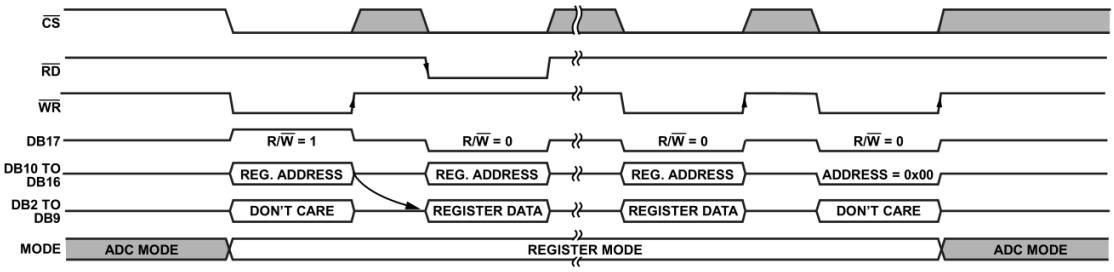

图 1-12 并行接口读写寄存器时序

不同于读取采样数据，寄存器模式下 DB 总线传输的是 16 位的寄存器访问数据（DB[17:2]），即每个指令字只需要一次传输。其中，DB17 为读写位，1 代表是读指令，0 表示写指令；DB[16:10]为待读写的寄存器地址；DB[9:2]为写入/读出的寄存器数据。

主控设备读取 ADC 寄存器需要分两步实现：

（1） 通过$\overline{CS}$和$\overline{WR}$信号，发送读指令。在$\overline{WR}$拉低后，设置 DB17 为 1，通过 DB[16:10]给出待读取的寄存器地址。数据会在$\overline{WR}$上升沿被 ADC 锁存。

（2） 通过$\overline{CS}$和$\overline{RD}$信号，接收 ADC 回读的数据。此时，主控制器需要释放 DB 总线，ADC 会在$\overline{RD}$低电平时，将 DB17 设置为 0，并通过 DB[16:10]输出寄存器地址，通过 DB[9:2]输出寄存器数据。

而写 ADC 寄存器，只需要在$\overline{CS}$和$\overline{WR}$信号拉低时，将 DB17 设置为 0，将待修改的寄存器地址和数据分别更新到 DB[16:10]和 DB[9:2]上即可，ADC 会在$\overline{WR}$信号的上升沿对数据锁存，进而配置寄存器。

AD7606C-18 对于写寄存器的时序要求如下图 1-13 所示：

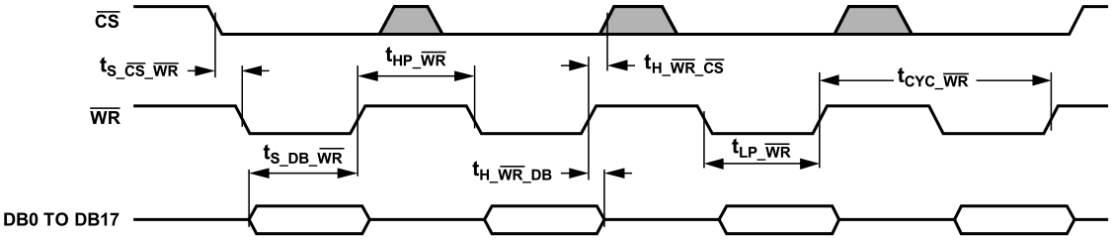

图 1-13 并行接口写寄存器时序图

有关写寄存器时的时序参数，可参考图 1-11。

1.1.4.2串行接口时序

通过设置 ACM7606C 模块的 H2 跳线帽为上拉，AD7606C-18 可以被配置为串行接口。串行接口下，$\overline{RD}$将作为串行时钟输入信号，ADC 转换数据通过 DoutA~DoutH 输出，用户指令则是通过 SDI 写入。与并行接口一样，串行接口下，AD7606C-18 同样分为硬件与软件两个大模式，软件模式下又分为 ADC 模式与寄存器模式，不同模式下的接口对应关系可以参考图 1-7。

在硬件模式下，AD7606C-18 固定使用 DoutA 和 DoutB 输出转换数据，如图 1-14 所示：

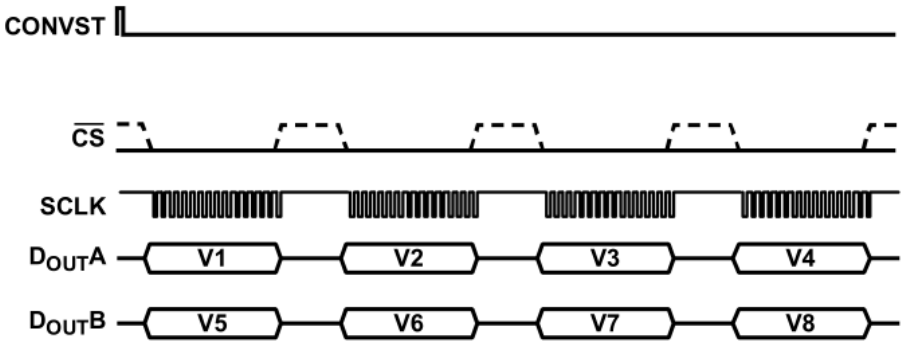

图 1-14 串行接口硬件模式输出格式

而在软件模式下，则可以通过配置 0x02 寄存器，对输出通道数进行修改，如下表 1-4 所示：

表 1-4 串行接口输出通道设置

DoutX Format	Address 0x02，Bit 4	Address 0x02，Bit 3
1 DoutX	0	0
2 DoutX	0	1
4 DoutX	1	0
8 DoutX	1	1

通过设置 0x02 寄存器的 bit[4:3]，可以将 ADC 设置为 1 通道、2 通道、4 通道、8 通道输出。不同输出通道对应的数据输出格式则如下图 1-15 和图 1-16 所示：

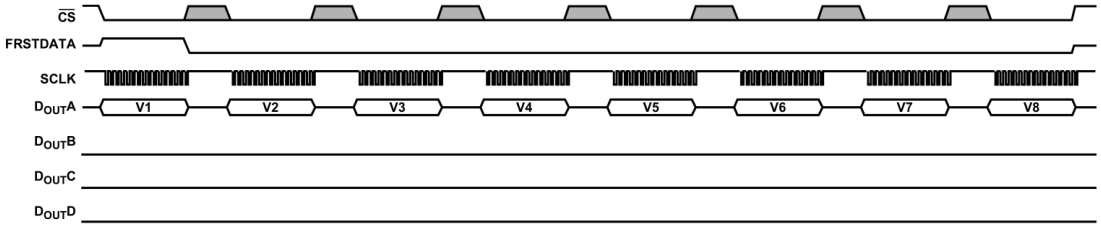

图 1-15 串行接口，1 通道输出

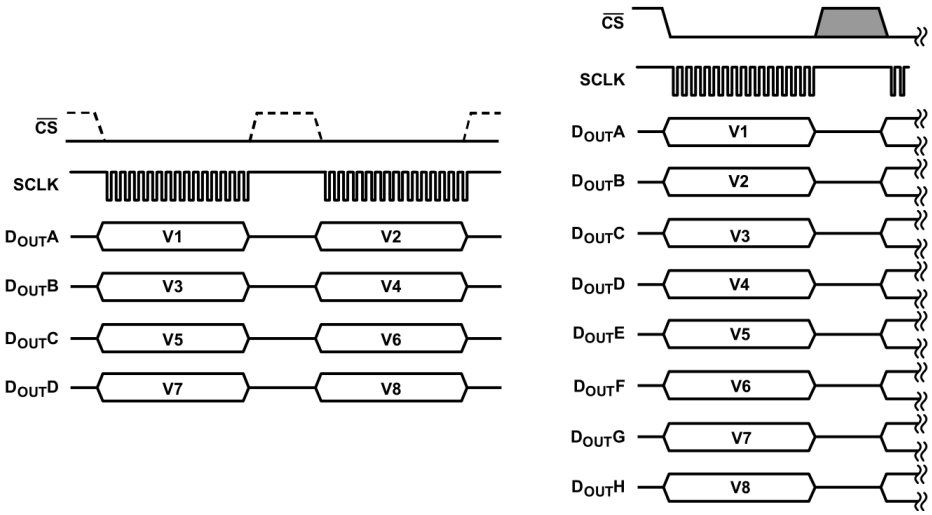

图 1-16 串行接口，4 通道（左）与 8 通道（右）输出

单个通道的传输时序如下图 1-17 所示：

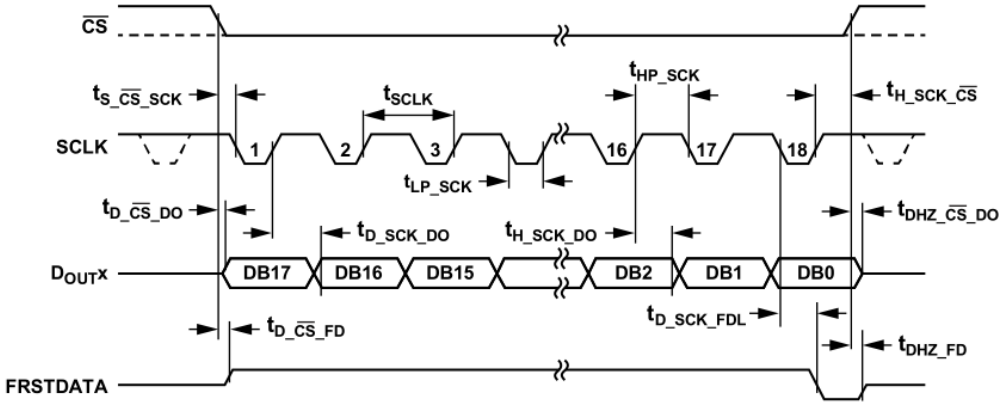

图 1-17 串行接口，硬件/ADC 模式时序

Dout 总线会在 CS 的下降沿脱离三态， 然后通过 SCLK 串行时钟，将数据逐位移出。每一个 SCLK 下降沿都会触发内部移位寄存器更新，将下一位数据输出到 DOUTx 总线上，数据位按 DB17 → DB16 → DB15 → … → DB1 → DB0 的顺序从高到低依次输出，主机可在 SCLK 上升沿对数据进行锁存采样，此时时序裕量最充足、读取最稳定。由于 ADC 转换数据为 18 位，所以，读取单个通道转换数据，需要 18 个 SCLK 周期。

对应地，图 1-17 中的时序参数如下图 1-18 所示：

Parameter	Min	Typ	Max	Unit	Description
f_{SCLK}					SCLK frequency, $f_{SCLK} = 1/t_{SCLK}$
			60	MHz	V_{DRIVE} > 2.7 V
			40	MHz	V_{DRIVE} < 2.7 V
t_{SCLK}	$1/f_{SCLK}$			µs	Minimum SCLK period
$t_{S_\overline{CS}_SCK}$	2			ns	$\overline{CS}$ to SCLK falling edge setup time
$t_{H_SCK_\overline{CS}}$	2			ns	SCLK to $\overline{CS}$ rising edge hold time
t_{LP_SCK}	$0.4 \times t_{SCLK}$			ns	SCLK low pulse width
t_{HP_SCK}	$0.4 \times t_{SCLK}$			ns	SCLK high pulse width
$t_{D_\overline{CS}_DO}$			18	ns	Delay from $\overline{CS}$ until $D_{OUT}x$ three-state disabled
$t_{D_SCK_DO}$					Data out access time after SCLK rising edge
			17	ns	V_{DRIVE} > 2.7 V
			25	ns	V_{DRIVE} < 2.7 V
$t_{H_SCK_DO}$					Data out hold time after SCLK rising edge
	7			ns	V_{DRIVE} > 2.7 V
	10			ns	V_{DRIVE} < 2.7 V
$t_{S_SDI_SCK}$	9			ns	Data in setup time before SCLK falling edge
$t_{H_SCK_SDI}$	0			ns	Data in hold time after SCLK falling edge
$t_{DHZ_\overline{CS}_DO}$			25	ns	$\overline{CS}$ rising edge to $D_{OUT}x$ high impedance
$t_{\overline{WR}}$	25			ns	Time between writing and reading the same register or between two writes, if f_{SCLK} >50 MHz
$t_{D_\overline{CS}_FD}$			16	ns	Delay from $\overline{CS}$ until $D_{OUT}x$ three-state disabled or delayed from $\overline{CS}$ until MSB valid
$t_{D_SCK_FDL}$			18	ns	18th SCLK falling edge to FRSTDATA low
t_{DHZ_FD}			20	ns	$\overline{CS}$ rising edge until FRSTDATA three-state enabled

图 1-18 串行接口，硬件/ADC 模式时序参数

与并行接口时一样，串行接口下，要想进入寄存器模式，同样需要进行一次读寄存器操作。该操作由两个 16 位指令帧组成，其中第一帧的需求如下：

- 通过 SDI 引脚串行移入的第 1 位必须置 0，用于使能写命令；
- 通过 SDI 引脚串行移入的第 2 位必须置 1，用于选择读命令操作；
- 通过 SDI 引脚串行移入的第 3~8 位，为待读取的寄存器地址；对应寄存器的内容，将在下一个指令帧中从 DOUTA 引脚串行移出；
- 后续的第 9~16 位 SDI 输入数据会被芯片忽略，无实际作用。

如果 AD7606C-18 当前处于 ADC 采样模式，在第 9~16 位的时钟周期内，各 DOUTx 引脚仍会持续输出 ADC 采样数据；该帧传输完成后，芯片自动切换至寄存器配置模式。如果芯片已处于寄存器模式，则无论上一帧指令是读操作还是写操作，DOUTx 引脚都会返回上一次寻址的寄存器内容。而用户要想退出寄存器模式、返回正常 ADC 采样状态，只需将 SDI 引脚保持低电平，并持续输入 16 个 SCLK 时钟周期即可。

对应时序如下图 1-19 和图 1-20 所示：

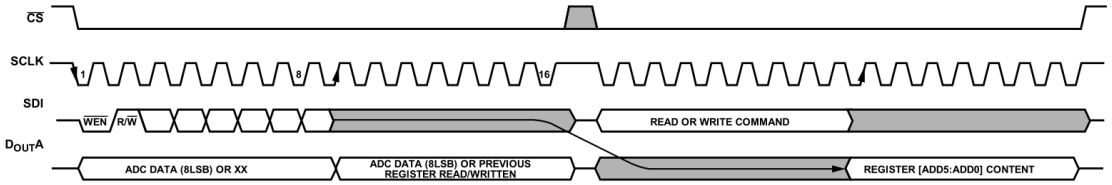

图 1-19 串行接口读命令：第一帧提供寄存器地址，第二帧输出寄存器内容

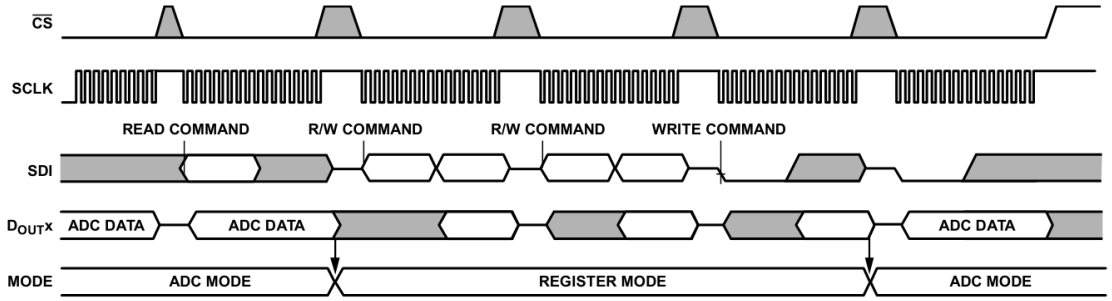

图 1-20 AD7606C-18 寄存器模式

只有进入了寄存器模式，才能对 ADC 寄存器进行配置。单次寄存器写操作由 1 个 16 位 SPI 帧完成，该帧的位定义及时序如下：

- SDI 串行移入的第 1 位必须置 0，用于使能写命令；
- SDI 串行移入的第 2 位（读 / 写控制位）必须为 0，标识当前为写操作；
- SDI 串行移入的 ADD5~ADD0 共 6 位，为待写入的寄存器地址；
- 后续 8 位（DIN7~DIN0）为要写入目标寄存器的配置数据。

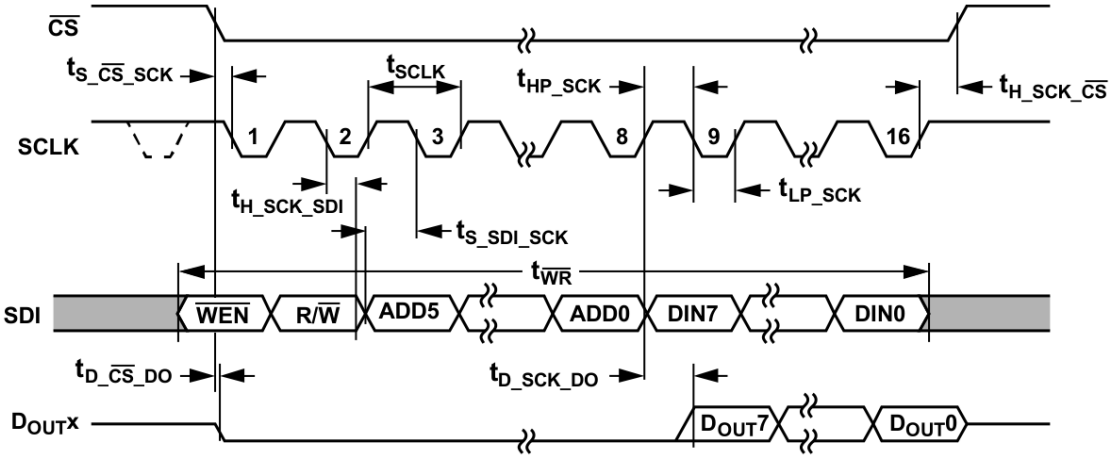

图 1-21 串行接口，寄存器读写时序

用户通过 SDI 写入的数据会在 SCLK 的下降沿被芯片锁存，在传输过程中，DoutA（寄存器模式下仅 DoutA 有效），会输出上一帧操作所寻址的寄存器内容。

1.1.5 ADC 传递函数

根据设置的模拟输入量程不同，ACM7606C 有着两套输出编码规则。对于单端单极性模拟输入量程，输出采用二进制码；对于单端双极性模拟输入量程，输出采用二进制补码。两种编码格式对应的理想传输特性曲线如下图 1-22 所示：

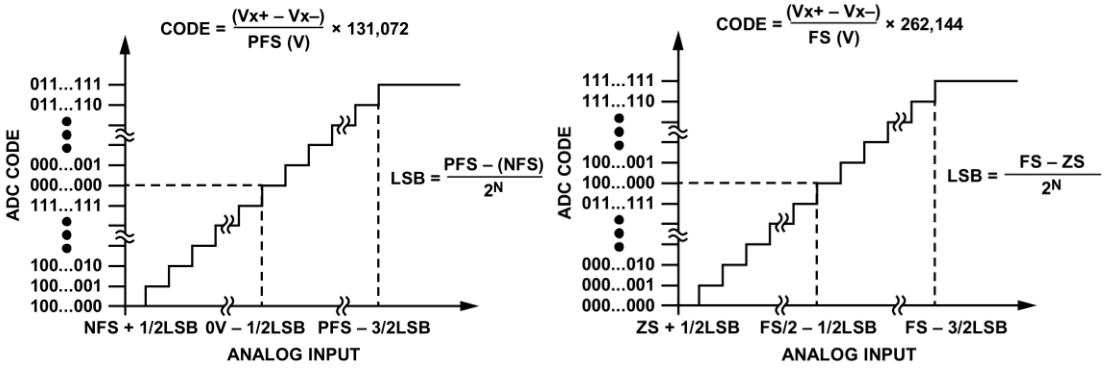

图 1-22 ADC 传递函数，单极性模拟输入（左）与双极性模拟输入（右）

下表 1-5 给出了不同模拟输入范围下的满量程点、中点、负满量程点以及对应的最小分辨电压 LSB。由表中数据可以看出，在双极性输入模式下，模拟输入以 0 V 为中点，正、负满量程分别对应输出范围的两端；在单极性输入模式下，输入范围以 0 V 为起点，满量程对应输入范围的上限。由于 ADC 的输出位宽固定，因此不同量程下的 LSB 会随输入范围变化，量程越小，分辨率越高。

表 1-5 模拟输入范围与 ADC 传递函数

模拟输入范围	PFS (V)	Midscale (V)	NFS (V)	LSB (μV)
单端，双极性				
±12.5 V	+12.5	0	−12.5	95.36
±10 V	+10	0	−10	76.3
±6.25 V	+6.25	0	−6.25	47.7
±5 V	+5	0	−5	38.1
±2.5 V	+2.5	0	−2.5	19
单端，单极性				
0 V to 12.5 V	12.5	6.25	0	47.7
0 V to 10 V	10	5	0	38.1
0 V to 5 V	5	2.5	0	19

在本章所设计的 AXI DMA 数据采集系统中，ADC 传递函数不仅决定了采样值的物理意义，也直接影响到数据缓存、通道排序以及上位机显示结果的解释方式。基于上述器件特性，下面将结合 ACM7606C 模块的实际应用场景，对多通道 ADC 数据采集实验进行介绍。

1.2 实验介绍

本实验使用 ACM7606C 模块的并行接口和软件模式。PL 读取每个通道的完整 18 位转换结果，将选中通道的样本打包后送入 AXI DMA。DMA 经 HP0 将数据写入 PS 侧 DDR。

以太网通信使用 PS 内部的以太网控制器，通过 EMIO 将 GMII 接口引出至 PL，经 GMII to RGMII IP 转换后连接板载以太网 PHY，实现与上位机的 TCP 通信。

设计整体由 PL 和 PS 两部分构成。PL 端负责 ACM7606C 模块的驱动、数据采集和 DMA 数据通路的构建，同时完成以太网 GMII 与 RGMII 接口的转换。PS 端通过 C 程序运行 lwIP 协议栈，接收并解析上位机下发的 TCP 指令，通过 AXI 接口配置 PL 端的采样参数；启动 AXI DMA 数据搬运，并通过 DMA 中断确认传输完成；随后读取 DDR 中的采样数据，通过 TCP 发送至上位机。

为了确保指令的准确性，指令需要按照下表 1-6 所示的帧格式发送：

表 1-6 指令帧格式

帧头		指令类型	数据				帧尾
0x55	0xA5	Addr [7:0]	data[31:24]	data[23:16]	data[15:8]	data[7:0]	0xF0

其中，指令类型 Addr 用于表明指令的作用，不同的值代表不同功能，如下：

表 1-7 指令类型

指令类型	功能
0	restart，重采样脉冲信号，代表一次传输的开始
1	ChannelSel，通道选择，每一位对应一个通道
2	DataNum，本次请求的采样字节数
3	ADC_Speed_Set，ADC 采样速率设置
4	ADC_Soft_Mode，ADC 软件模式下的配置
5～254	预留
255	devinfo，设备信息查询

指令 0~3 用于 ADC 采样控制，指令相较简单。指令 4 则用于寄存器配置，该指令的位定义如下图 1-23 所示：

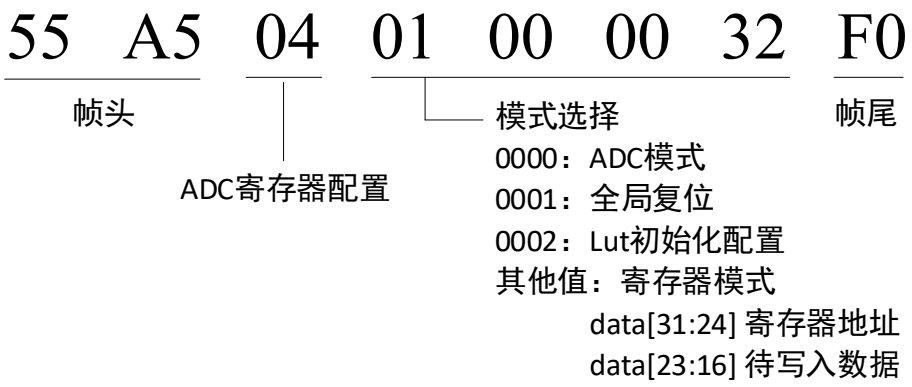

图 1-23 指令 4 的位定义

实际运行时，只需要根据需求发送对应指令即可，当所有指令发送完毕后，发送指令 0，写入任意值，便能驱使 ADC 以新的配置方式工作。

1.2.1 PL 端逻辑系统设计

为了便于说明，本文先完成硬件逻辑系统的搭建，最后，再基于完整的系统为大家讲解整个系统的工作流程与设计思路。

1.2.1.1IP 添加与配置

本次设计所需添加的 IP 如下图 1-24 所示，需要说明的是，如果开发板上使用的是 RTL8211E 芯片，则不需要添加其中的 ft_gmii_retime 模块：

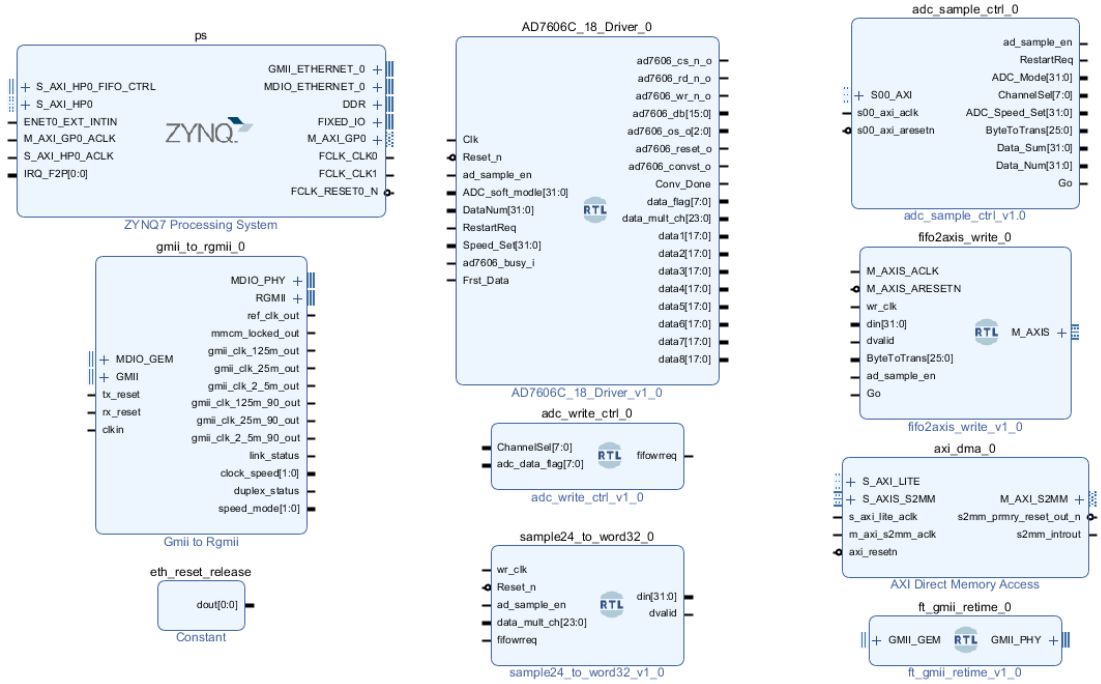

图 1-24 设计所需 IP

其中，各个模块在本次设计中所担任的功能如下：

表 1-8 模块说明

模块	功能
ZYNQ 核	提供 PS 处理系统，运行 TCP 通信程序，通过 AXI 接口配置采样参数，并通过 HP0 接口接收 DMA 写入 DDR 的采样数据。
AXI DMA	将 PL 输出的 AXI4-Stream 采样数据搬运至 PS 侧 DDR，并在传输完成后向 PS 发出中断。
Gmii to Rgmii	将 PS 通过 EMIO 引出的 GMII 接口转换为 RGMII 接口，连接板载以太网 PHY。
Constant	输出固定高电平，使以太网 PHY 的低有效复位信号保持释放状态。
fifo2axis_write	缓存打包后的采样数据，根据传输字节数生成 AXI4-Stream 数据流及帧结束信号，送入 AXI DMA。
adc_write_ctrl	根据通道选择配置和采样数据有效标志，生成选中通道的数据写入使能。
sample24_to_word32	将选中通道的 24 位样本连续拼接为 32 位数据，适配后级 FIFO 的写入位宽。
AD7606C_18_Driver	AD7606C-18 软件驱动模块，实现 ADC 的复位、寄存器配置和采样时序控制，通过并行接口读取各通道完整的 18 位转换结果，并输出数据有效标志。
adc_sample_ctrl	ADC 采样控制模块，负责接收 PS 的用户指令，并将其转化为 ADC 控制信号
ft_gmii_retime	对 GMII 接收通路进行重定时，在接收时钟下降沿寄存 RXD、`RX_DV` 和 RX_ER 后送入 PS GEM，时钟和发送信号直接透传。 注：RTL8211E 芯片不需要添加该模块

其中，adc_write_ctrl、AD7606C_18_Driver、sample24_to_word32、fifo2axis_write 是由 Verilog 模块设计转换而来的 IP（通过 add module 的方式添加）。AD7606C_18_Driver 负责 ADC 读写，adc_write_ctrl 根据通道掩码和 data_flag 产生采样写请求，sample24_to_word32 将 3 字节样本拼成 32 位字，fifo2axis_write 将这些字送入 AXI4-Stream。读者在添加 IP 后，可以在 Sources 中查看到这些模块的源码，如下图 1-25：

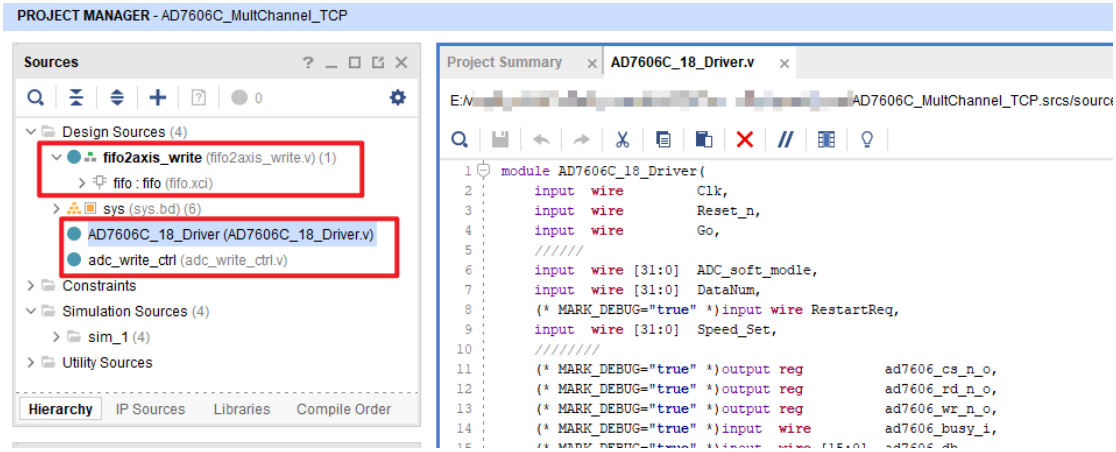

图 1-25 查看部分 IP 源码

需要注意的是，这里， fifo2axis_write 还包含了一个 fifo，添加 fifo2axis_write 模块后，需要为其创建一个名为 fifo 的 FIFO IP 核，该 IP 的配置如下：

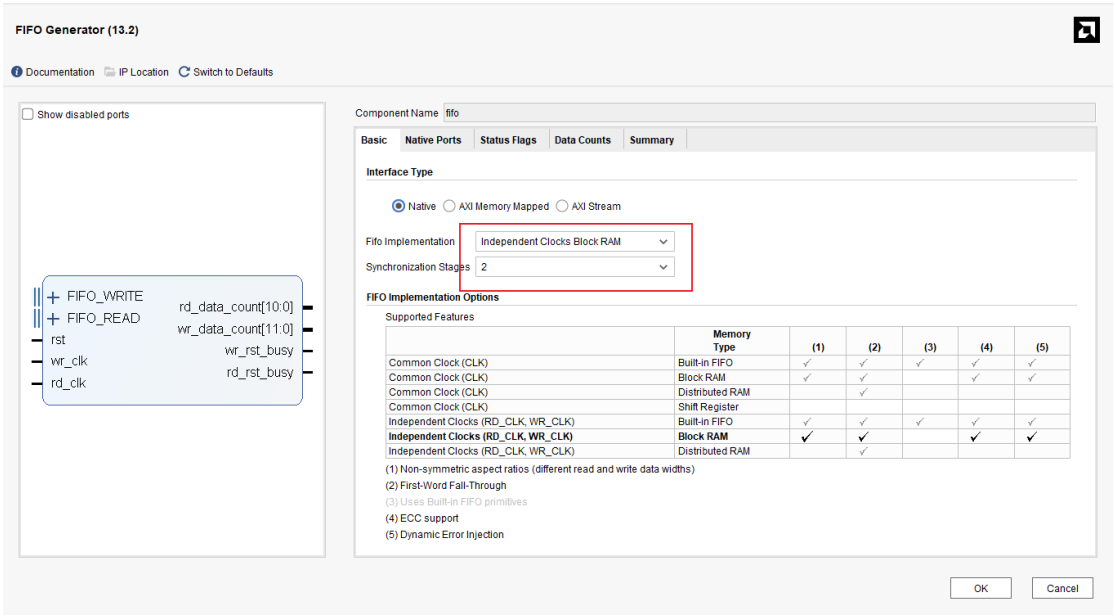

图 1-26 fifo 配置 1

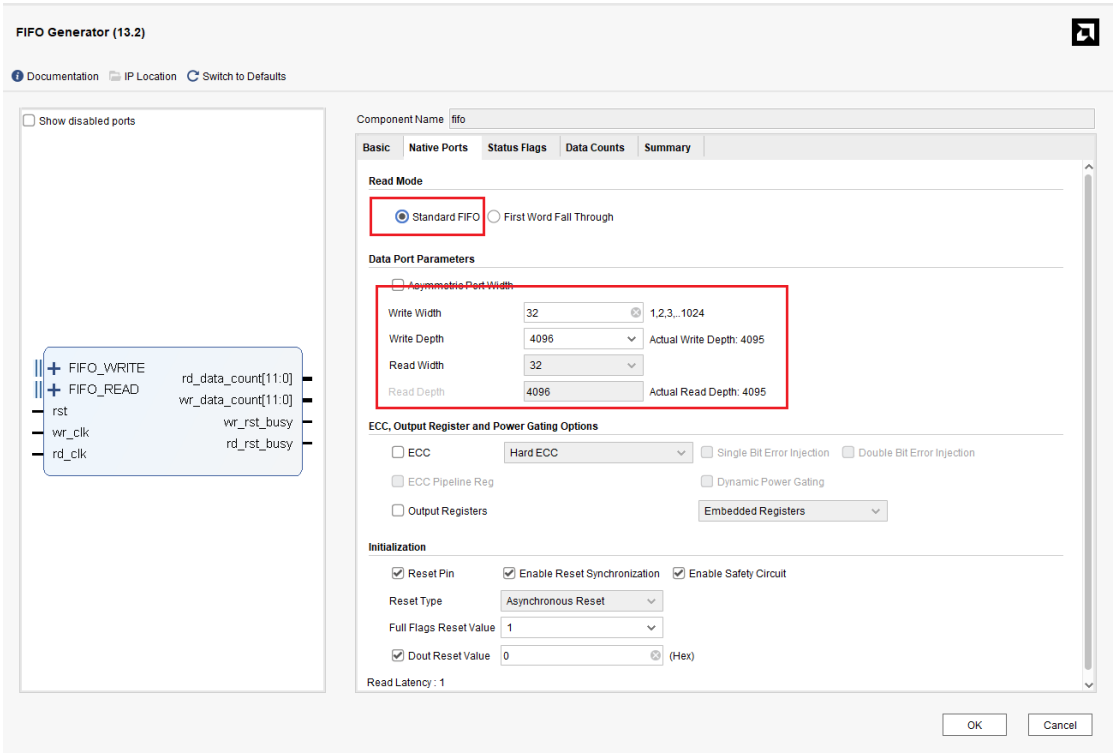

图 1-27 fifo 配置 2

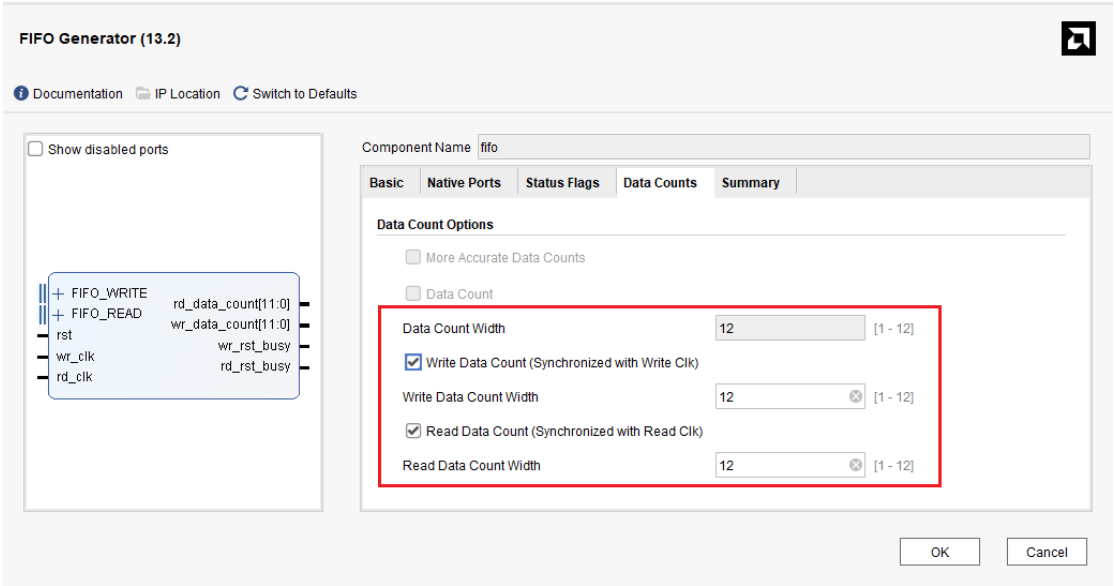

图 1-28 fifo 配置 3

又由于 fifo 是包含于 fifo2axis_write 中被添加进 BD，所以在生成 FIFO 输出文件时，需要将综合选项设置为 Global，如图 1-29 所示：

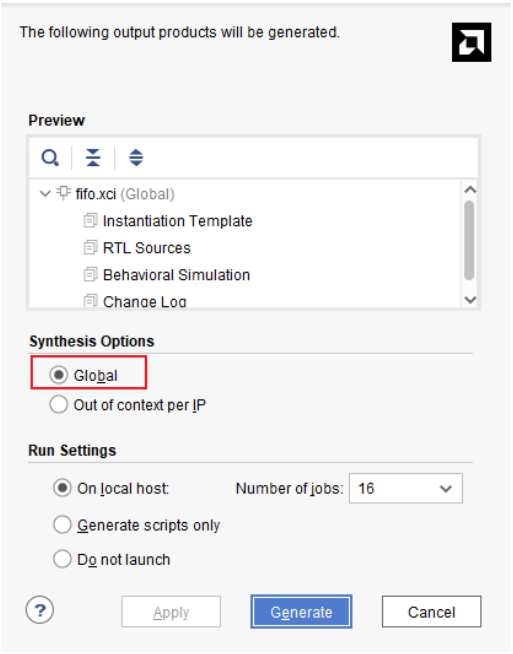

图 1-29 生成 FIFO 输出文件

至于 adc_sample_ctrl，则是基于 AXI4-Lite 模板创建的，用于控制 ADC 采样的自定义 IP。PS 端在接收并解析用户指令后，可以通过 AXI4-lite 接口，配置 IP 内部寄存器，输出用于控制 ADC 采样的相关驱动信号。部分代码如下：

```
// Add user logic here
assign ad_sample_en = slv_reg0[0];
assign Go = slv_reg0[8];
assign RestartReq = slv_reg0[16];
assign ChannelSel = slv_reg1[7:0];
assign ADC_Speed_Set = slv_reg2;
assign ByteToTrans = slv_reg3[25:0];
assign Data_Sum = slv_reg4;
assign ADC_Mode = slv_reg5;
assign Data_Num = slv_reg6;//每个通道需要采样的数据个数
```


完整代码用户可以自行查看，这里不作讲解。所有 IP 添加完成后，接下来对 IP 进行配置。

（1）ZYNQ 核

首先是 ZYNQ 核，对于设计而言，我们需要使能 PS 以太网控制器来驱动 PL 以太网口，所以这里我们使能任意一个 ETH 控制器外设即可。需要注意的是，其引脚路由方式需要设置为 EMIO。为了方便调试，这里我们还需要使能 PS 串口，用于后续的信息打印，同时注意设置 Bank 的电压。因此，I/O 配置如图 1-30 所示：

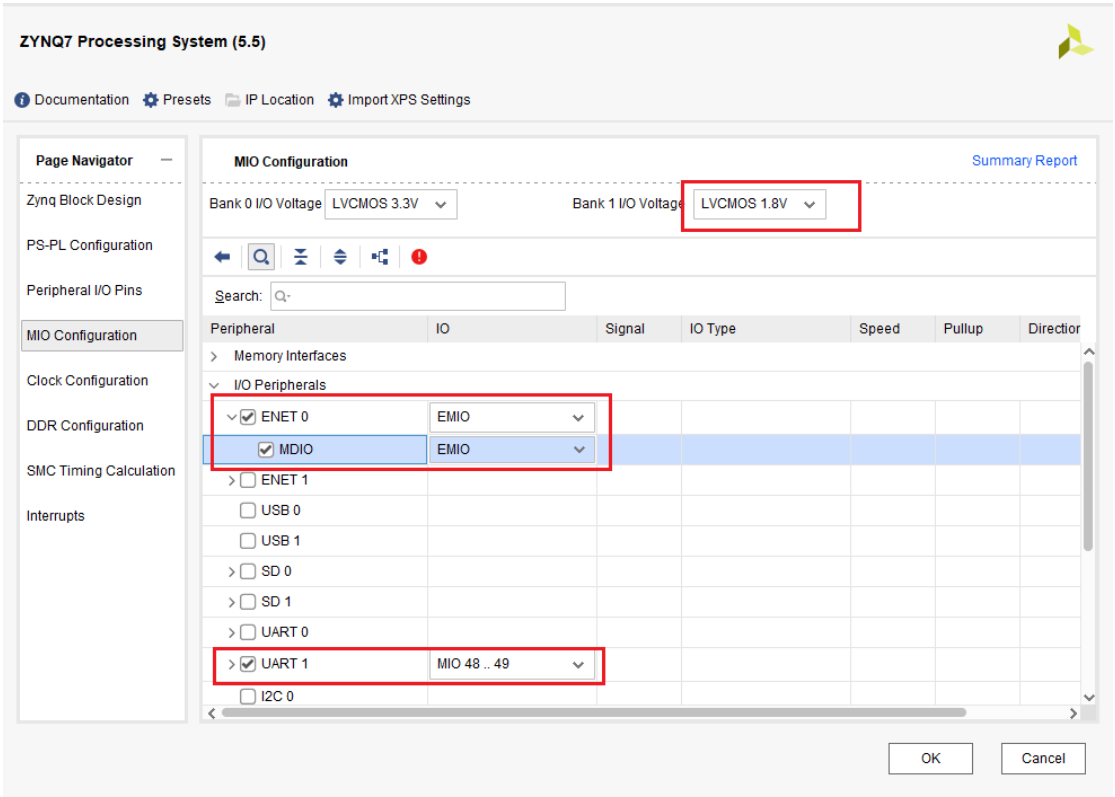

图 1-30 使能外设

这里如果读者使用的是小梅哥其他系列开发板，请根据实际的 MIO 管脚编号进行配置。

然后是 DDR 控制器的配置，如图 1-31 所示：

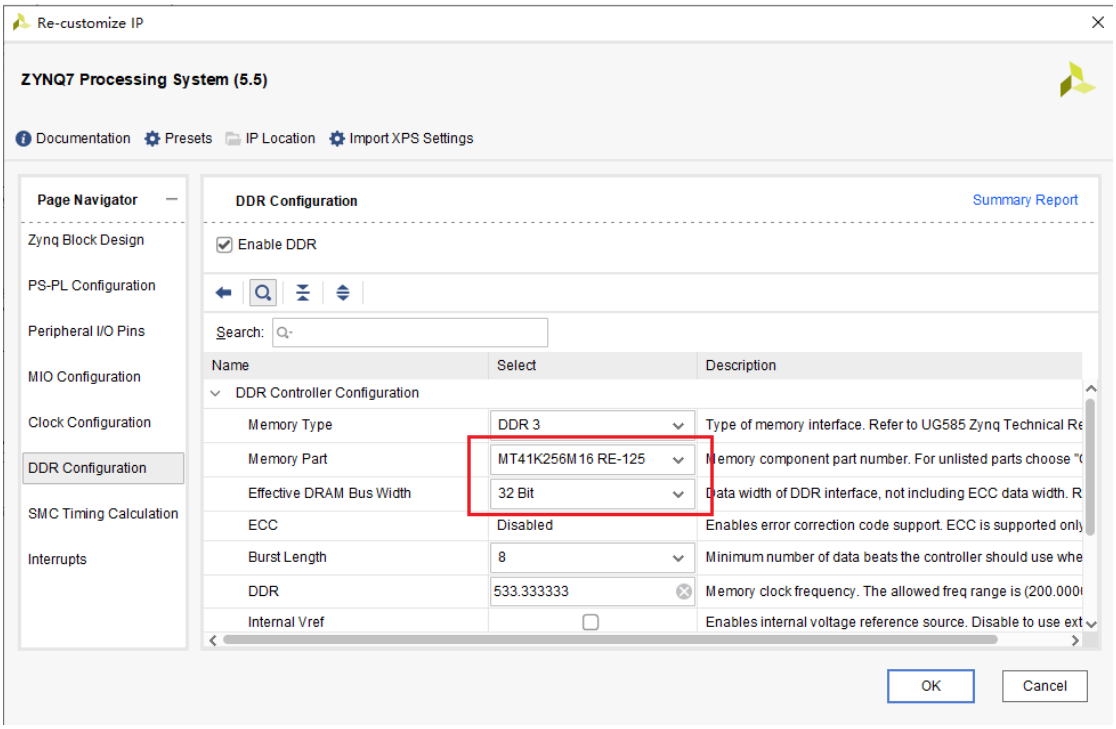

图 1-31 DDR 控制器配置

这里读者在配置时，需根据自身所使用开发板的实际 DDR 型号以及位宽来进行配置，若不清楚相关配置，可参考对应开发板例程中的配置方案。

接着是时钟配置，这里我们需要输出两路时钟，一路为 100MHz，主要作为 AXI 总线的工作时钟，另一路为 200MHz，用来作为 GMII to RGMII 的工作时钟。具体配置如图 1-32 所示：

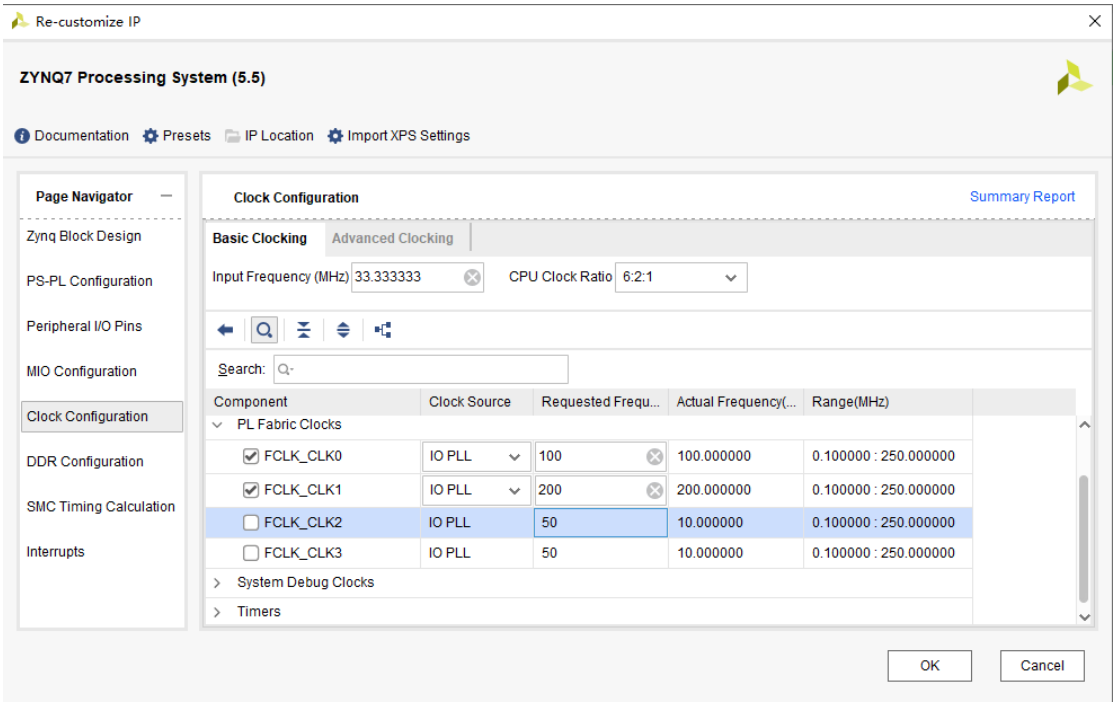

图 1-32 输出时钟配置

最后，是 AXI 接口以及中断接口使能。这里我们需要使能一个 GP 主接口，用于 PS 读写控制相关 IP；使能一个 HP 从接口，用于 DMA 搬运数据。另外，使能 PL-PS 中断，用于连接 DMA 中断。相关配置如下图：

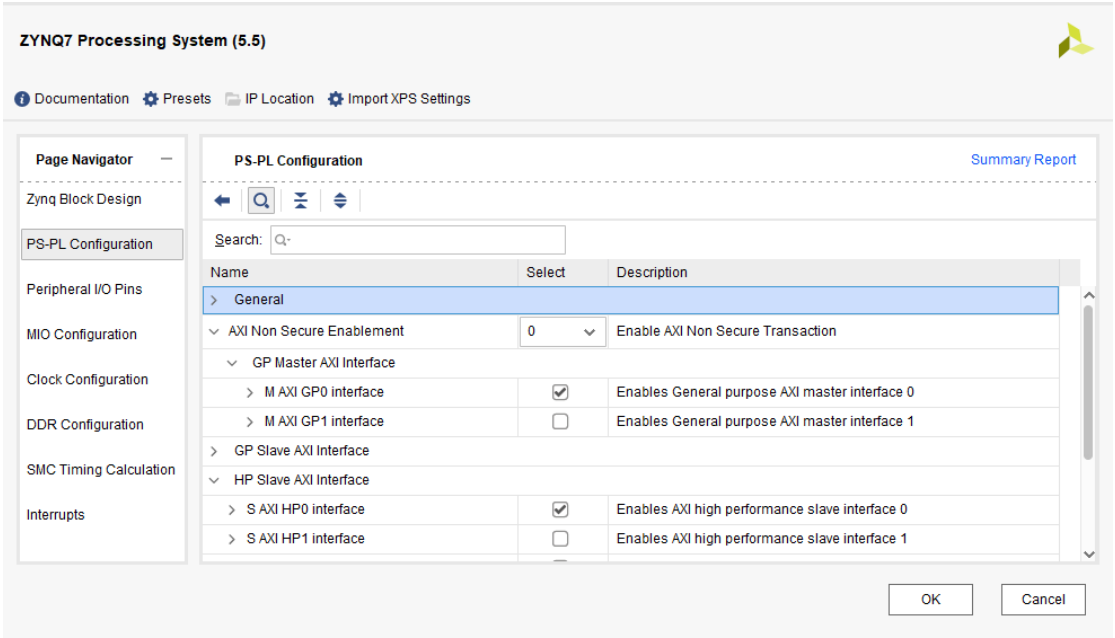

图 1-33 使能 AXI 接口

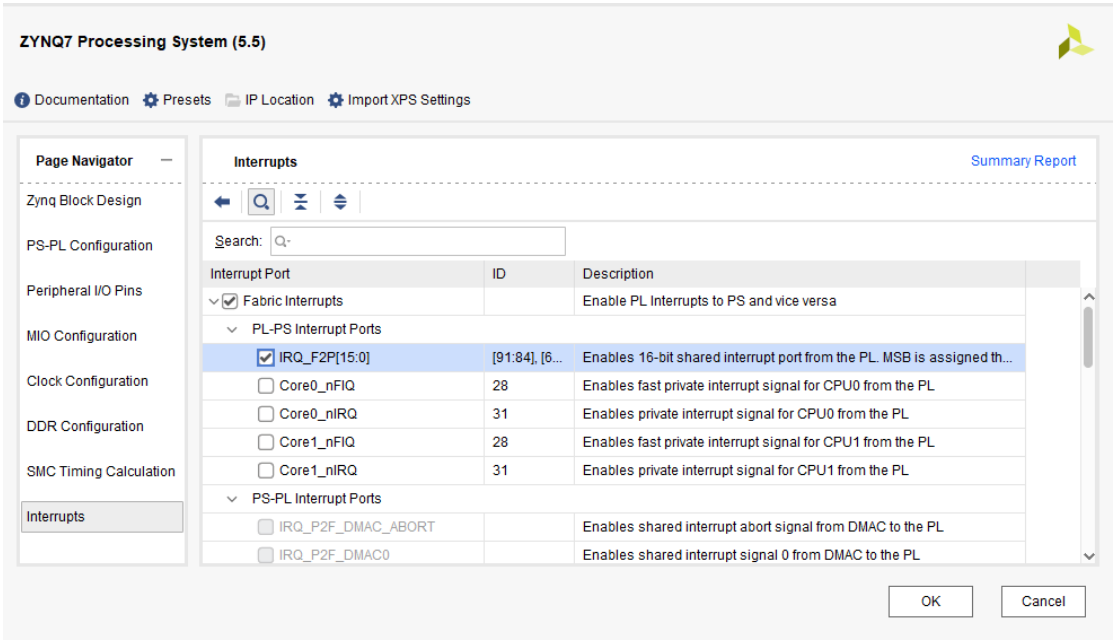

图 1-34 使能 PL-PS 中断

至此，便完成了 ZYNQ 核的配置，此时 IP 如图 1-35 所示：

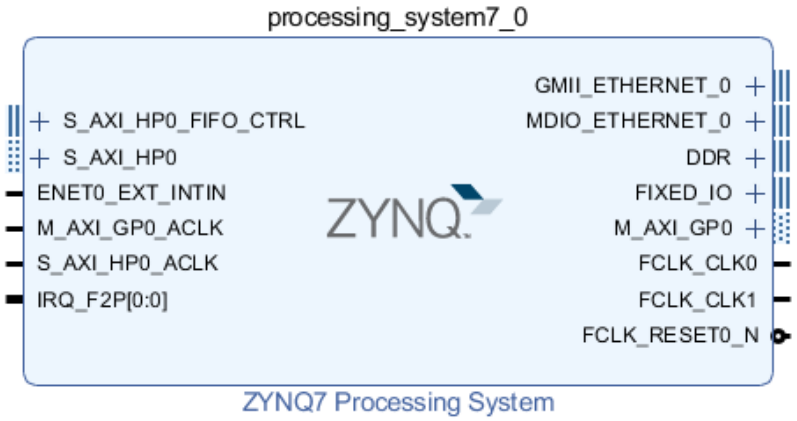

图 1-35 配置完成 ZYNQ 核

（2）DMA 核

本次设计中 DMA 核用于将 ADC 采样后的数据搬运进 DDR 中，因此只需要使能写通道即可，对应配置如图 1-36 下：

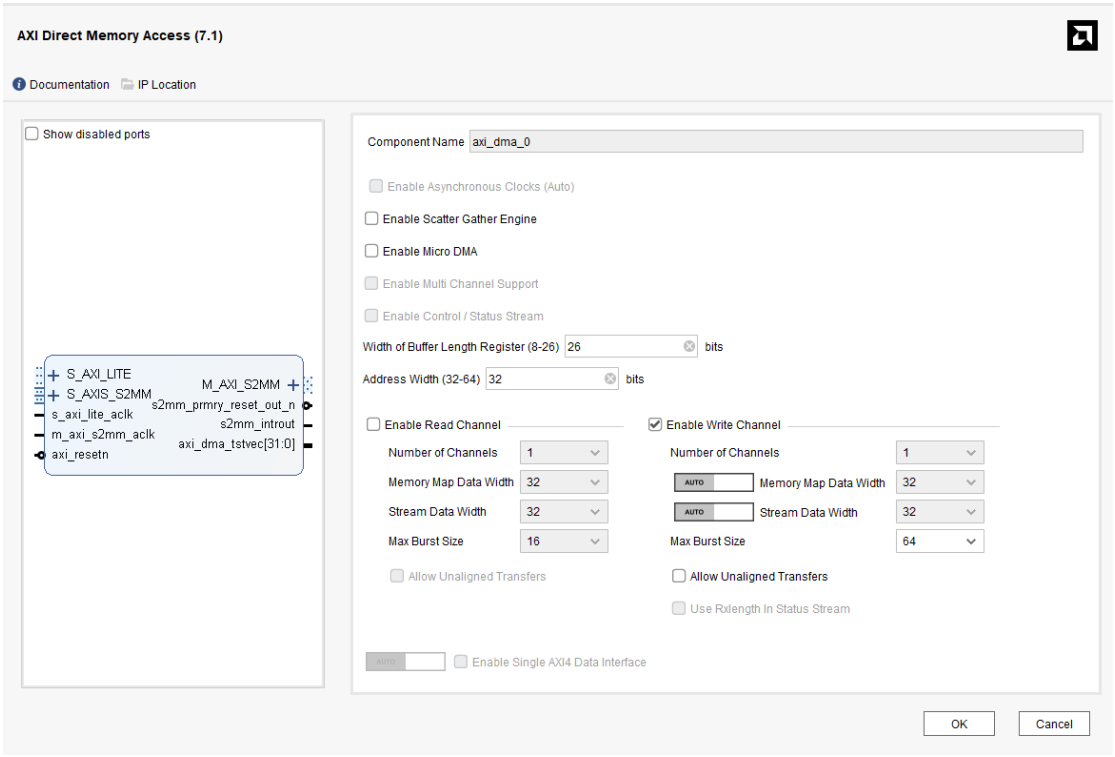

图 1-36 DMA 配置

需要注意的是，这里 DMA 的 Buffer Length Register 位宽设为 26，单次传输最多可指定 2^26−1 字节，略小于 64 MiB。本例程序以 0x03FF0000 字节为单段传输上限，超过该长度的采集请求需要分段搬运，每段传输完成后产生一次完成中断。

配置完成后的 DMA 核如图 1-37 所示：

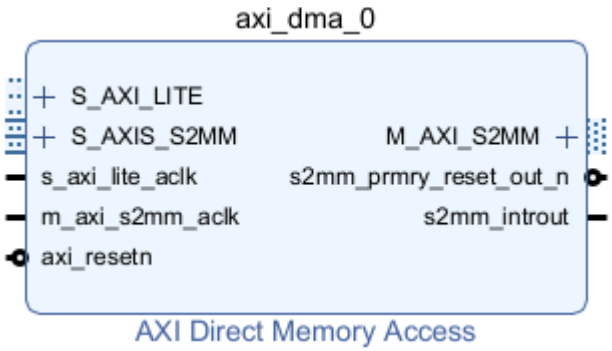

图 1-37 配置完成后 DMA 核

（3）GMII to RGMII

由于 PS 侧的 ETH 控制器面向 PL 侧，提供的是 GMII 接口，所以便需要 GMII to RGMII 核来进行接口转换。该 IP 的配置如下：

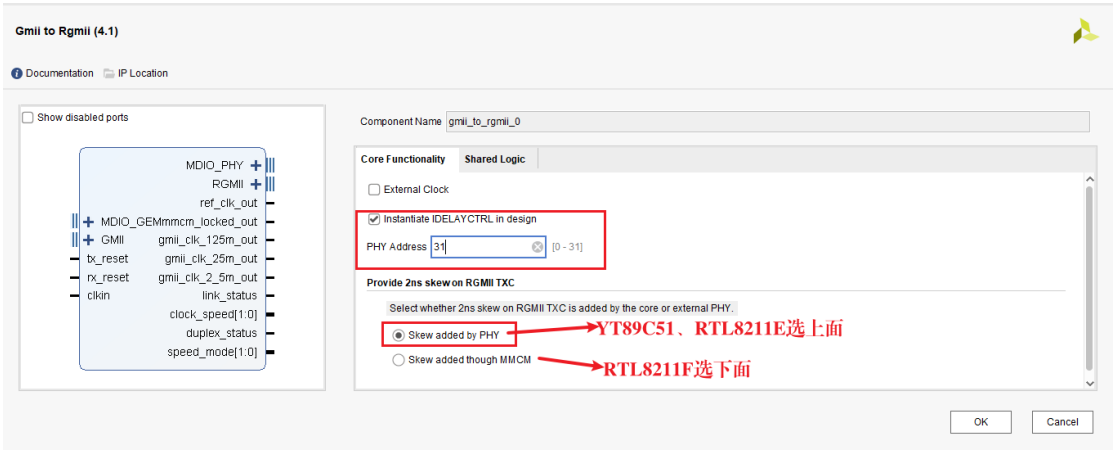

图 1-38 配置 GMII to RGMII

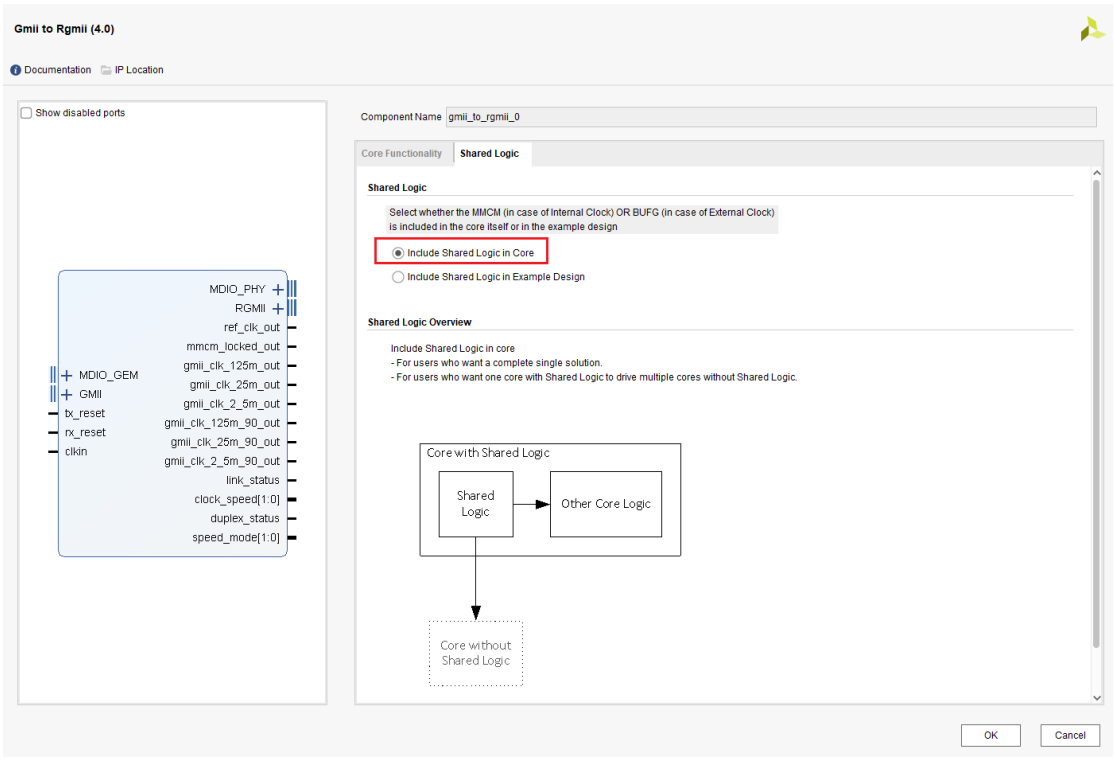

图 1-39 配置 GMII to RGMII

这里讲一下涉及到的几项配置：

- PHY Address，该项不是以太网 PHY 地址，而是用于标识 MDIO 事务中 core 的虚拟地址。因此该项只需设置一个与板载 PHY 不同的地址即可。
- Provide 2 ns Skew on RGMII TXC：选择为 RGMII 的 TXC 添加 2ns 的偏斜，用户可以选择由外部 PHY 添加，或由 IP 核通过 MMCM 添加。
- Shared Logic：选择是否需要共享时钟资源。当设计中只有一个 core，或者存在多个 core 但需要一个 core 共享时钟资源，以驱动其余核时，需要勾选 Include Shared Logic in the Core。而只有当设计中存在多个 core，且已经有一个 core 共享了时钟资源用来驱动其他核时，才需要勾选 Include Shared Logic in the Example Design。

这里的 Provide 2 ns Skew on RGMII TXC 配置项，需要根据实际的以太网 PHY 芯片型号决定，用户如果不清楚当前开发板的以太网 PHY 型号，可以通过原理图或芯片上的丝印进行区分。

配置完成后的 GMII to RGMII 核如图 1-40 所示：

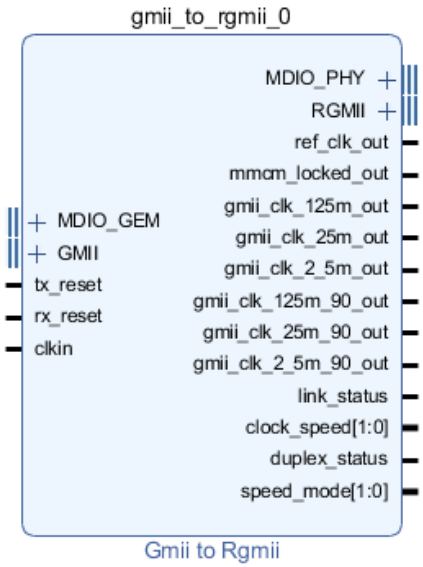

图 1-40 配置完成后的 GMII to RGMII 核

（4）Constant

Constant 模块用于输出常量，在本次设计中，该模块用于驱动 ETH PHY 芯片的复位信号，通过输出高电平，确保以太网 PHY 芯片不处于复位状态。IP 核的配置如下：

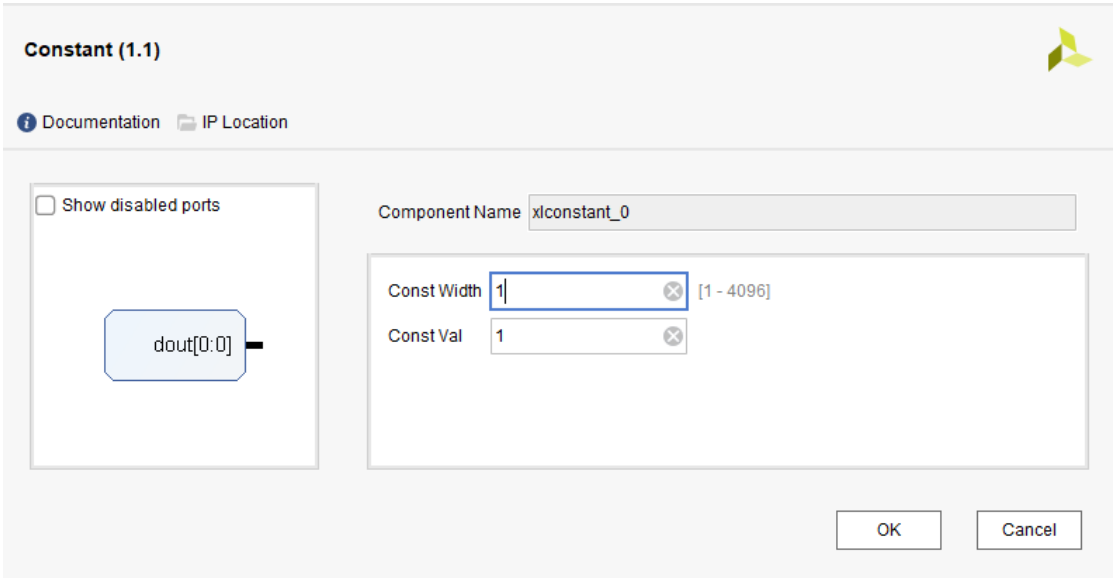

图 1-41 Constant 配置

至此，所有 IP 便配置完成，接下来开始端口连接。

1.2.1.2端口连接

依次点击 Run Block Automation 和 Run Connection Automation，让软件帮我们自动连线。自动连线时，注意修改连接到 gmii to rgmii 的时钟，如图 1-42 所示：

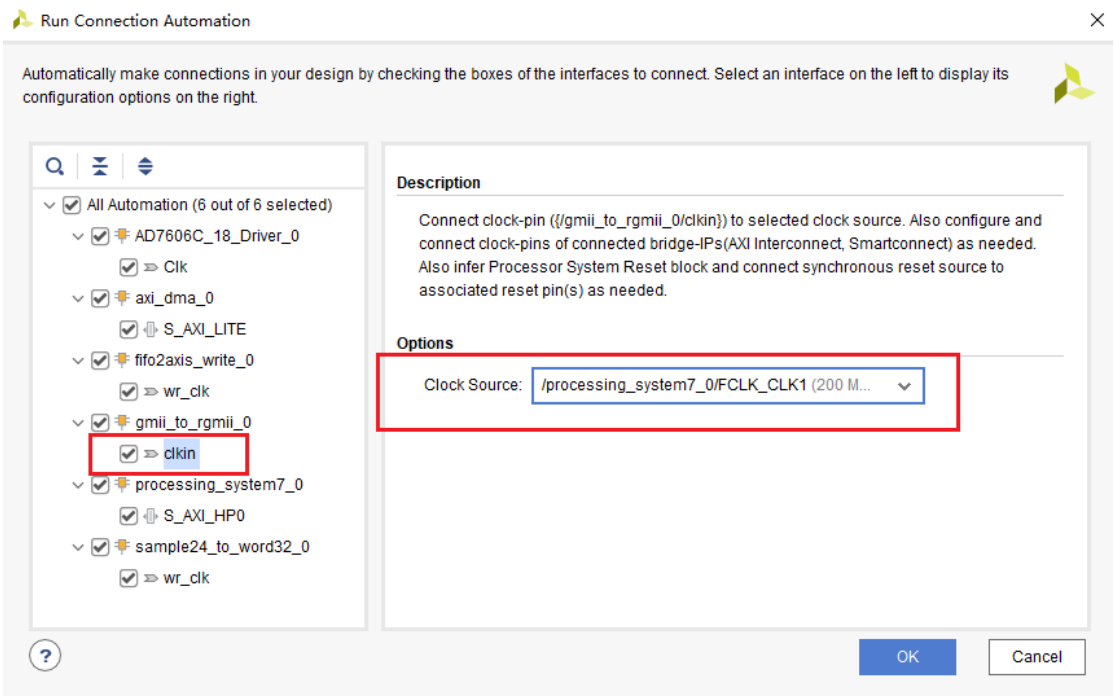

图 1-42 运行自动化连接

软件会帮我们插入互联 IP 以及复位管理模块，并连接部分接口信号。这里我们先手动补充时钟、复位信号的连接，如图 1-43 所示：

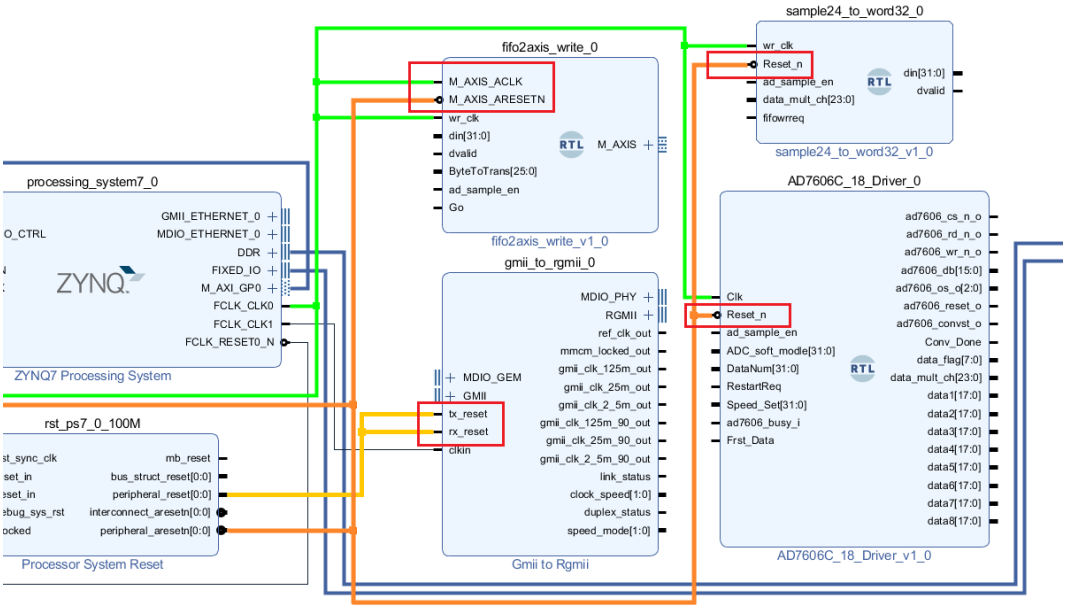

图 1-43 补接时钟和复位信号

接着，连接以太网相关信号。这里只需连接 GMII 与 MDIO 相接口，然后导出 RGMII、MDIO_PHY 接口以及以太网复位信号即可。如图 1-44 所示：

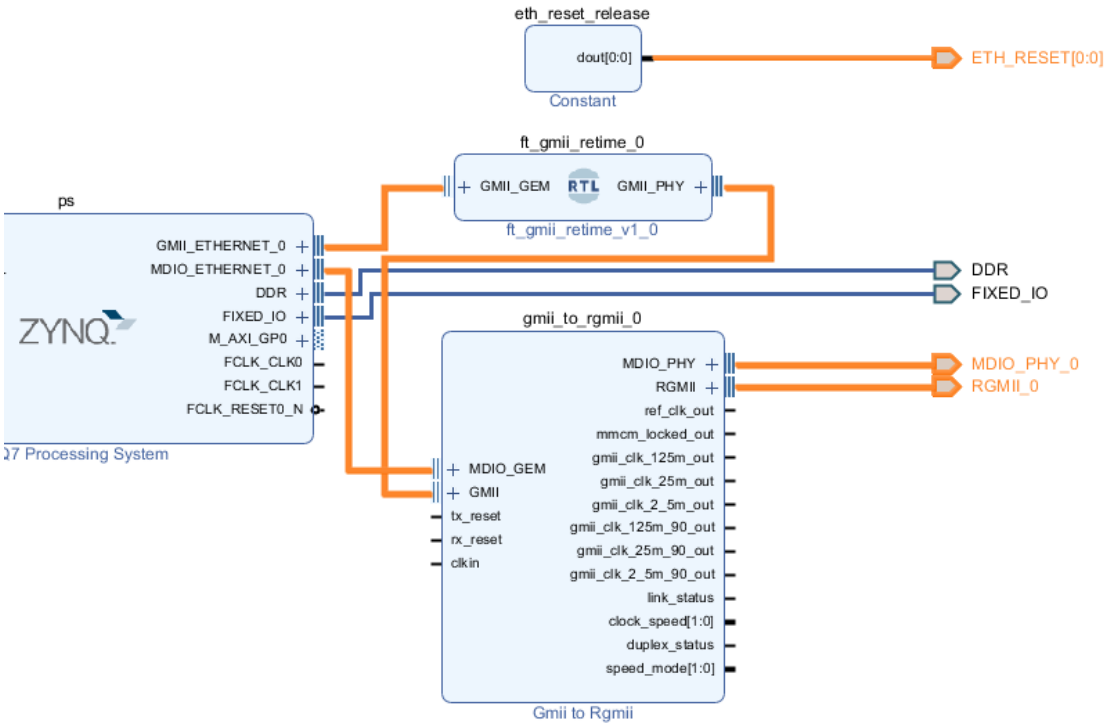

图 1-44 连接以太网相关信号

接下来按照功能依次连接 ADC 相关信号，首先是 ADC 采样控制指令。根据信号名，为 adc_sample_ctrl 连接相关接口信号，如下：

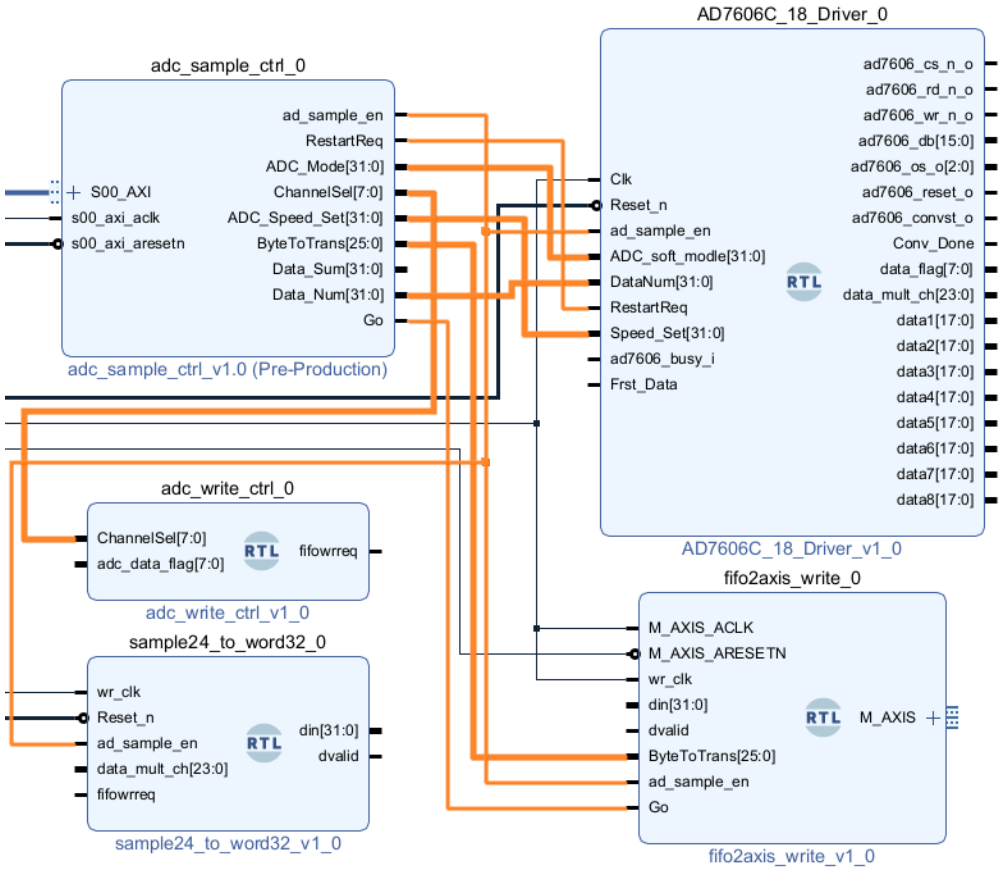

图 1-45 连接 ADC 采样驱动信号

接着，导出 ADC 驱动相关信号，如图 1-46 所示：

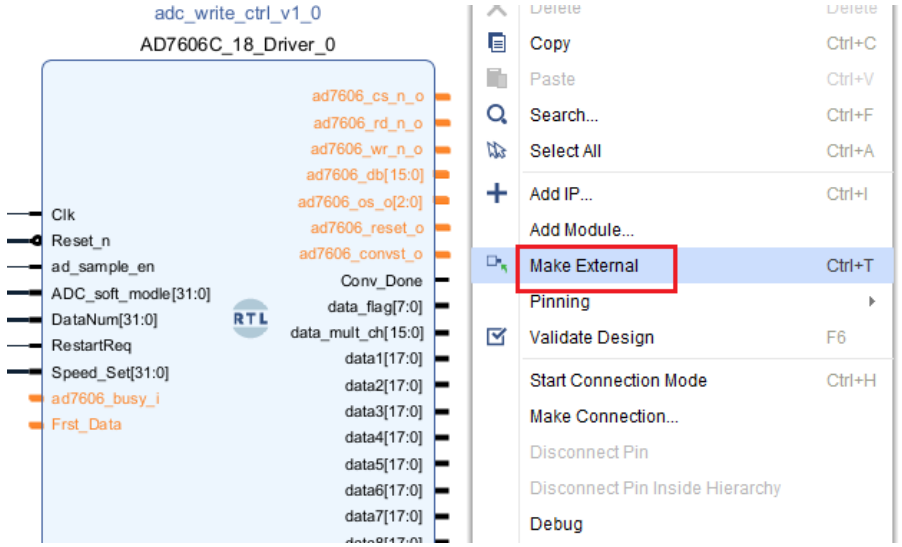

图 1-46 导出 ADC 驱动信号

然后，根据 ADC 数据流向，搭建信号链路，如图 1-47 所示：

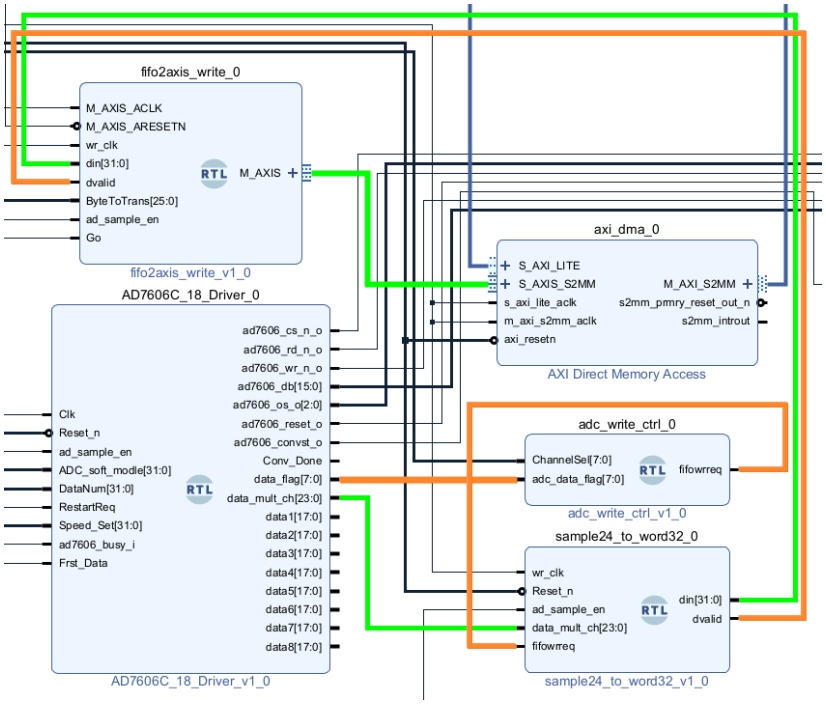

图 1-47 搭建 ADC 数据链路

最后，连接 DMA 中断，如此便完成了硬件逻辑系统的搭建，如图 1-48 所示：

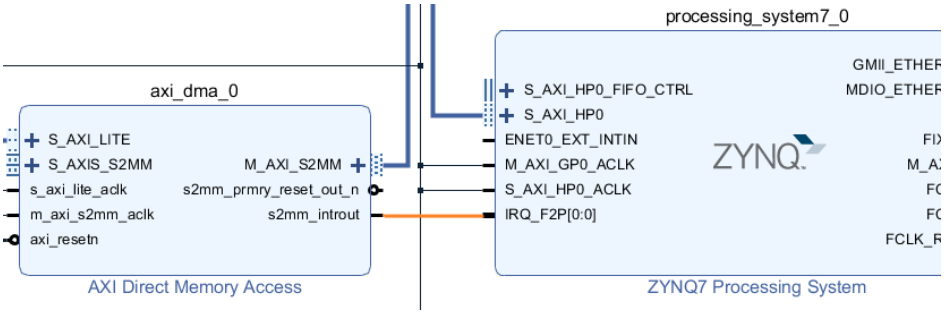

图 1-48 DMA 中断连接

此时，完整的硬件逻辑系统如下图 1-49：

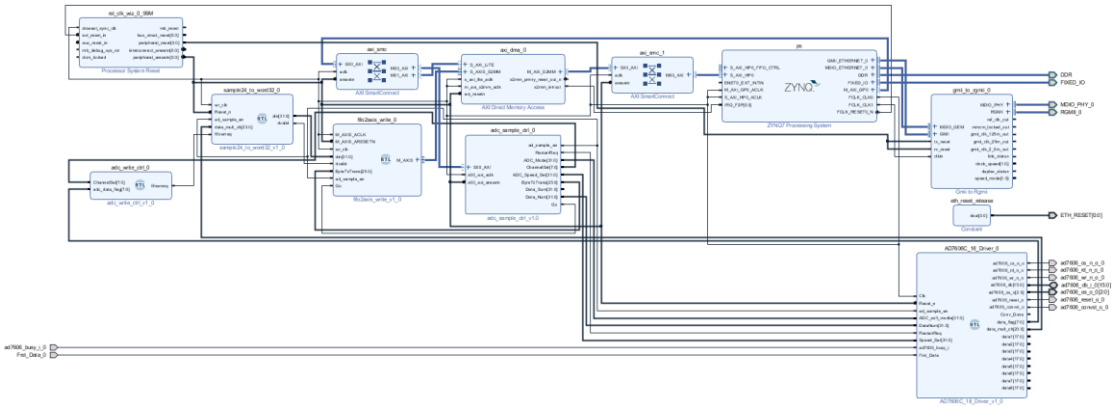

图 1-49 硬件逻辑系统

上图如果不清晰，用户可直接参看例程。接下来，我们分为三个部分来分析硬件逻辑系统。

1.2.1.3硬件逻辑系统分析

1.2.1.3.1 ADC 采样控制

ADC 的采样控制通过 GP 接口和 adc_sample_ctrl 实现，如图 1-50 所示：

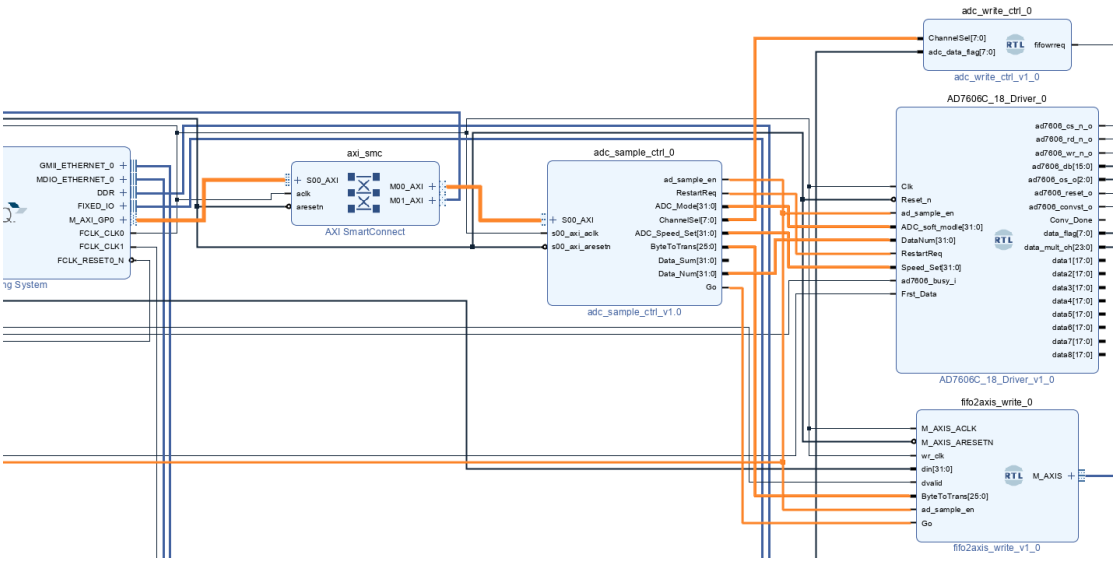

图 1-50 ADC 采样模式控制

PS 端在获取到用户通过以太下发的指令后，会对指令进行拆分处理，并通过 GP 接口，驱动 adc_sample_ctrl 产生相关控制信号，实现对 ADC 的采样控制。

adc_sample_ctrl 的部分端口功能如下：

表 1-9 adc_sample_ctrl 部分端口功能

端口	位宽	功能
ad_sample_en	1	ADC 采样使能信号
RestartReq	1	ADC 重新采样脉冲信号
ChannelSel	8	采样通道控制，bit[7:0]依次对应 CH8~CH1，置 1 使能

ADC_Speed_Set	32	采样频率控制字，控制 ADC 采样频率
ByteToTrans	26	当前 DMA 传输所需的字节数
ADC_Mode	32	ADC 软件模式控制字，参考图 1-23 中的模式选择
Data_Num	32	ADC 采样轮次
Go	1	DMA 传输使能

前面配置 DMA 核时我们讲过，26 位长度字段使单次 DMA 传输最多可指定 2^26−1 字节。因此，当采集数据量较大时，程序需要将其拆分为多段 DMA 传输。为了保持段间采样的连续性，设计对 ADC 采样和 DMA 搬运分开控制：使用 Go 和 ByteToTrans 控制单段 DMA 数据流，使用 ad_sample_en 和 Data_Num 控制整次 ADC 采样。

当采样使能有效的时候，adc_sample_ctrl 模块会根据配置信号，获取对应通道的数据并进行采样，最终的数据会被存入 fifo2axis_write 的 FIFO 中。当 FIFO 中首次缓存了至少 256 个 32 位字，也就是 1024 字节后，模块会将 en_send 信号置 1，随后，fifo2axis_write 模块便会通过 AXI-Stream（M_AXIS）接口将数据交由 DMA 搬运。根据用户采样数量的多少，数据可能会被拆分为多段 DMA 传输，只有满足 Data_Num 轮次的数据被 DMA 搬运后，传输才算完成。

为了避免下一次采集读到上一批残留的数据，fifo2axis_write 会在整次采集结束、ad_sample_en 拉低后复位 FIFO。若一次采集需要分成多段 DMA 传输，段间仅切换 Go，ad_sample_en 保持有效，因此不会触发整批复位，FIFO 中的后续数据可以继续搬运。

```
case (mst_exec_state)
    IDLE:
        mst_exec_state  <= INIT_COUNTER;

    INIT_COUNTER:
    if ( (count == C_M_START_COUNT - 1) && en_send )
        mst_exec_state  <= SEND_STREAM; // 计数够时间后跳转到下一个状态 SEND_STREAM
    else
        begin
        count <= count + 1;
        mst_exec_state  <= INIT_COUNTER;
        end

    SEND_STREAM:
    if (tx_done)begin
        mst_exec_state <= IDLE;    //发送完成后回到 IDLE 状态
    end
    else
        mst_exec_state <= SEND_STREAM;
endcase

assign axis_tvalid = ((mst_exec_state == SEND_STREAM) && (read_pointer <
ByteToTrans[25:2]) && (rd_data_count >=2) && en_send);
assign axis_tlast = axis_tvalid && (read_pointer == ByteToTrans[25:2]-1);
```

```
case(state)
    ......
    2:if(Go)begin
        if(rd_data_count >= 256)
            state <= 3;
        end

    3: state <= 4;

    4:begin
        en_send <= 1;
        state <= 5;
        end

    5:
    if(tx_done)begin
        en_send <= 0;
end
else if(!ad_sample_en)begin
    state <= 0;
end
    else if(!Go)begin
        state <= 2;
    end
endcase
```

1.2.1.3.2 ADC 软件模式驱动设计

软件模式的驱动通过 AD7606C_18_Driver 模块实现，通过将 OS[2:0]设置为 3’b111，AD7606C-18 便可以进入到软件模式，因此模块首先会对 OS 进行设置。

```
assign ad7606_os_o = 3'b111;
```

软件模式下，ADC 又分为 ADC 模式和寄存器模式，前者主要负责驱动 ADC 进行数据采集，后者则是负责寄存器的读写配置。结合前面 1.1.4 小节中对于两种模式下的时序要求，这里模块采用了状态+序列机的方式，将功能拆分为了表 1-10 中的 11 个状态。

表 1-10 状态机说明

状态名	编码	功能说明
IDLE	12'b0000_0000_0001	上电初始空闲态
Full_Rst	12'b0000_0000_0010	AD7606C 硬件完整复位时序
Part_Rst	12'b0000_0000_0100	AD7606C 硬件部分复位时序
Mode_Sel	12'b0000_0000_1000	模式判断分支（复位 / 寄存器配置 / 采集）
Config_Mode	12'b0000_0001_0000	寄存器初始化配置入口（发送读指令进入寄存器模式）
Reg_Mode	12'b0000_0010_0000	单寄存器配置（写地址→读→写数据）
ADC_Mode	12'b0000_0100_0000	8 通道 AD 循环采集（内含子状态机）
WR_Reg	12'b0000_1000_0000	通用寄存器写时序子状态

RD_Command	12'b0001_0000_0000	通用寄存器读时序子状态
Lut_Config	12'b0010_0000_0000	寄存器初始化配置
Empty_Cmd	12'b0100_0000_0000	保持一个写周期的低电平，退出寄存器模式

首先是空闲态，空闲态主要是完成状态的跳转，代码如下：

```
IDLE: begin
    state <= Full_Rst;
end
```

然后是复位态，这里状态实现的是完全复位，代码如下：

```
Full_Rst:begin      //大复位
    case(seq_cnt)
        00:begin
            cnt_high <= cnt_high + 1'b1;
            ad7606_reset_o <= 1;
            if(cnt_high >= full_reset)begin
                seq_cnt <= seq_cnt +1'b1;
                cnt_high <=0;
            end
        end

        01:begin
            cnt_low <= cnt_low + 1'b1;
            ad7606_reset_o <= 0;
            if (cnt_low >= full_reset_setup)begin
                seq_cnt <= 0;
                cnt_low <=0;
                state <= Mode_Sel;
            end
        end
    endcase
end
```

根据图 1-11 中，对于完全复位的时序参数可知，复位高脉冲至少应持续 3200ns，复位结束 274us 后，ADC 才能开始工作。因此，这里对于复位参数的定义如下：

```
parameter full_reset_setup = 27400; // >= 274000ns
parameter full_reset  = 340;   // >= 3200ns
```

全复位后，ADC 回到默认的 ±10 V 量程。驱动在复位等待结束后进入 Mode_Sel。此时需要用户手动发送指令 4、参数 0x00000001 。驱动会进入 Config_Mode，并通过初始化表将 0x03～0x06 写为 0x11，把八路量程设为 ±5 V。

Mode_Sel 状态中，状态机会等待 RestartReq 触发，并根据锁存的 ADC_soft_modle 跳转到全复位、初始化、单寄存器配置或采样状态。

```
Mode_Sel:begin
    case(seq_cnt)
        00:begin
            cnt_DataNum <= 32'd0;
```

```
            ad7606_cs_n_o <= 1'b1;
            data_mult_ch <= 0;
            if (RestartReq) begin
                ADC_soft_modle_r <= ADC_soft_modle;
                seq_cnt <= seq_cnt + 1'b1;
            end
        end

        01:begin
            if(ADC_soft_modle_r[15:0] == 16'd2)begin
                state <= Full_Rst;
                seq_cnt <= 0;
            end else if(ADC_soft_modle_r[15:0] == 16'd1) begin /// 进入软件模式，初始化寄存器表
                seq_cnt <= 0;
                Wr_Done <= 0;
                Rd_Done <= 0;
                state <= Config_Mode;
            end else if (ADC_soft_modle_r != 32'd0) begin // 进入软件模式，修改单个寄存器
                seq_cnt <= 0;
                state <= Reg_Mode;
            end else begin
                Data_En <= 0;
                seq_cnt <= 0;
                state <= ADC_Mode;
            end
        end
    endcase
end
```

其中， ADC_soft_modle 为模式选择指令，该指令的具体位定义，可以参考图 1-23。状态机会对该指令打拍缓存，防止状态机运行过程中，因为指令发生了更改，而导致状态机运行出错。

当 ADC_soft_modle_r[15:0]的值为 1 时，代表用户希望对 ADC 进行初始化配置，此时，状态会跳转到 Config_Mode，该状态代码如下：

```
Config_Mode:begin
    Mode_Record <= 3'b001;
    case(seq_cnt)
        00:begin        //读寄存器指令
            if(Wr_Done)begin
                seq_cnt <= seq_cnt + 1'b1;
                Wr_Done <= 0;
            end
            else begin
                data_o <= {1'b1,7'h2f,8'h00};  // 将 DB15 设置为 1 进入寄存器模式。
                state <= WR_Reg;
```

```
            end
        end

        01:begin //读第一帧
            if(Rd_Done)begin
                seq_cnt <= 0;
                Rd_Done <= 0;
                state <= Lut_Config;//读完成，进入寄存器模式，准备通过 lut 配置寄存器
            end else begin
                    state <= RD_Command;
            end
        end
    endcase
end
```

Config_Mode 只是配置模式的入口，负责让 ADC 从 ADC 模式进入到寄存器模式。进入寄存器模式的时序可参考图 1-12，整体上分为两步：

（1） 将 DB[15]设置为 1，DB[14:8]设置待读取的寄存器，通过$\overline{WR}$信号，发送读指令

（2） 释放 DB 总线，通过$\overline{RD}$信号，读取寄存器数据

状态中通过序列 seq_cnt 的值，来实现这两步。读指令的发送，通过 WR_Reg 状态实现，代码如下：

```
WR_Reg:begin
    case(data_seq_cnt)
        00:begin
            ad7606_cs_n_o <= 0; // 提供需要读取的寄存器地址
            wr_rd_flag <= 1;
            data_seq_cnt <= data_seq_cnt + 1'b1;
        end

        01:begin
            cnt_wr <= cnt_wr + 1'b1;
            ad7606_wr_n_o <= 0; // 拉低写使能，准备好数据，
            if(cnt_wr >= LOW_WR) begin
                cnt_wr <= 0;
                ad7606_cs_n_o <= 1;
                ad7606_wr_n_o <= 1; // 在上升沿锁存到芯片
                data_seq_cnt <= data_seq_cnt + 1'b1;
            end
        end

        02:begin
            cnt_wr <= cnt_wr + 1'b1;
            if (cnt_wr >= HIGH_WR) begin
                cnt_wr <= 0;
                Wr_Done <= 1'b1;
                data_seq_cnt <= 0;
                if(Mode_Record == 3'b001)
```

```
                    state <= Config_Mode;
                else if(Mode_Record == 3'b000)
                    state <= Reg_Mode;
                else if(Mode_Record == 3'b011)
                    state <= Empty_Cmd;
                else if(Mode_Record == 3'b100)
                    state <= Lut_Config;
            end else begin
                state <= state;
            end
        end
    endcase
end
```

WR_Reg 状态主要通过控制$\overline{CS}$和$\overline{WR}$信号，完成寄存器数据的写入，该部分时序可以参考图 1-12。为了方便状态的回跳，状态机在相关状态中还会通过 Mode_Record 对状态进行记录，这样就能根据 Mode_Record 的值进行跳转。Mode_Record 与状态的对应关系如下表 1-11 所示：

表 1-11 Mode_Record 与状态对应关系

Mode_Record	状态
3'b000	Reg_Mode
3'b001	Config_Mode
3'b010	ADC_Mode
3'b011	Empty_Cmd
3'b100	Lut_Config

寄存器数据的读取则是通过 RD_Command 状态实现，代码如下：

```
RD_Command:begin
    case(data_seq_cnt)
        00: begin ad7606_cs_n_o <= 0; data_seq_cnt <= data_seq_cnt +
1'b1;wr_rd_flag <= 0; end
        01: begin ad7606_rd_n_o <= 0; data_seq_cnt <= data_seq_cnt + 1'b1; end //
开始读取的寄存器的数据  拉低 3 个，拉高一个
        02: begin ad7606_rd_n_o <= 0; data_seq_cnt <= data_seq_cnt + 1'b1; end
        03: begin ad7606_rd_n_o <= 0; data_seq_cnt <= data_seq_cnt + 1'b1; data_i<=
ad7606_db_i; data_mult_ch<= {16'd0,ad7606_db_i};end
        04: begin
            ad7606_rd_n_o <= 1;
            data_seq_cnt <= 0;
            wr_rd_flag <= 1;
            ad7606_cs_n_o <= 1;
            Rd_Done <= 1'b1;
            if(Mode_Record == 3'b001)
                state <= Config_Mode;
            else if(Mode_Record == 3'b000)
                state <= Reg_Mode;
```

```
            else if(Mode_Record == 3'b010)
                state <= ADC_Mode;
            else if(Mode_Record == 3'b100)
                state <= Lut_Config;
        end
    endcase
end
```

该状态通过控制$\overline{CS}$和$\overline{RD}$信号，读取寄存器数据。其中 wr_rd_flag 为读写标志信号，用来实现 DB 总线的三态控制。

```
assign ad7606_db_t = ~wr_rd_flag ? 16'hffff : 16'h0000;
```

DB 总线会在 data_seq_cnt 序列为 0 时释放，交由 ADC 控制。随着$\overline{CS}$和$\overline{RD}$信号都被拉低，ADC 会将数据更新到总线上，待数据稳定后，即$\overline{RD}$上升沿，状态机会读取 DB 总线上的值，从而完成对寄存器的读取。随后，根据 Mode_Record，再次回到 Config_Mode 状态。

此时，由于已经完成了寄存器的读取，ADC 成功进入寄存器模式，状态机便会跳转到 Lut_Config 状态，该状态代码如下：

```
Lut_Config: begin
    Mode_Record <= 3'b100;
    case(seq_cnt)
        00:begin
            if (Wr_Done) begin
                Wr_Done <= 0;
                seq_cnt <= seq_cnt +1'b1;
            end
            else begin
                data_o <= {1'b0, lut[14:0]};
                state <= WR_Reg;
            end
        end

        01:begin
            seq_cnt <= 0;
            if (cnt_table >= 6'd39) begin
                cnt_table <= 0;
                state <= Empty_Cmd;
            end
            else begin
                cnt_table <= cnt_table +1'b1;
            end
        end
        endcase
end
```

这里主要是根据预先写好的 AD7606C-18 初始化配置表 lut，通过 WR_Reg 模式，进行寄存器批量配置。状态通过 cnt_table 对已配置的寄存器数量进行计数，当初始化配置完成后，状态会跳转到 Empty_Cmd 态，该状态代码如下：

```
Empty_Cmd: begin // 保持一个 WR 周期低电平，退出寄存器模式，回归等待状态
    Mode_Record <= 3'b011;
    case (seq_cnt)
        6'd0: begin
            if (Wr_Done) begin
                Wr_Done <= 1'b0;
                state   <= Mode_Sel;
            end else begin
                data_o <= {1'b0, 7'h00, 8'h00};
                state  <= WR_Reg;
            end
        end
    endcase
end
```

该状态会让 DB 总线在一个写周期内保持低电平，从而让 ADC 退出寄存器模式，回到 ADC 模式。随后，状态机会回到 Mode_Sel 态，等待用户的下一次指令。

如果用户发送的指令 ADC_soft_modle 为除 0,1,2 外的其他值，那么便代表要进行单个寄存器的配置。此时，ADC_soft_modle[31:24]为待配置的寄存器地址，ADC_soft_modle[23:16]为待写入的数据。状态机会跳转到 Reg_Mode 态，进行寄存器配置。该状态代码如下：


```
Reg_Mode: begin // 配置并读写单个自定义寄存器
    Mode_Record <= 3'b000;
    case (seq_cnt)
        6'd0: begin
            if (Wr_Done) begin
                seq_cnt <= seq_cnt + 1'b1;
                Wr_Done <= 1'b0;
            end else begin
                data_o        <= {1'b1, ADC_soft_modle_r[30:16]};
                state         <= WR_Reg;
                ad7606_cs_n_o <= 1'b0;
            end
        end
        6'd1: begin
            if (Rd_Done) begin
                seq_cnt <= seq_cnt + 1'b1;
                Rd_Done <= 1'b0;
            end else begin
                state <= RD_Command;
            end
        end
        6'd2: begin
```

```
            if (Wr_Done) begin
                seq_cnt <= 6'd0;
                Wr_Done <= 1'b0;
                state   <= Empty_Cmd;
            end else begin
                data_o <= {1'b0, ADC_soft_modle_r[30:16]};
                state  <= WR_Reg;
            end
        end
    endcase
end
```

该状态会先读取待配置的寄存器，让 ADC 进入到寄存器模式，随后，再来配置寄存器。配置完成后，状态会跳转到 Empty_Cmd 态，在一个写周期内将 DB 拉低，从而退出寄存器模式，并再次回到 Mode_Sel 态等待下一次指令。

最后，如果用户发送的指令 ADC_soft_modle 为 0，便代表要进行 ADC 采样。此时，状态会跳转到 ADC_Mode。该状态主要是根据图 1-10 中并行接口的读时序要求，驱动 ADC 完成采样数据的读取。代码如下：

```
ADC_Mode:begin
    case(seq_cnt)
        00:begin
            wr_rd_flag <= 0;
            ad7606_cs_n_o <= 1'b1;
            if(ad_sample_en && trig)begin
                seq_cnt <= seq_cnt + 1'b1;
                ad7606_convst_o <= 0;
                ad7606_rd_n_o <= 1;
                ad7606_cs_n_o <= 1;
                Conv_Done <= 0;
                ad7606_reset_o <= 0;
            end
            else begin
                seq_cnt <= seq_cnt;
                ad7606_rd_n_o <= 1;
                ad7606_convst_o <= 1;
                ad7606_cs_n_o <= 1;
            end
        end

        01: begin seq_cnt <= seq_cnt + 1'b1; ad7606_convst_o <= 1;end
        02:if(ad7606_busy_r==2'b01)begin  /// 等待芯片转换完成，
                seq_cnt <= seq_cnt + 1'b1;
                ad7606_cs_n_o <= 0;
            end else  begin
                seq_cnt <= seq_cnt;
            end
        03:begin ad7606_rd_n_o <= 0;seq_cnt <= seq_cnt + 1'b1;end
```

```
////////////////////////////////////////////////////////////////////////////
        04: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        05: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        06: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;data1[17:2] <=
ad7606_db_i;data_mult_ch[17:2] <=ad7606_db_i;end


        07: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        08: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        09: begin ad7606_rd_n_o <= 0;  data1[1:0] <=
ad7606_db_i[15:14];data_mult_ch[1:0] <=ad7606_db_i[15:14];// 通道 1
            if(!Frst_Data) begin
                seq_cnt <= 04;
                Data_err <= 1'b1;
            end
            else begin
                seq_cnt <= seq_cnt + 1'b1;
                Data_err <= 1'b0;
            end
        end
////////////////////////////////////////////////////////////////////////////
        10: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        11: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        12: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1; data2[17:2] <=
ad7606_db_i;data_mult_ch[17:2] <=ad7606_db_i;end

        13: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        14: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        15: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1; data2[1:0] <=
ad7606_db_i[15:14];data_mult_ch[1:0] <=ad7606_db_i[15:14];end // 通道 2
////////////////////////////////////////////////////////////////////////////
        16: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        17: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        18: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1; data3[17:2] <=
ad7606_db_i;data_mult_ch[17:2] <=ad7606_db_i;end

        19: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        20: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        21: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1; data3[1:0] <=
ad7606_db_i[15:14];data_mult_ch[1:0] <=ad7606_db_i[15:14];end // 通道 3
////////////////////////////////////////////////////////////////////////////
        22: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        23: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        24: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1; data4[17:2] <=
ad7606_db_i;data_mult_ch[17:2] <=ad7606_db_i;end

        25: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        26: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
```

```
        27: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1; data4[1:0] <=
ad7606_db_i[15:14];data_mult_ch[1:0] <=ad7606_db_i[15:14];end // 通道 4
//////////////////////////////////////////////////////////////////////////////
        28: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        29: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        30: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1; data5[17:2] <=
ad7606_db_i;data_mult_ch[17:2] <=ad7606_db_i;end

        31: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        32: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        33: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1; data5[1:0] <=
ad7606_db_i[15:14];data_mult_ch[1:0] <=ad7606_db_i[15:14];end // 通道 5
//////////////////////////////////////////////////////////////////////////////
        34: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        35: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        36: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1; data6[17:2] <=
ad7606_db_i;data_mult_ch[17:2] <=ad7606_db_i;end

        37: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        38: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        39: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1; data6[1:0] <=
ad7606_db_i[15:14];data_mult_ch[1:0] <=ad7606_db_i[15:14];end // 通道 6
//////////////////////////////////////////////////////////////////////////////
        40: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        41: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        42: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1; data7[17:2] <=
ad7606_db_i;data_mult_ch[17:2] <=ad7606_db_i;end

        43: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        44: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        45: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1; data7[1:0] <=
ad7606_db_i[15:14];data_mult_ch[1:0] <=ad7606_db_i[15:14];end // 通道 7
//////////////////////////////////////////////////////////////////////////////
        46: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        47: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        48: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1; data8[17:2] <=
ad7606_db_i;data_mult_ch[17:2] <=ad7606_db_i;end

        49: begin ad7606_rd_n_o <= 0; seq_cnt <= seq_cnt + 1'b1;end
        50: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1;end
        51: begin ad7606_rd_n_o <= 1; seq_cnt <= seq_cnt + 1'b1; data8[1:0] <=
ad7606_db_i[15:14];data_mult_ch[1:0] <=ad7606_db_i[15:14];end // 通道 8
//////////////////////////////////////////////////////////////////////////////
        52: begin seq_cnt <= seq_cnt + 1'b1; Conv_Done <= 1;end
        53: begin seq_cnt <= seq_cnt + 1'b1; Conv_Done <= 0;ad7606_cs_n_o <=
1'b1;end
        54: begin seq_cnt <= seq_cnt + 1'b1;  end
        55: begin seq_cnt <= seq_cnt + 1'b1;  end
```

```
        56: begin seq_cnt <= seq_cnt + 1'b1;  end
        57: begin seq_cnt <= seq_cnt + 1'b1;  ad7606_reset_o <= 0;end
        58: begin seq_cnt <= seq_cnt + 1'b1; end
        59: begin seq_cnt <= seq_cnt + 1'b1; cnt_DataNum <= cnt_DataNum + 1'b1;
            if(cnt_DataNum >= 0)begin//舍去第一轮
                Data_En <= 1;
            end
        end
        60: begin
            if (cnt_DataNum >= DataNum +2) begin  // 判断采集转换的数量是否完成指令所
需
                state <= Mode_Sel;              // 然后跳回到 Mode_Sel 状态，等待新的
指令，硬件？寄存器表？单个寄存器
                cnt_DataNum <= 0;
                cnt_table <= 0;
                cnt_high <= 0;
                cnt_low <= 0;
                seq_cnt <= 0;
                Data_En <= 0;
            end  else begin
                seq_cnt <= 0;
            end
        end
    endcase
end
```

为了保证 ADC 每个通道的采样速率能够达到 1MSPS，这里我们选择在 busy 信号的上升沿采样数据。此时，如果读取数据，得到的便是 ADC 上一轮采样的结果，因此，状态机通过 cnt_DataNum 对采样进行计数，并通过 Data_En，将第一轮读取到的数据舍弃，确保数据的正确性。

由于 ADC 的精度为 18 位，而总线位宽只有 16 位，因此，每个 ADC 通道转换的数据，都需要通过两个 RD 脉冲读取。第一次读脉冲，ADC 会通过总线输出高 16 位数据，第二次读脉冲，ADC 会输出低 2 位数据，拼接后便可得到完整的转换数据。

模块通过 24 位的 data_mult_ch（高 6 位为 0）输出转换数据，通过 data_flag 作为标志信号，指示当前数据对应的通道以及数据的有效性。当 8 个通道的采样结果全部读取完成，便代表了一轮传输的结束。此时状态机会产生 Conv_Done 信号脉冲，并通过 cnt_DataNum 对采样轮次进行计数。状态机会将 cnt_DataNum 与所需的采样轮次（DataNum +2），进行比较，从而判断采样是否结束。当采样轮次达到用户需求，便代表本次 ADC 采样完成，此时，状态便会回到 Mode_Sel 态，等待下一个用户指令。否则，便代表本次 ADC 采样还未结束，继续开始下一轮 ADC 采样。

1.2.1.3.3 数据拼接

ADC 单个通道转换后的数据为 18bit，驱动模块通过高位补零的形式，将其拓展为 24 位输出。然而，后级 FIFO 的写入位宽为 32 位，每次需要写入 4 字节数据。为了避免造成

资源上的浪费，设计还会对数据进行拼接。通过 sample24_to_word32 模块，将筛选后的采样数据连续拼接为 32 位。模块收到 fifowrreq 后执行的拼接逻辑如下。

```
if (fifowrreq) begin
    phase <= phase + 1'b1;
    case (phase)
        2'd0: begin
            remain_data <= data_mult_ch;
        end
        2'd1: begin
            din <= {data_mult_ch[7:0], remain_data};
            dvalid <= 1'b1;
            remain_data <= {8'd0, data_mult_ch[23:8]};
        end
        2'd2: begin
            din <= {data_mult_ch[15:0], remain_data[15:0]};
            dvalid <= 1'b1;
            remain_data <= {16'd0, data_mult_ch[23:16]};
        end
        2'd3: begin
            din <= {data_mult_ch, remain_data[7:0]};
            dvalid <= 1'b1;
            remain_data <= 24'd0;
        end
    endcase
end
```

结合代码来看，phase 每接收一个有效样本加 1，remain_data 用来暂存还没有写入 FIFO 的字节。以依次收到 A、B、C、D 四个样本为例，收到 A 时先将它暂存。收到 B 时，取 B 的最低字节补到 A 的高位，组成第一个 32 位字，同时将 dvalid 置 1。

接下来，收到 C 时，用 C 的低 16 位接上 B 剩余的 16 位。收到 D 时，把 D 的 24 位接上 C 剩余的 8 位。这样，四个样本共 12 字节，正好写成三个 32 位字。每个字按低字节在前的顺序存放，这样上位机按字节流每 3 字节就能还原一个采样值。

这里的 phase 按收到的样本计数，可以跨采样轮次。例如只使能 3 个通道，第一轮收到 A、B、C，下一轮的第一个样本就作为 D，继续完成这组拼接。实验中的采样点数按每通道整数 K 设置，1 K 为 1024 点，因此无论选择几路，总样本数都是 4 的倍数。

如果自行发送非整 K 的采样指令，需保证本次请求的总字节数能被 4 整除。例程的 AXI4-Stream 发送逻辑按 ByteToTrans[25:2] 计算需要发送的字数，末尾不足 4 字节的部分不会进入本次 DMA 数据流。完成拼接后，数据进入 FIFO，接下来由 AXI DMA 将其搬运到 DDR。

1.2.1.3.4 AXI DMA 数据搬运

fifo2axis_write 通过 AXI-Stream 接口发送的数据会给到 AXI DMA，由 AXI DMA 通过 AXI4 接口（M_AXI_S2MM），经由 HP0 搬运到 PS 侧的 DDR 中存储。

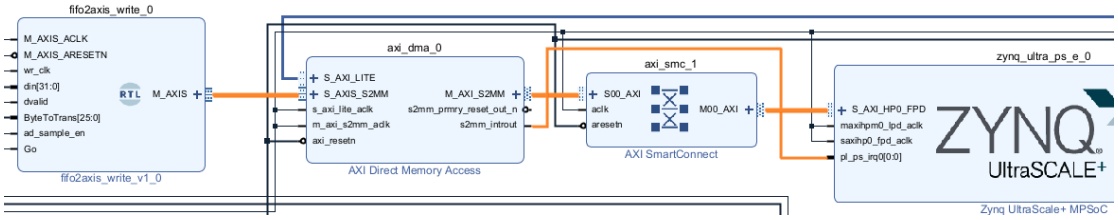

图 1-51 AXI DMA 数据搬运和中断连接

由于 DMA 长度字段有限，一次完整的采集请求可能需要通过多段 DMA 传输完成。每段传输完成后，DMA 都会产生一次完成中断。PS 根据剩余待搬运的字节数判断是否需要提交下一段 DMA 传输，直至本次请求的数据全部写入 DDR。

1.2.1.3.5 指令接收与数据发送

ADC 采样数据通过 AXI DMA 搬运至 PS 侧 DDR 后，PS 端程序读取数据，通过 lwIP 协议栈和 PS 以太网控制器发送给上位机。本次设计将 GEM0 的 GMII 接口通过 EMIO 引出至 PL，经 GMII to RGMII IP 完成接口转换后，连接板载以太网 PHY 芯片，最终通过 PL 网口发送至 PC，也就是下图所示链路：

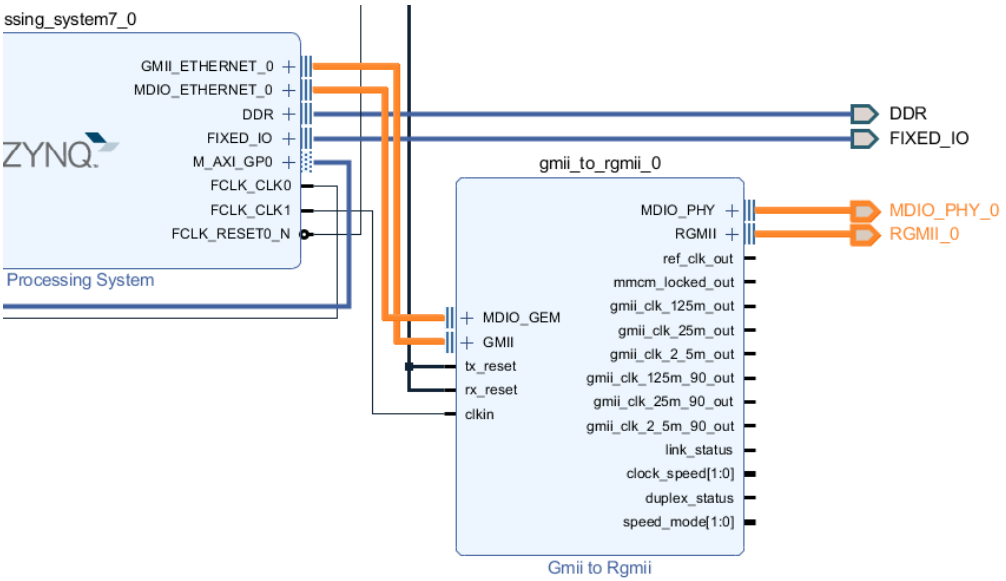

图 1-52 PL 以太网链路

同理，用户指令沿该链路的反方向传入，由 PS 端程序通过 lwIP 接收和解析，再通过 AXI 接口配置 PL 端的采样参数。

以上，便是整个 PL 端的硬件逻辑系统的工作流程和设计思路。由于设计涉及到了 PL 端的资源，所以我们还需要为设计分配管脚。

1.2.1.4管脚分配

本次设计，AC920 的管脚约束如表 1-12 所示。

表 1-12 管脚约束表

信号名	引脚		信号名	引脚
ad7606_os_o_0[2]	U9		ad7606_db_0[4]	T4
ad7606_os_o_0[1]	U12		ad7606_db_0[3]	V4
ad7606_os_o_0[0]	U11		ad7606_db_0[2]	V5

ad7606_busy_i_0	V7		ad7606_db_0[1]	U5
ad7606_convst_o_0	AB2		ad7606_db_0[0]	U6
ad7606_cs_n_o_0	W7		RGMII_0_rxc	W17
ad7606_rd_n_o_0	Y11		RGMII_0_rx_ctl	Y15
ad7606_reset_o_0	Y10		RGMII_0_rd[0]	AA14
ad7606_wr_n_o_0	U10		RGMII_0_rd[1]	Y14
Frst_Data_0	AB1		RGMII_0_rd[2]	W15
ad7606_db_0[15]	AB11		RGMII_0_rd[3]	U16
ad7606_db_0[14]	AA11		RGMII_0_txc	W16
ad7606_db_0[13]	AB12		RGMII_0_tx_ctl	U15
ad7606_db_0[12]	AA12		RGMII_0_td[0]	W18
ad7606_db_0[11]	W12		RGMII_0_td[1]	Y16
ad7606_db_0[10]	V12		RGMII_0_td[2]	U20
ad7606_db_0[9]	V9		RGMII_0_td[3]	V20
ad7606_db_0[8]	V10		MDIO_PHY_0_mdio_io	AB20
ad7606_db_0[7]	T6		MDIO_PHY_0_mdc	AA19
ad7606_db_0[6]	R6		ETH_RESET	Y19
ad7606_db_0[5]	U4			

按照管脚分配表完成管脚分配后我们还需对管脚电平约束，将管脚电平设置为 3.3V 即可，即设置为 LVCMOS33*。

完成以上操作后，为设计生成 bitstream，随后将 bit 与硬件一起导出，便可以打开 vitis，开始应用程序设计。

1.2.2 PS 端应用程序设计

打开 vitis 后，我们首先需要根据导出的.xsa 文件创建一个 platform 工程，并使能 LWIP 库，如图 1-53 所示：

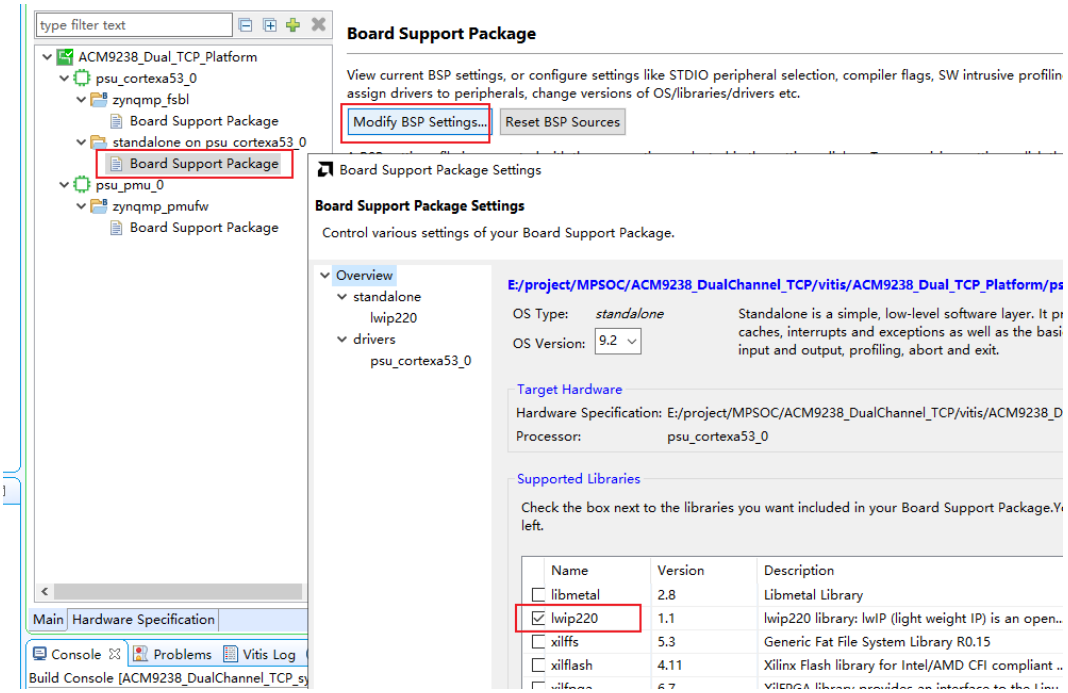

图 1-53 使能 lwip 库

随后对 platform 工程进行编译，等待编译完成后，开始应用工程的搭建。PS 端的应用程序需要基于 xilinx 提供的 LWIP Echo Server 模板进行搭建，也就是在创建 application project 时，选择 LWIP Echo Server 模板，如图 1-54 所示：

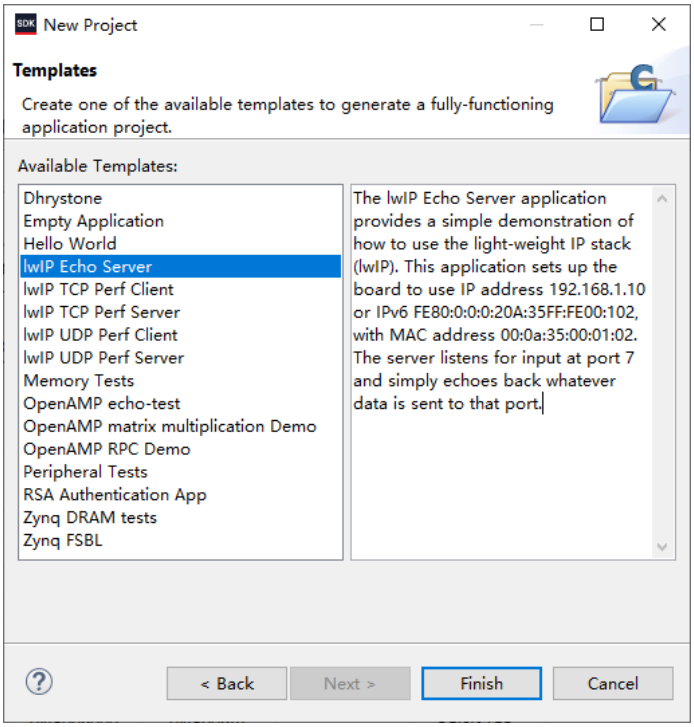

图 1-54 选择工程模板

该模板工程会将开发板设置为服务端，并设置开发板的 IP 地址和 MAC 地址，侦听指定端口上的 TCP 数据包，回环发送。

1.2.2.1修改库函数（仅针对 RTL8211F，AC880 不需要操作）

TCP_ECHO 模版默认使用 Realtek 的 RTL8211E 芯片，如果开发板上使用的是 RTL8211F 芯片，则需要修改 xilinx 官方库中的 xemacpsif_physpeed.c 文件（如果使用的是 YT8531，则不需要修改）。首先，在板级支持包中找到 xemacpsif_physpeed.c 文件，如图 1-55 所示：

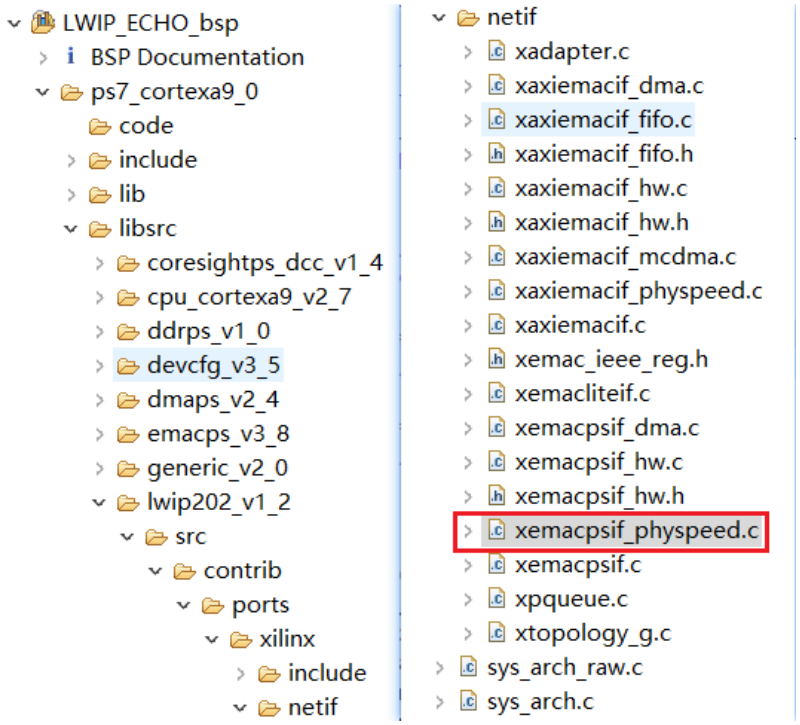

图 1-55 打开 xemacpsif_physpeed.c 文件

然后，在文件的头文件包含后面增加以下代码，重新定义寄存器地址以及掩码等数值

```
//此处为了兼容 RTL8211FDI 重定义寄存器地址和掩码
#undef IEEE_SPECIFIC_STATUS_REG
#undef IEEE_SPEED_MASK
#undef IEEE_SPEED_1000
#undef IEEE_SPEED_100

#define IEEE_SPECIFIC_STATUS_REG        0X1A
#define IEEE_SPEED_MASK                 0x30
#define IEEE_SPEED_1000                 0x20
#define IEEE_SPEED_100                  0x10
```

再接着，找到文件里的 get_Realtek_phy_speed 函数，将下图所示位置由“0x400”修改为“0x4”。

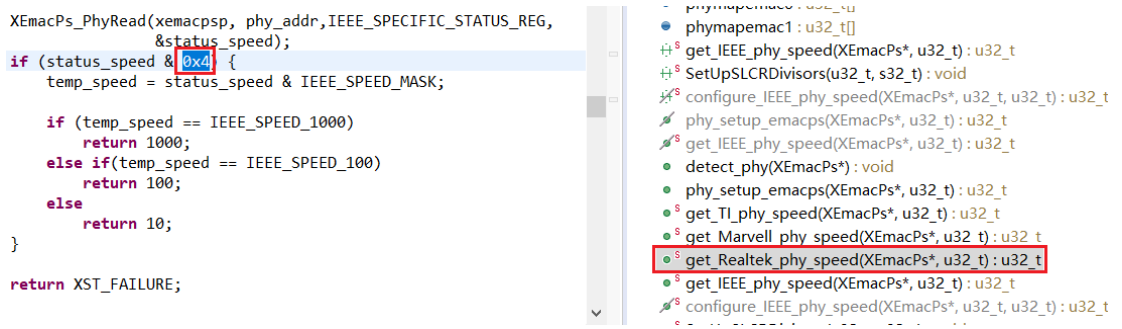

图 1-56 修改速率获取函数

最后，在 626 行左右的“xil_printf("Start PHY autonegotiation \r\n");”语句下方添加以下内容：

```
//关闭 RGMII 的 TX 时钟内部延时
#if 1
    XEmacPs_PhyWrite(xemacpsp, phy_addr, 0x1F, 0x0D08);
    XEmacPs_PhyWrite(xemacpsp, phy_addr, 0x11, 0x0009);
    XEmacPs_PhyWrite(xemacpsp, phy_addr, 0x1F, 0x0000);
#endif
#if 1
    //参考 uboot 启动
    XEmacPs_PhyWrite(xemacpsp, phy_addr, 0x1F, 0x0D04);
    XEmacPs_PhyWrite(xemacpsp, phy_addr, 0x10, 0x617F);
    XEmacPs_PhyWrite(xemacpsp, phy_addr, 0x1F, 0x0000);
#endif
```

该段语句仅在 PS 以太网控制器通过 EMIO 控制 PL 网口时需要添加。需要注意的是，以上所有配置均是对 bsp 中的文件进行修改，一旦 bsp 重新编译，程序就会从官方源文件重新拷贝，并对文件进行覆盖。因此，要想修改永久有效，需要修改官方库函数的源文件，相关修改方法可参考总结内容。

如此，便完成了驱动库对于 RTL8211F 的适配。接下来便可以开始修改设计代码了。

1.2.2.2代码修改

由于我们在 ZYNQ 与 GMII to RGMII 间插入了一个 ft_gmii_retime 模块，这会导致原本会自动识别的 GMI tO RGMI 核的地址宏缺失，这里我们需要在 xemacpsif_p.yspeed.c 中手动对其进行补充（未添加便不需要补充）。在该文件的头文件声明下方，添加以下内容：

```
/* 当前工程 GMII to RGMII 转换器的 MDIO 地址为 31 */
#ifndef XPAR_GMII2RGMIICON_0N_ETH0_ADDR
#define XPAR_GMII2RGMIICON_0N_ETH0_ADDR 31
#endif
```

注意，这里的地址编号需要与 GMII to RGMII 中设置的 PHY 地址对应，由于图 1-38 中，我们将 PHY 地址设置成了 31，因此这里也相对应地设置为 31。

修改完成后，保存设计。为了实现设计目标，我们还需要对应用工程中的 echo.c、main.c、platform_zynq.c 三个文件内容进行修改，由于涉及到的内容较多，用户可以直接从例程中拷贝替换。同时，用户还需要将 ADC_Info.h、CMD_Handle.c、CMD_Handle.h 文件拷贝到工程中来。以上涉及到的六个文件，其内部函数主要实现的功能如下表 1-13：

表 1-13 源文件功能说明

源文件	功能
main.c	硬件以及 LWIP 初始化、接收指令处理、采样数据的发送等
echo.c	TCP 端口绑定以及接收处理
platform_zynq.c	ZYNQ 硬件初始化，中断使能
ADC_Info.h	存储 ADC 相关信息
CMD_Handle.c	用户指令处理
CMD_Handle.h	CMD_Handle.c 中相关变量和函数声明以及宏定义

将所有文件拷贝至工程并添加 src 文件夹至工程路径后，应用工程便搭建完毕。下面，我们将按照程序运行流程，逐步分析代码实现。首先是主函数中关于 LWIP 的初始化。

1.2.2.3LWIP 功能初始化

LWIP 功能的初始化，该部分内容涉及较多，首先是 platform 的初始化，通过 init_platform()函数完成。该函数位于 platform_zynq.c 中，函数内容如下：

```
void init_platform()
{
    platform_setup_interrupts();
    platform_setup_timer();
    platform_setup_axidma();

    return;
}
```

该函数中调用了以下三个函数：

表 1-14 硬件初始化相关函数

函数	功能
platform_setup_interrupts()	初始化 GIC，注册异常中断、私有定时器中断和 AXI DMA 接收中断的处理函数，开启 GIC 中对于私有定时器和 AXI DMA 接收中断的使能
platform_setup_timer()	初始化私有定时器，设置装载值
platform_setup_axidma()	初始化 AXI DMA，设定中断触发极性和优先级

这里的私有定时器，用于定时处理 TCP 超时事件，因此设计会用到私有定时器中断。而 AXI DMA 的数据搬运也会用到中断，为了避免出现多中断冲突的问题，设计中将 AXI DMA 中断的初始化也放在了 platform_setup_interrupts 函数中。这里 AXI DMA 的接收中断处理函数被链接为 RxIntrHandler，函数内容会在 DMA 中断处理与数据发送小节讲解。

硬件初始化完成后，代码中通过 lwip_init()函数初始化 LWIP，然后使能硬件中断，初始化网络接口。该过程中会修改开发板的 IP 地址和网关地址，为了方便实验，这里将默认的 IP 地址修改为 192.168.0.2，网关地址修改为 192.168.0.1。

```
/* initialize IP addresses to be used */
IP4_ADDR(&ipaddr,  192, 168,   0, 2);
IP4_ADDR(&netmask, 255, 255, 255,  0);
IP4_ADDR(&gw,      192, 168,   0,  1);
```

除了 IP 地址之外，代码中也将 MAC 地址修改为了 00_0a_35_01_fe_c0。

```
/* the mac address of the board. this should be unique per board */
unsigned char mac_ethernet_address[] =
{ 0x00, 0x0a, 0x35, 0x01, 0xfe, 0xc0 };
```

1.2.2.4建立 TCP 通信

TCP 通信的建立由 start_application()函数完成，函数具体代码如下：

```
int start_application()
{
    struct tcp_pcb *pcb;
    err_t err;
    unsigned port = 5000;

    /* create new TCP PCB structure */
    pcb = tcp_new_ip_type(IPADDR_TYPE_ANY);
    if (!pcb) {
        xil_printf("Error creating PCB. Out of Memory\n\r");
        return -1;
    }

    /* bind to specified @port */
    err = tcp_bind(pcb, IP_ANY_TYPE, port);
    if (err != ERR_OK) {
        xil_printf("Unable to bind to port %d: err = %d\n\r", port, err);
        return -2;
    }

    /* we do not need any arguments to callback functions */
    tcp_arg(pcb, NULL);

    /* listen for connections */
    pcb = tcp_listen(pcb);
    if (!pcb) {
        xil_printf("Out of memory while tcp_listen\n\r");
        return -3;
    }

    /* specify callback to use for incoming connections */
    tcp_accept(pcb, accept_callback);

    xil_printf("TCP echo server started @ port %d\n\r", port);

    return 0;
}
```

函数主要是实现 TCP 端口的绑定以及将服务端设置为监听模式，代码中涉及到的主要函数如表 1-15 所示：

表 1-15 建立 TCP 通信相关函数

函数	功能
tcp_new_ip_type()	创建新的 TCP 控制块（PCB）
tcp_bind()	将本地 IP 地址、端口号与 PCB 绑定，将绑定完毕的控制块插入 pcb 链表中
tcp_arg()	设置 TCP 连接的回调函数参数
tcp_listen()	将 TCP 控制块设置为监听状态，等待客户端连接
tcp_accept()	注册新连接建立后的回调函数

结合表格可知，start_application()函数依次进行了以下四个操作:

（1） 通过 tcp_new_ip_type ()函数创建新 PCB
（2） 使用 tcp_bind()绑定 IP 地址和端口号
（3） 使用 tcp_listen()对连接进行监听
（4） 使用 tcp_accept()设置连接回调函数。

这也是 TCP 服务端与客户端建立连接的一般步骤。值得注意的是，这里的 TCP_PCB 又被称之为 TCP 控制块，是一个十分重要的结构体。其内部定义了大量的成员变量，基本定义了整个 TCP 协议运作过程所需的关键参数/字段，如发送窗口、接收窗口、数据缓冲区、超时处理、拥塞控制、滑动窗口等。

除了普通的 TCP 控制块外，lwIP 还定义了用于监听状态的结构体 tcp_pcb_listen，它只保留监听连接所需的字段，以减少内存占用。tcp_listen()会创建监听控制块，并从原控制块中保留本地地址、端口等监听信息，随后释放原控制块。当客户端建立连接后，lwIP 会为该连接创建独立的 TCP 控制块，再通过 accept_callback()将其交给应用程序处理。

accept_callback()注册 recv_callback()，后续收到 TCP 数据时由 lwIP 调用它，该函数的代码如下：

```
err_t accept_callback(void *arg, struct tcp_pcb *newpcb, err_t err)
{
    static int connection = 1;

    /* set the receive callback for this connection */
    tcp_recv(newpcb, recv_callback);

    /* just use an integer number indicating the connection id as the
       callback argument */
    tcp_arg(newpcb, (void*)(UINTPTR)connection);

    /* increment for subsequent accepted connections */
    connection++;
    serpcb=newpcb; // 将建立的 pcb 赋予全局变量 serpcb;
    return ERR_OK;
}
```

这段代码主要实现以下两个功能：

1、设置了接收回调函数 recv_callback（），并将连接 ID 作为回调参数。

2、保存当前连接的 PCB 指针，用于主函数中服务端将 ADC 采样的数据发送到客户端

1.2.2.5用户指令提取

当接收到客户端的数据包后，回调函数 recv_callback()会被调用。此时，我们需要它根据接收到的数据还原出指令。该函数的代码内容如下：

```
err_t recv_callback(void *arg, struct tcp_pcb *tpcb,
                                struct pbuf *p, err_t err)
{
    int i;
    /* do not read the packet if we are not in ESTABLISHED state */
    if (!p) {
        tcp_close(tpcb);
        tcp_recv(tpcb, NULL);
        return ERR_OK;
    }

    /* indicate that the packet has been received */
    tcp_recved(tpcb, p->len);

    //对接收到的数据进行分析，还原出指令内容
    unsigned char cmd_num = (p->len)/8;//计算指令数

    if (tcp_sndbuf(tpcb) > p->len) {
        for(i=0;i<cmd_num;i++)
        {
            cmd_analysis(p->payload + 8*i);
        }

//  /* echo back the payload */
//  /* in this case, we assume that the payload is < TCP_SND_BUF */
//  if (tcp_sndbuf(tpcb) > p->len) {
//      err = tcp_write(tpcb, p->payload, p->len, 1);
    } else
        xil_printf("no space in tcp_sndbuf\n\r");

    /* free the received pbuf */
    pbuf_free(p);

    return ERR_OK;
}
```

其中涉及到的函数，功能如下：

表 1-16 函数功能表

函数	功能
tcp_close()	关闭 PCB 连接
tcp_recv()	注册或取消接收回调函数
tcp_recved()	通知协议栈应用已处理接收数据，更新接收窗口

tcp_sndbuf()	获取发送 buffer 的大小
pbuf_free()	释放数据包 pbuf 资源
cmd_analysis()	指令分析，将接收到的数据按照帧格式转换为对应的指令

回调函数会确认接收数据的长度，并根据长度计算出包含多少个完整的指令，只要数据不超过接收缓冲区，就会通过 cmd_analysis()函数对数据进行解析。解析完成后，通过 pbuf_free()释放掉 pbuf 所占用的内存，以免造成内存资源浪费。

这里 cmd_analysis()函数代码内容如下：

```
void cmd_analysis(uint8_t *cmd_buffer)
{
    unsigned char cmd[8] = {0};
    for(int i=0;i<8;i++){
        cmd[i] = *(cmd_buffer+i);
    }

    if((cmd[0] == 0x55) && (cmd[1] == 0xA5) && (cmd[7] == 0xF0))
    {
        switch(cmd[2])
        {
            case 0: restart = 1;break;
            case 1: ChannelSel = cmd[6];break;
            case 2: DataNum = (cmd[3] << 24) | (cmd[4] << 16) | (cmd[5] << 8) |
cmd[6];break;
            case 3: Speed_Set = (cmd[3] << 24) | (cmd[4] << 16) | (cmd[5] << 8) |
cmd[6];break;
            case 4: ADC_Mode = (cmd[3] << 24) | (cmd[4] << 16) | (cmd[5] << 8) |
cmd[6];break;
            case 255: devinfo = 1; break;
            default:break;
        }
    }
}
```

函数会以每 8 个字节为一组，按照表 1-6 中的指令格式对数据进行分析，还原出指令内容和对应的指令数据，并给出对应的标志信号。这里支持的指令共有 6 个，指令 0~3 为一组，用来控制采样使能、采样通道、采样个数、采样速率，当采样使能为 1 时指令才会被执行。

指令 4 为一组，用于 ADC 软件模式下的功能选择/配置。

指令 255 为一组，用于上位机查询当前 ADC 的相关参数，从而方便上位机进行参数设定。

最终，这些指令和指令数据会在主函数的 while 循环中进行处理。

1.2.2.6指令处理

首先是指令 255 的处理，指令 255 用于获取 ADC 参数（新版上位机暂不支持该指令）。上位机可以通过下发指令查询 ADC 的参数。此时设计便会通过 Send_ADC_Info()函数，将预先设定好的参数信息发送给上位机。该函数代码如下：

```
void Send_ADC_Info()
{
    if(devinfo == 1)
    {
        memcpy(buffer, &ACM7606C_18_info, sizeof(ADC_info));
        tcp_write(serpcb,&ACM7606C_18_info,sizeof(ADC_info),1);
        tcp_output(serpcb);
        pbuf_free(serpcb->refused_data);
        devinfo = 0;
    }
}
```

函数会将结构体 ADC_info 中的内容拷贝出来，然后通过 TCP 传输给上位机。需要注意的是，这里的 tcp_write()函数只是用于组建 TCP 报文，实际上数据的发送是通过 tcp_output()函数实现。发送完成后，通过 pbuf_free()函数释放掉内存，然后将 devinfo 标志信号置 0。

这里的 ADC_info 结构体存储着 ACM7606C_18 的参数信息，该部分内容位于 ADC_Info.h 中，对应代码如下：

```
typedef struct{
    char Company[16];
    unsigned int ID;//ID 序列号
    char Label[16];//产品型号
    unsigned int Max_Sample;//当前最大采样率
    unsigned char Channel_Num;//最大通道数
    unsigned char Data_Type;//数据类型,1 为补码，0 为无符号
    unsigned char Resolution;//分辨率
    unsigned char mode;//模式
    unsigned int reserved;//预留

    float Max_Volta;//最大输入电压
    float Min_Volta;//最小输入电压
    unsigned int Clk_Freq;//工作时钟 Hz
    float Cycle;//周期 ns
    unsigned char reserve[192];//确保共 256 字节
}__attribute__((__packed__))ADC_info;


static const ADC_info ACM7606C_18_info = {
    .Company = "xiaomeigefpga",
    .ID = 0x00000003,
    .Label = "ACM7606C-18",
    .Max_Sample = 1000000,
```

```
    .Channel_Num= 8,
    .Data_Type   = 1,
    .Resolution   = 18,
    .mode = 1,
    .reserved = 0,
    .Max_Volta  = 10.0,
    .Min_Volta  = -10.0,
    .Clk_Freq   = 100000000,
    .Cycle   = 10,
    .reserve = {0}
};
```

该函数作用主要是为上位机提供当前 ADC 信息，方便上位机做进一步处理。

然后是 ADC 的用户配置指令，本次设计的配置指令共有 5 种，对应表 1-7 中的指令 0~4。5 个指令分为两组，0~3 为一组，用于控制 AD7606C-18 的采样。

当指令 0，也就是 restart 指令有效时，代表新的传输即将开始。此时，需要按照指令 1~3 的内容配置 ADC 的采样通道、采样个数、采样速率，然后使能 ADC 采样。同时，还需要提交 AXI DMA 接收传输，并使能完成中断，以便在数据搬运完成后进行处理。对应的代码如下：

```
void Restart_ADC_Sample(u32 Bytes)
{
    uint32_t Trans_Turn;//每个通道采样的次数
    uint32_t Channel_SUM;//通道数

    if(restart)
    {
        //清零 ADC_Mode
        ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,START_MODE,ADC_Mode);

        //存储要发送的数据量
        send_data = Bytes;
        Channel_SUM = count_bits32(ChannelSel);
        Trans_Turn = (send_data/3 + Channel_SUM -1)/Channel_SUM;

        if(Bytes > 0x3ff0000){
            start_trans_data(0x3ff0000, ADC_Data_Addr);
            data_rest = Bytes - 0x3ff0000;
            Bytes = 0x3ff0000;

        }
        else{
            start_trans_data(Bytes, ADC_Data_Addr);
            data_rest = 0;
        }
        //使能 restart
        ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,SAMPLE_EN_GO,1<<16);
```

```
        //总采样字节数
        ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,DATA_SUM,send_data);
        //通道选择
        ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,CH_SEL,ChannelSel);
        //速度设置
        ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,SPEED_CTRL,Speed_Set);
        //本次传输字节数
        ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,BYTES_Trans,Bytes);
        //本次传输 ADC 采样轮次
        ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,DATA_NUM,Trans_Turn);
        //使能 Go 和 sample_en; bit[8]和 bit[0]
        ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,SAMPLE_EN_GO,0x00101);

        restart = 0;

        sample_send_flag = 1;
    }
}
```

函数会先产生 restart 脉冲，代表新一次 ADC 采样的开始。然后，根据指令，计算出 ADC 需要传输的轮次 Trans_Turn，并根据采样数量的大小，设定本段 DMA 传输。如果单段 DMA 传输无法搬运全部数据，函数便会记录剩余待搬运字节数，供后续提交下一段传输使用。

DMA 接收传输通过 start_trans_data()函数提交，代码如下：


```
int start_trans_data(uint32_t len,UINTPTR RxBufferPtr)
{
    int Status;

    /* Disable all interrupts before setup */

    XAxiDma_IntrDisable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DMA_TO_DEVICE);

    XAxiDma_IntrDisable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DEVICE_TO_DMA);

    XAxiDma_IntrEnable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DMA_TO_DEVICE);

    XAxiDma_IntrEnable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DEVICE_TO_DMA);

    /* Initialize flags before start transfer test  */
    RxDone = 0;
    Error = 0;

    Status = XAxiDma_SimpleTransfer(&AxiDmaInstance,RxBufferPtr,
```

```
            len, XAXIDMA_DEVICE_TO_DMA);

    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    return XST_SUCCESS;
}
```

这里的 XAxiDma_SimpleTransfer()函数用于向系统提交一次 AXI DMA 的传输申请，函数中，接收数据的存储起始地址被设定为 0x1800000。

提交本段 AXI DMA 传输后，通过 ADC_SAMPLE_CTRL_mWriteReg()函数，将总采样字节数、采样通道、采样速度、单段 DMA 传输字节数、ADC 采样轮次等信息通过 AXI4-Lite 接口写入 adc_sample_ctrl 模块，再由该模块产生相应信号，实现对 ADC 采样系统的控制。

随后，使能 Go 和 Sample_en 信号，开始新的 ADC 采样传输。

指令 4 则用于软件模式下的配置，代码如下：

```
void Set_ADC_Mode()
{
    if(ADC_Mode){
        //设置 ADC_Mode
        ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,START_MODE,ADC_Mode);
        //产生 restart 脉冲
        ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,SAMPLE_EN_GO,1<<16);
        ADC_Mode = 0;
        ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,SAMPLE_EN_GO,0);
    }
}
```

这里函数并不会对指令 4 做任何处理，而是产生 restart 脉冲，将指令 4 原原本本地交由 adc_sample_ctrl 模块，最终，由 AD7606C_18_Driver 模块进行解析。指令 4 的解析方法，可以参考图 1-23 或者 1.2.1.3.2 小节。

1.2.2.7DMA 中断处理与数据发送

ADC 采样开始后，DMA 会将采样的数据搬运到 DDR 的指定地址中，当搬运的数据量达到预设值时，便会触发接收完成中断。此时，我们需要禁用 DMA 传输使能信号，对应的 DMA 中断处理如下：

```
static void RxIntrHandler(void *Callback)
{
    uint32_t IrqStatus;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);
```

```
    /* Acknowledge pending interrupts */
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

    /*
     * If no interrupt is asserted, we do not do anything
     */
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
        return;
    }

    /*
     * If error interrupt is asserted, raise error flag, reset the
     * hardware to recover from the error, and return with no further
     * processing.
     */
    if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {

        Error = 1;

        /* Reset could fail and hang
         * NEED a way to handle this or do not call it??
         */
        XAxiDma_Reset(AxiDmaInst);

        TimeOut = RESET_TIMEOUT_COUNTER;

        while (TimeOut) {
            if(XAxiDma_ResetIsDone(AxiDmaInst)) {
                break;
            }

            TimeOut -= 1;
        }

        return;
    }

    /*
     * If completion interrupt is asserted, then set RxDone flag
     */
    if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {
        ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,SAMPLE_EN_GO,0x001);//禁用 Go 和
sample_en; bit[8]和 bit[0]
        RxDone = 1;
    }
}
```

当中断状态为 XAXIDMA_IRQ_IOC_MASK，也就是中断完成时，通过 ADC_SAMPLE_CTRL_mWriteReg()函数将 Go 信号置 0，关闭 DMA 传输使能信号。同时，拉高接收完成标志信号 RxDone，用于下一步处理。

当采集数据量较大时，数据搬运可能需要分成多段 DMA 传输。RxDone 置位后，主循环会禁用 DMA 中断，并判断是否还有待搬运的数据，以决定是否提交下一段传输。当本次请求的数据全部搬运完毕后，再通过 TCP 将数据发送给上位机进行波形绘制。

设计中，这部分内容通过 Data_Recive_Trans()函数实现，代码如下：

```
void Data_Recive_Trans()
{
    uint32_t buff;

    if(RxDone) {
        XAxiDma_IntrDisable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DMA_TO_DEVICE);
        XAxiDma_IntrDisable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DEVICE_TO_DMA);
        if(data_rest >0){

            RxDone = 0;
            RX_BUFFER_BASE = RX_BUFFER_BASE + 0x3ff0000;

            if(data_rest > 0x3ff0000){

                ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,BYTES_Trans,0x3ff0000);/
/通道选择
                ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,SAMPLE_EN_GO,0x101);//使
能 sample_en 和 Go; bit[8]和 bit[0]
                start_trans_data(0x3ff0000, RX_BUFFER_BASE);
                data_rest = data_rest - 0x3ff0000;
            }
            else {

                ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,BYTES_Trans,data_rest);/
/通道选择
                ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,SAMPLE_EN_GO,0x101);//使
能 sample_en 和 Go; bit[8]和 bit[0]
                start_trans_data(data_rest, RX_BUFFER_BASE);
                data_rest = 0;
                RX_BUFFER_BASE = ADC_Data_Addr;
            }
        }
        else{
            ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,SAMPLE_EN_GO,0);//置低 Go 和
sample_en; bit[8]和 bit[0]
            buff = tcp_sndbuf(serpcb);
            if(buff >= send_data){
```

```
                Xil_DCacheFlushRange(0x1800000 + uiIdx,send_data);
                tcp_write(serpcb,(void*)(0x1800000 + uiIdx),send_data,1);
                tcp_output(serpcb);
                RxDone = 0;
                send_data = 0;
                sample_send_flag = 0;
                uiIdx = 0;
                pbuf_free(serpcb->refused_data);
            }
            else{
                Xil_DCacheFlushRange(0x1800000 + uiIdx,buff);
                tcp_write(serpcb,(void*)(0x1800000 + uiIdx),buff,1);
                tcp_output(serpcb);
                send_data = send_data - buff;
                uiIdx += buff;
                pbuf_free(serpcb->refused_data);
            }
        }
    }
}
```

由于用户指令给出的采样字节数可能超过单段 DMA 传输长度，所以每当 DMA 接收完成中断触发后，函数都会检查剩余待搬运字节数 data_rest，据此决定是否提交下一段 DMA 传输，以及该段需要搬运的字节数。

而为了确保接收数据的连续，代码会将 DMA 搬运的数据存储到以 ADC_Data_Addr（0x1800000）为起始地址的空间中，等到采样的 ADC 数据满足用户指令给出的字节数时，才会通过 TCP 将数据发送给上位机。

发送前需要先判断待发送数据与 tcp 发送 buffer 容量的关系，如果数据大于发送 buffer 容量，需要分多次读取发送。这里使用 uiIdx 来记录当前发送了多少字节数据，因此待发送数据的地址可以用数据起始地址 0x1800000+uiIdx 组成。

当数据发送完成后，将接收完成标志位 RxDone 拉低，将 uiIdx 清零。同时，为了避免内存的占用，还需要通过 pbuf_free()函数释放掉 tcp 发送 buffer 所占的内存。

以上，便是 PL 端逻辑系统设计和 PS 端应用程序设计的内容，接下来，我们通过实际的硬件来对设计进行验证。

1.3　板级验证

1.3.1 实验所需硬件

3、AC880 开发板 x1
4、ACM7606C 模块 x1
5、信号发生器 x1
6、千兆网线 x1
7、DC 电源线 x1
8、Type-c 下载线 x1

1.3.2 硬件连接

实验硬件连接如图 1-57 所示：

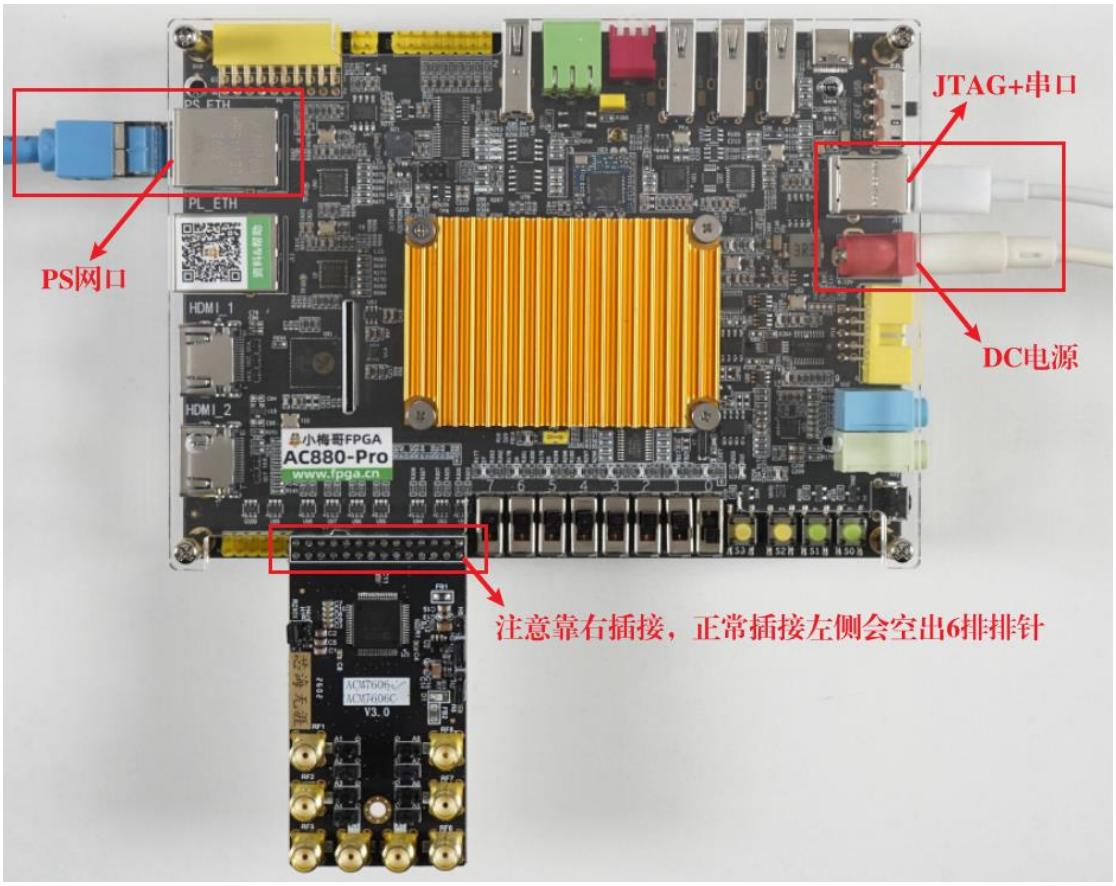

图 1-57 硬件连接

设计使用到的是 PL 侧网口，因此使用千兆网线时，注意网线一端连接开发板 PL 网口（开发板背面丝印 PL_ETH），一端连接电脑网口。ACM7606C 模块与开发板上的 GPIO 进行插接，注意插接时应当 1 脚对 1 脚，正常连接情况下，左侧会空出两排排针。

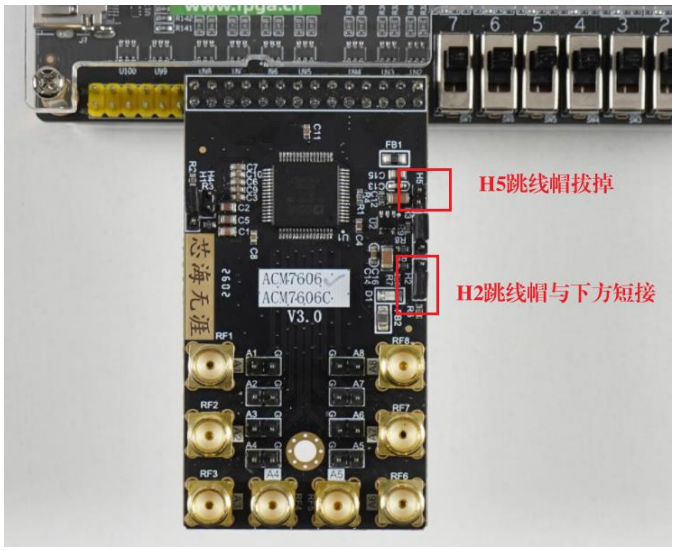

图 1-58 AD7606C 跳线帽设置

另外需要注意的是，由于本次设计使用的是并行接口+软件模式的 AD7606C，因此，在硬件上，需要将 H5 跳线帽拔掉（CONVST 与$\overline{WR}$分开控制），将 H2 与下方短接（选择并口）。连接并设置好硬件后，为开发板上电。

1.3.3 修改电脑 IP 地址

在 LWIP 功能初始化小节中，我们通过代码将开发板 IP 地址设置为了 192.168.0.2，因此，我们需要将电脑 IP 地址设为同一网段。

在电脑上进入【控制面板】->【网络和 Internet】->【查看网络状态和任务】，查看网络连接状态。需要看到在活动网络中有以太网连接存在，才表明开发板和电脑的网络已经连通。至于显示的无法连接到网络选项，意思是指无法连接到互联网获取网络上的数据，这是正常的，无需在意，如图 1-59 所示：



图 1-59 查看网络连接状态

点击“以太网”文字，以查看该网络状态，确认当前连接速度为千兆速率（1000.0 Mbps），如图 1-60 所示：



图 1-60 查看网络速度

在上述本地连接状态中，点击属性，并在弹出的属性对话框中双击【Internet 协议版本 4 （TCP/Ipv4）】选项，然后在弹出的属性对话框中设置静态 IP 地址（默认网关可以不设置）。如图 1-61 所示：

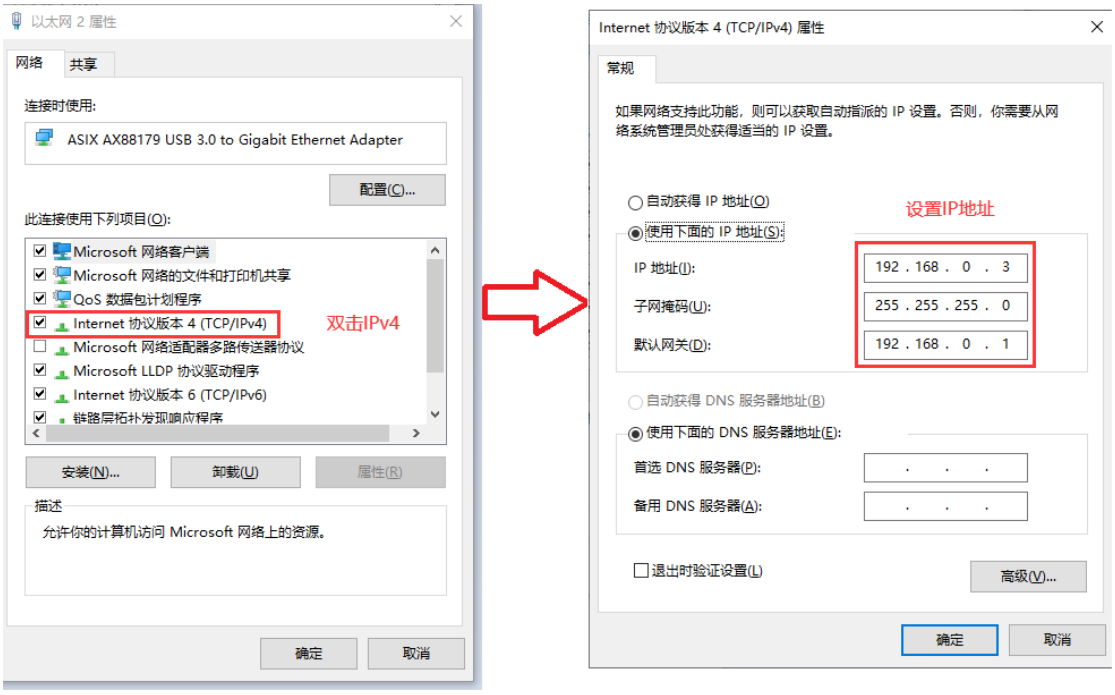

图 1-61 修改 PC 的 IP 地址

如果用户电脑中相关驱动缺失文件，会出现弹窗报错“出现了一个意外的情况，不能完成所有你在设置中所要求的更改。”导致修改失败，对于该情况用户可以参考下帖：

设置静态 IP 时提示“出现了一个意外的情况，不能完成所有你在设置中所要求的更改”

http://www.corecourse.cn/forum.php?mod=viewthread&tid=29229

(出处: 芯路恒电子技术论坛)

更改完 IP 地址后，还需要检查是否有 arp 绑定，以管理员权限打开 CMD，输入以下命令查询 arp：

```
arp -a
```

查询界面如图 1-62 所示：

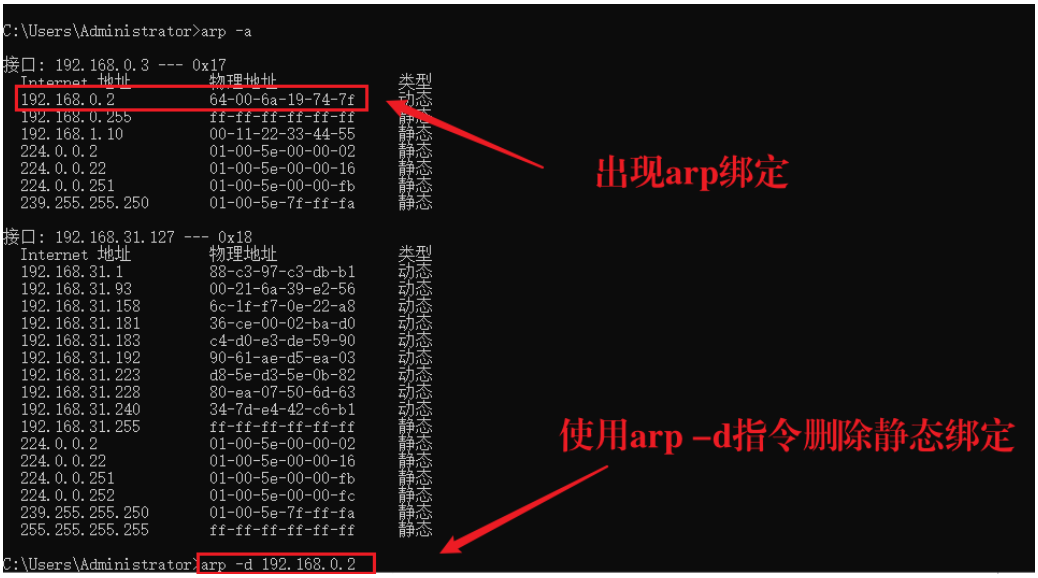

图 1-62 arp 查询与删除

如果出现了图中的静态绑定，则需要使用以下指令进行删除:

```
arp -d 192.168.0.2
```

1.3.4 代码烧录

由于 LWIP 的初始化需要一段时间，因此在烧录代码之前，我们需要连接串口，通过串口打印的信息观察。串口的端口号需要通过设备管理器查询，在端口栏中，名字为 CH9102 的便是 PS 端串口。譬如，这里笔者电脑上分配给开发板 PS 端串口的端口号为 COM11。

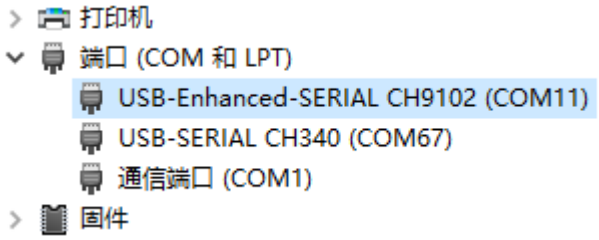

图 1-63 查询端口号

读者连接串口时，请以自己电脑上分配的端口号为准。连接完串口后，便可以烧录程序，LWIP 的初始化需要等待一段时间，当串口打印出图 1-64 所示信息时，便代表初始化完成，此时便可以连接以太网上位机进行功能验证。

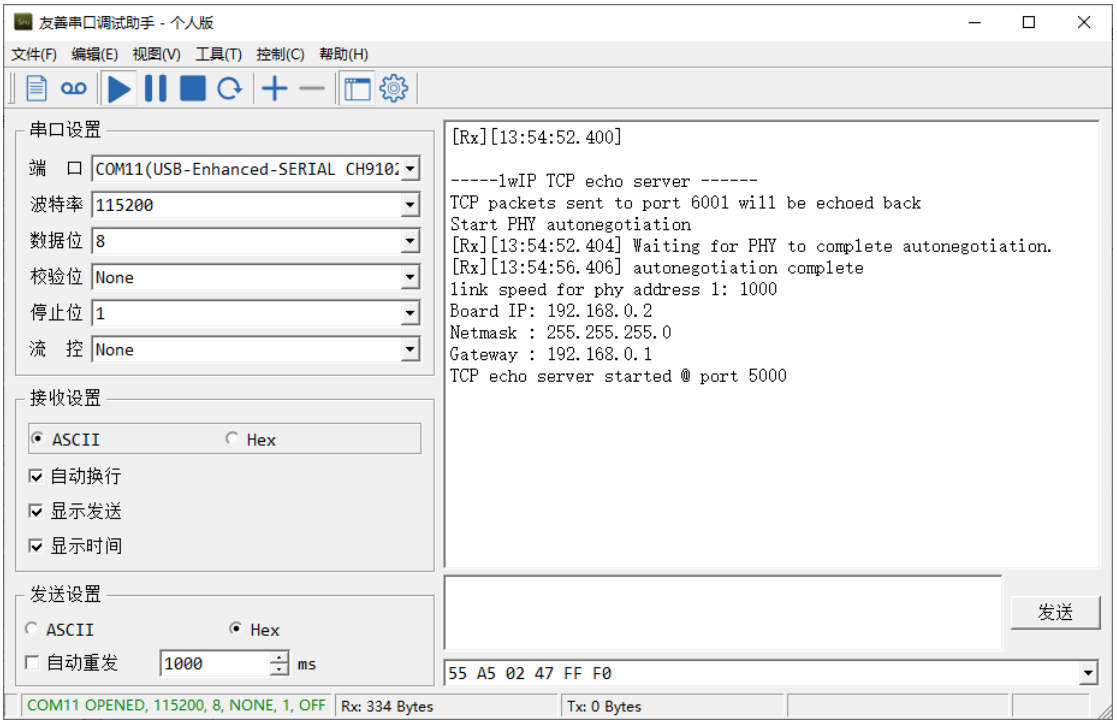

图 1-64 LWIP 初始化完成

1.3.5 功能验证

完成硬件连接和程序下载后，下面对采集链路和寄存器配置进行验证。实验使用 NetAssist 发送指令，并将接收的数据保存为 BIN 文件；再使用 ADC BIN 波形查看器解码并绘制波形。

首先打开 NetAssist，将协议类型设为 TCP Client，远程地址和端口分别填写 192.168.0.2 和 5000，发送模式选 HEX。图中电脑地址为 192.168.0.3，子网掩码为 255.255.255.0。具体配置如下：

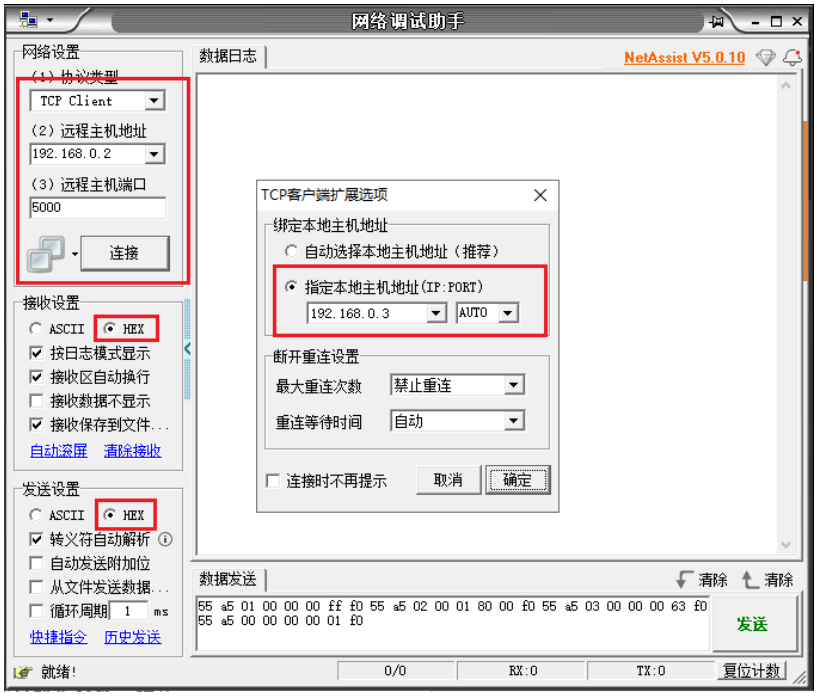

图 1-65 TCP 连接

连接成功后，日志会显示电脑的本地地址和临时端口。临时端口由系统分配，实际测试时可能与图中不同；开发板的服务端口为 5000。

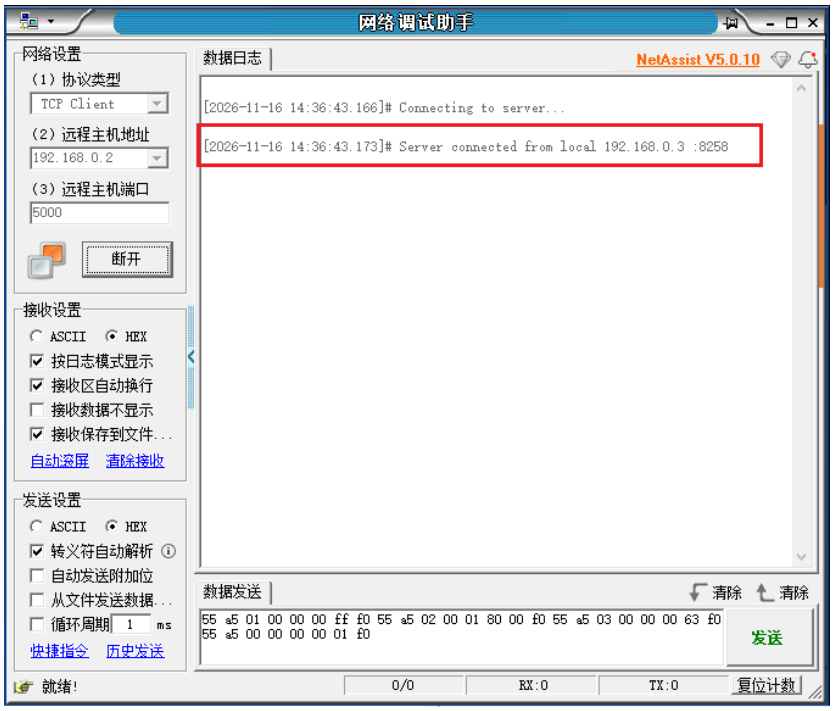

图 1-66 连接成功

接下来勾选“接收保存到文件”，将保存类型设为“数据文件”。每轮采集可使用不同的文件名；如沿用原文件，应先清空文件。发送前可将 NetAssist 的收发计数清零，便于核对本轮收到的字节数。

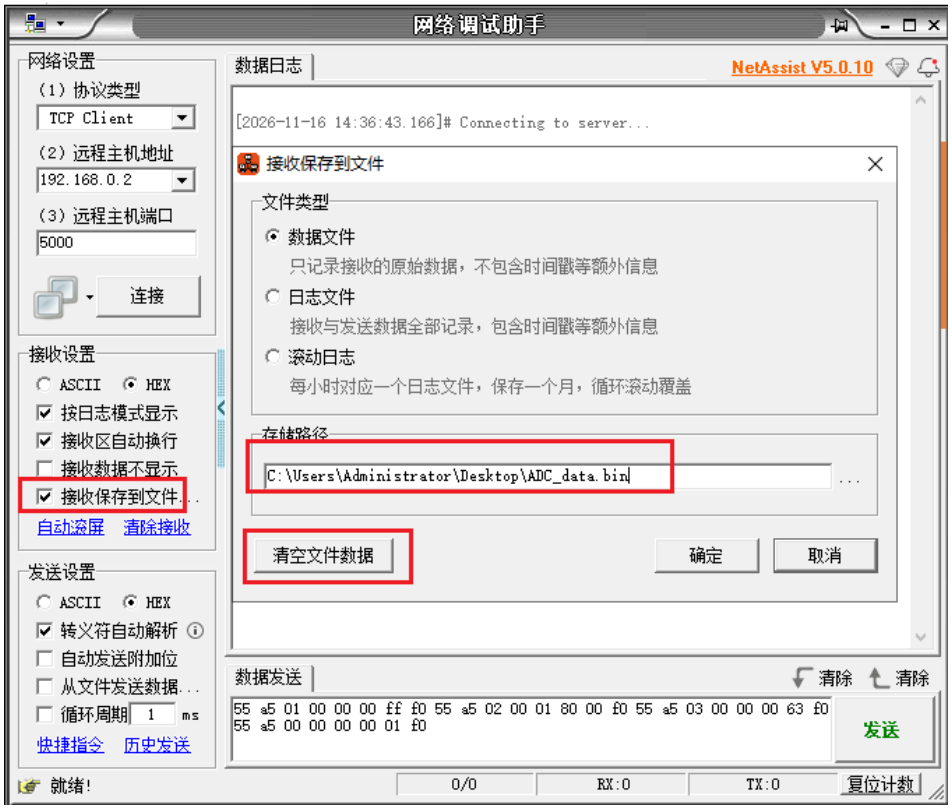

图 1-67 设置接收数据文件

保存路径设置完成后，将信号发生器的两路输出分别接到 CH1 和 CH2，并将信号地接到模块 GND。需要核对波形幅度时，可在 ADC 输入端测量实际峰峰值，作为对照。

1.3.5.1ADC 采样测试

程序运行后，ADC 首先会进行全复位，回到 ±10 V 量程和低带宽模式。这里通过信号发生器为 CH1 产生 2 kHz、幅值 ±10 V 的正弦波，为 CH2 产生 1 kHz、幅值 ±7.5 V 的锯齿波。

打开 netassist，发送以下指令控制 ADC 进行采样：

```
55 a5 01 00 00 00 ff f0 55 a5 02 00 01 80 00 f0 55 a5 03 00 00 00 63 f0 55 a5 00 00 00 00 01 f0
```

上述四帧指令依次设置通道选择、采集长度和采样速率，最后启动采集。指令会启用八个通道，每路采集 4,096 点，每点占 3 字节，因此总数据量为 98,304 字节。速率参数 0x63 对应约 1 MSPS/通道的配置值。各字段的含义可参考图 1-23 和表 1-7。

发送指令后，netassist 会接收到 ADC 转换后的原始数据，如图 1-68 所示。

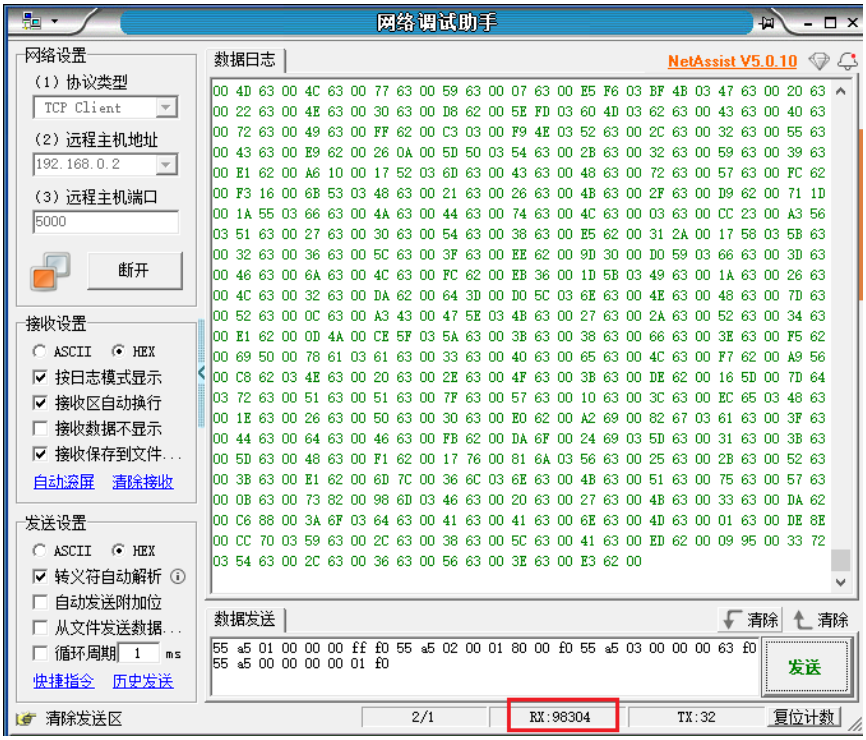

图 1-68 ADC 采样原始数据

图中 NetAssist 的接收计数为 98,304 字节，与指令设置的 0x18000 一致。确认保存的 BIN 文件大小相同后，打开 ADC BIN 波形查看器，按图 1-69 设置 CH1～CH8 的顺序，按采样轮次交织、小端、有符号补码，24 位存储、18 位分辨率（低 18 位有效）；采样率填 1000000 Hz，正满量程填 10 V，纵轴选电压。

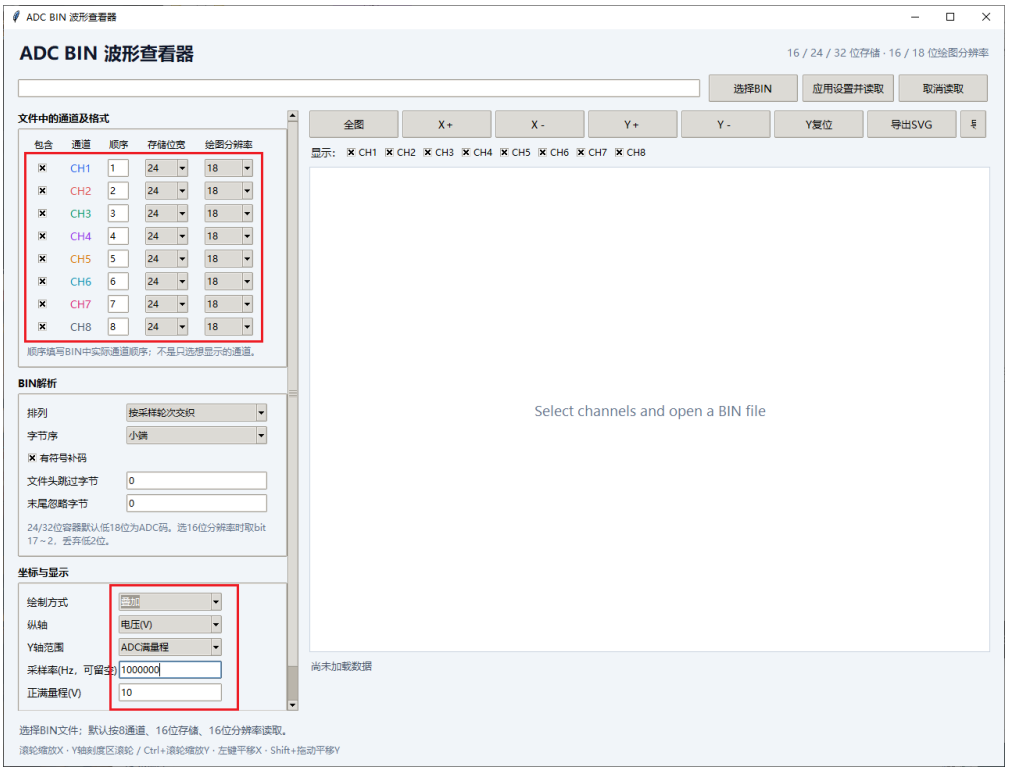

图 1-69 配置 ADC BIN 波形查看器

随后，选择对应的 BIN 文件，应用设置并读取，软件便会根据数据绘制出对应波形，如图 1-70 所示：

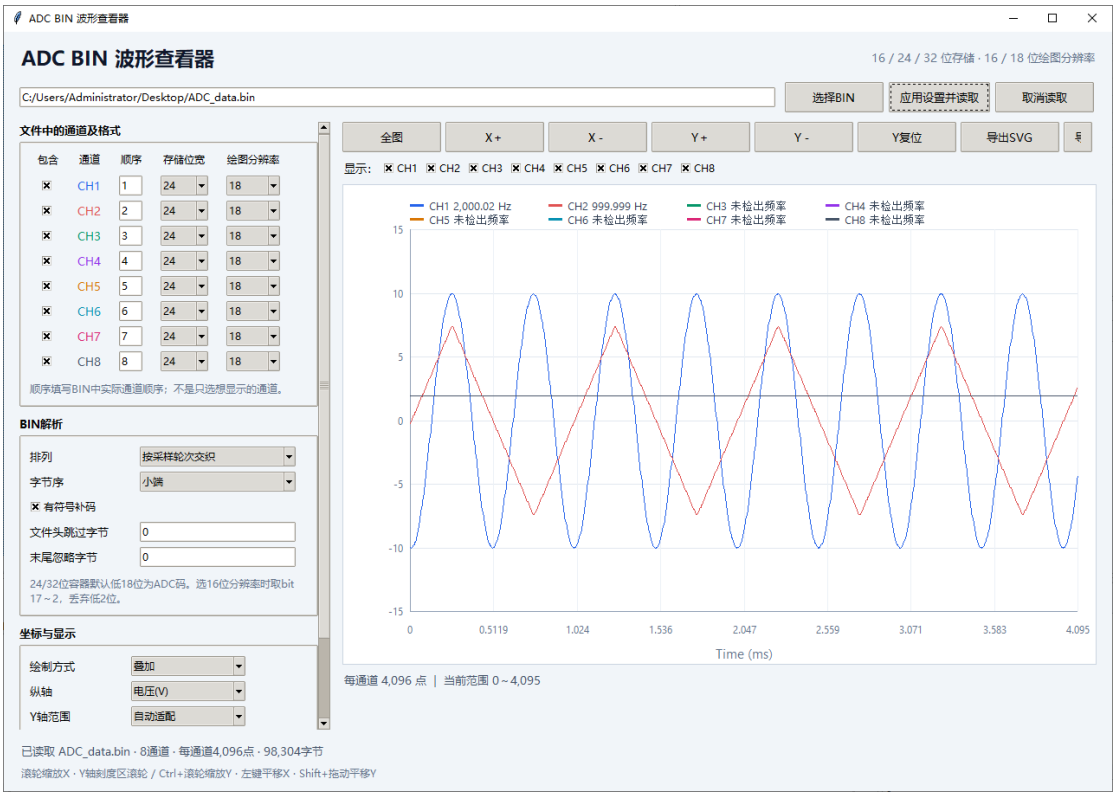

图 1-70 ADC 采样波形

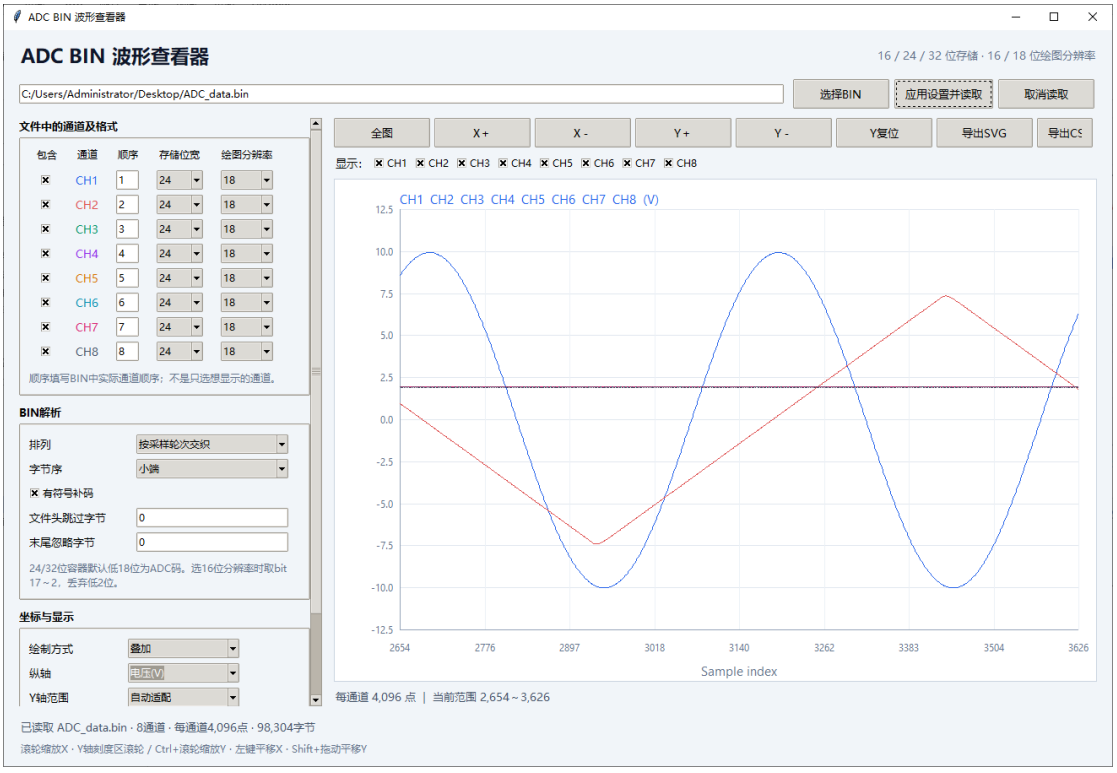

图 1-71 ADC 采样波形（局部放大）

可以看到，CH1 和 CH2 分别显示正弦波和锯齿波，与接入的信号对应。图 1-64 按采样点放大了局部波形，便于观察细节。其余通道由于未接入测试信号，所以得到的是悬空状态下的默认值。

1.3.5.2寄存器指令测试

前面的测试采用复位后的 ±10 V 量程。下面通过 LUT 将八路量程改为 ±5 V，带宽仍设为低带宽。配置前，先将信号发生的 CH1 调为 2 kHz、约 ±5 V 的正弦波，将 CH2 调为 1 kHz、约 ±3 V 的锯齿波，再新建接收文件或清空原文件，如下图所示：

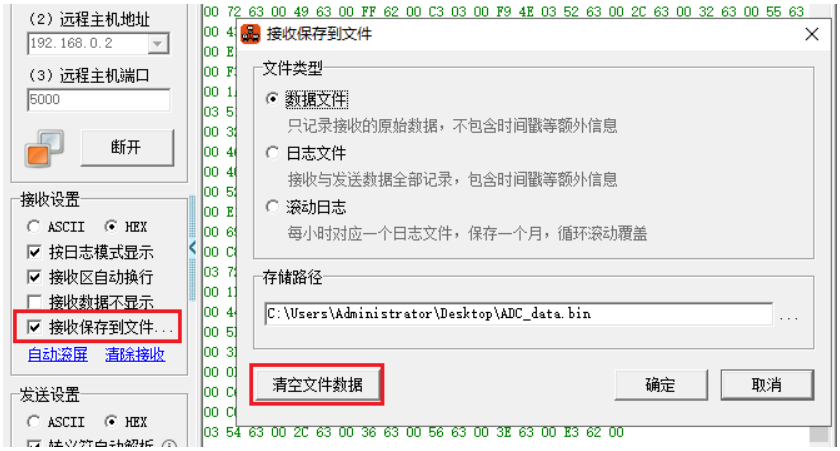

图 1-72 清空文件数据

随后，单独发送以下指令，启动 LUT 配置：

```
55 a5 04 00 00 00 01 f0
```

配置完成后，再发送与前面相同的四帧采集指令：

```
55 a5 01 00 00 00 ff f0 55 a5 02 00 01 80 00 f0 55 a5 03 00 00 00 63 f0 55 a5 00 00 00 00 01 f0
```

采集完成后，打开本轮保存的 BIN 文件。保留原有解析设置，将正满量程改为 5 V。为便于观察，可将 Y 轴范围设为“ADC 满量程”，然后点击“应用设置并读取”。结果如下：

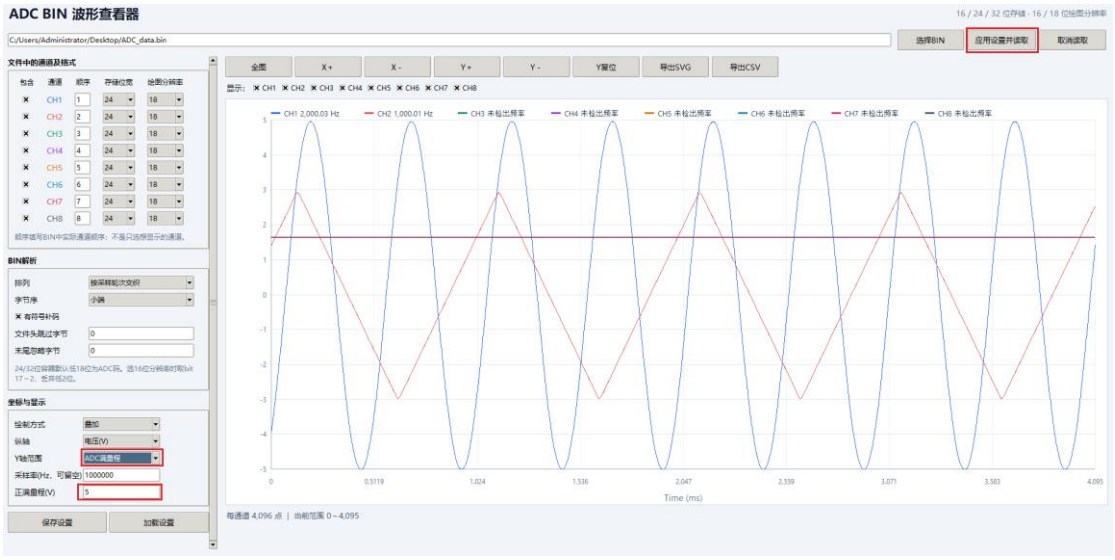

图 1-73 ADC 采集波形

按 ±5 V 量程换算后，两路波形与本次输入相符。接下来保持低带宽设置，将 CH1 的正弦波频率提高到 20 kHz，CH2 的锯齿波频率提高到 10 kHz，再进行一次采集。结果如图 1-74 所示：

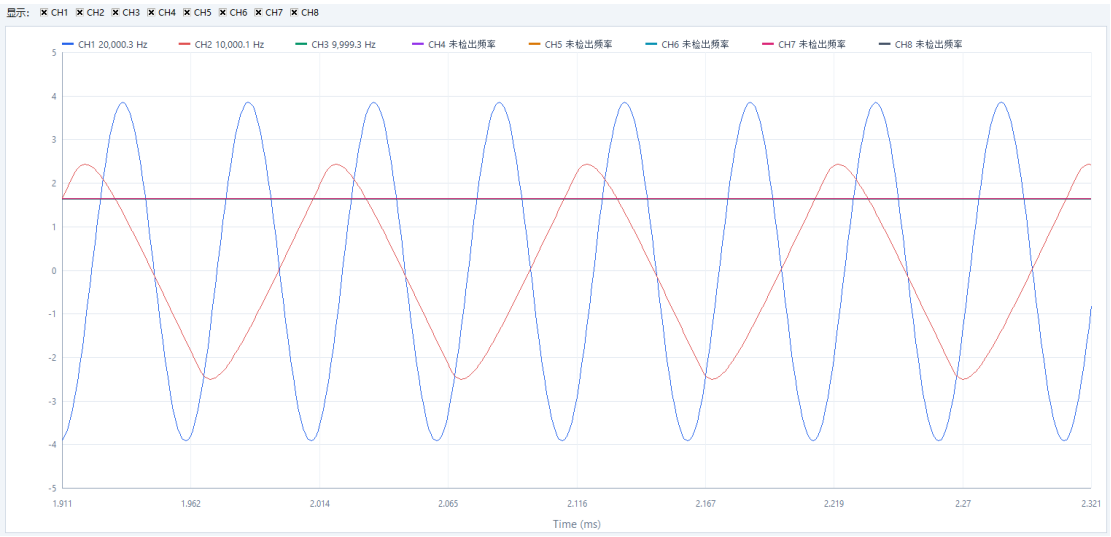

图 1-74 低带宽模式 20KHz，±5V 正弦波与 10KHz，±3V 锯齿波

输入频率提高后，可以看到两路波形的峰值都出现了一定程度的衰减。这时候，便可以通过寄存器指令使能高带宽模式来进行解决。AD7606C-18 手册中关于带宽寄存器的描述见图 1-75 所示：

Address: 0x07, Reset: 0x00, Name: BANDWIDTH

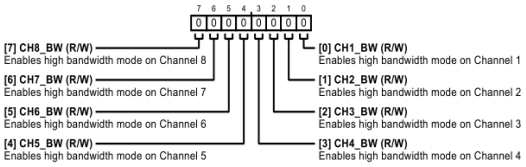

Table 38. Bit Descriptions for BANDWIDTH

Bits	Bit Name	Description	Reset	Access
7	CH8_BW	Enables high bandwidth mode on Channel 8.	0x0	R/W
6	CH7_BW	Enables high bandwidth mode on Channel 7.	0x0	R/W
5	CH6_BW	Enables high bandwidth mode on Channel 6.	0x0	R/W
4	CH5_BW	Enables high bandwidth mode on Channel 5.	0x0	R/W
3	CH4_BW	Enables high bandwidth mode on Channel 4.	0x0	R/W
2	CH3_BW	Enables high bandwidth mode on Channel 3.	0x0	R/W
1	CH2_BW	Enables high bandwidth mode on Channel 2.	0x0	R/W
0	CH1_BW	Enables high bandwidth mode on Channel 1.	0x0	R/W

图 1-75 AD7606C-18 带宽寄存器

由图可知，只需要向 0x07 寄存器中写入 0xFF，便可以将八个通道设为高带宽模式。对应的指令如下：

```
55 a5 04 07 ff 00 00 f0
```

单独发送带宽配置指令。配置完成后，换用新的接收文件，在不改变信号源的情况下再次采集。此时得到的波形如下：

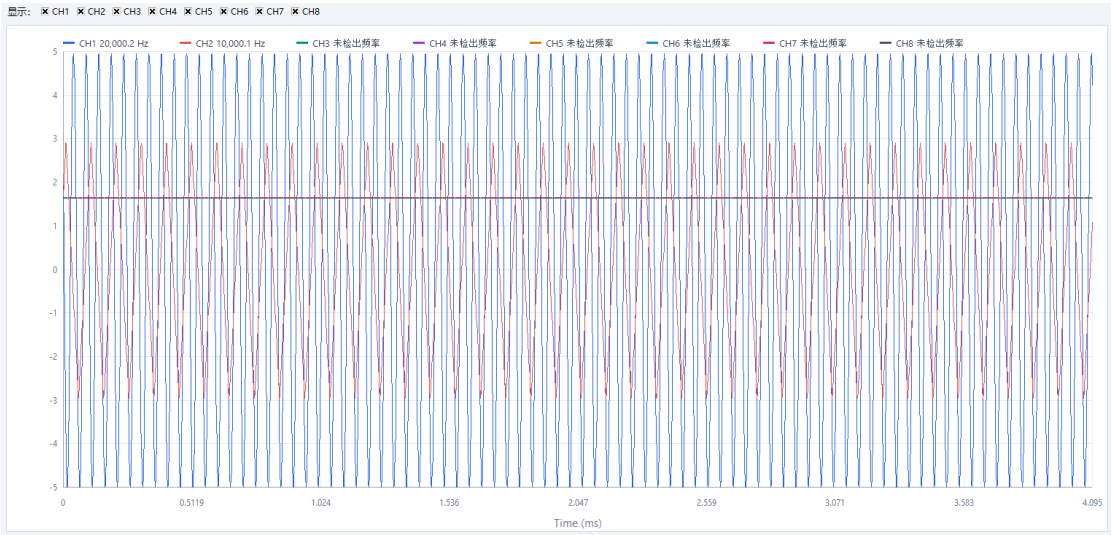

图 1-76 高带宽模式下采集波形

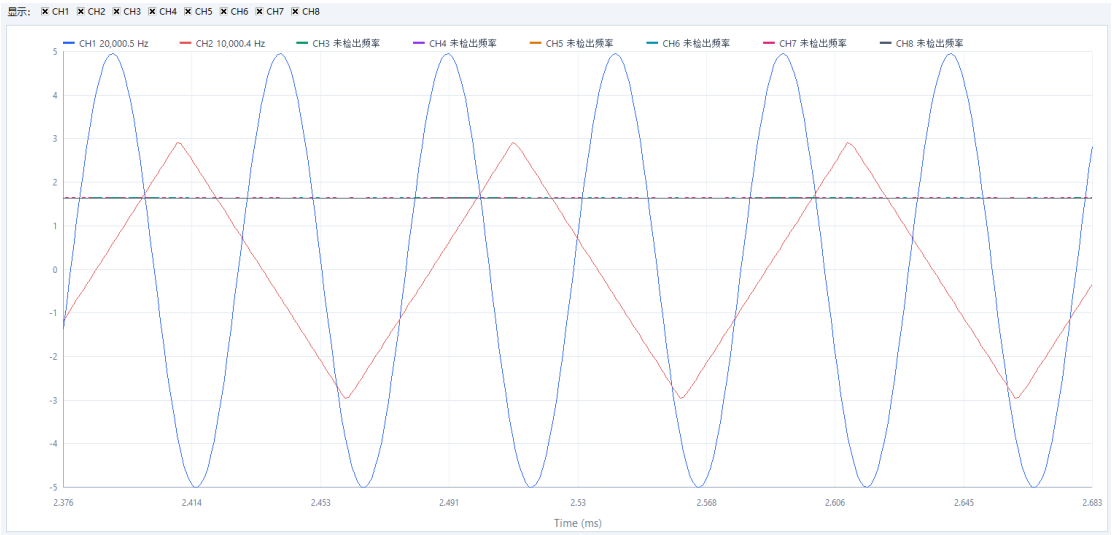

图 1-77 高带宽模式波形图（局部放大）

可以看到，在发送了单个寄存器的配置指令后，两路采样波形的峰峰值均有所增加，更加接近真实值，说明寄存器指令成功实现了高带宽模式的使能。再结合前面发送 Lut 配置指令的结果，也就验证了，无论是单寄存器配置，还是批量的 Lut 配置，设计都能达到预期的目标。

1.3.5.38 通道采样测试

前面的测试主要观察 CH1 和 CH2，下面进一步验证八个通道的采样结果。这里，为 CH1～CH8 分别接入从约 1 V 开始逐路升高的直流电压，按前面的方法采集并打开 BIN 文件。结果如下图 1-78 所示：

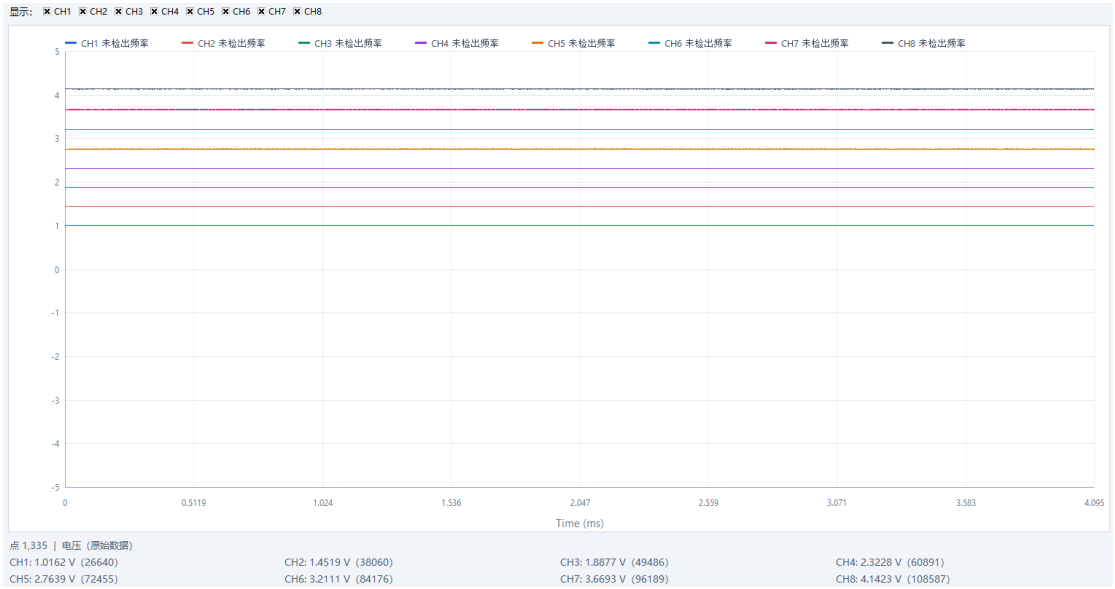

图 1-78 8 通道采样结果

可以看到，八路直流电平从 CH1 到 CH8 依次升高，与信号接入的顺序一致，说明本次采集的数据在回传和解码后仍能正确对应各输入通道。

至此，我们便完成了对设计功能的验证，无论是 ADC 的采集与回传、指令配置寄存器，还是八通道数据采样，都达到了预期的功能需求。而除了这些之外，设计同样也支持如采样通道、采样数据量、采样率的配置，并且当数据量过大时，还会自动进行 DMA 拆分，本章暂不做展开验证，感兴趣的读者可自行测试。

1.4　思考与总结

本次实验介绍了基于 ACM7606C 并口软件模式下的多通道数据采集以太网 TCP 收发设计，通过 AXI DMA+DDR+以太网 TCP+上位机的方式，实现了对 ADC 的采样控制、数据存储、波形显示等功能。借助这一方案，用户可以灵活、便捷地配置 ADC 运行模式，并直观地观察与分析采样结果。

设计中只是对 ACM7606C 以及以太网 TCP 为例进行讲解，实际使用时，用户也可以基于该架构，根据自身需求，修改 ADC 驱动以及通信端口，从而应用于更多的场景。

最后，这里再为大家补充一个可能遇到的问题：

（1） 无法修改 IP 为静态 IP 地址

出现这种情况，通常都是相关驱动缺少部分文件造成的，此时我们可以使用管理员权限打开 cmd，在其中输入以下代码来修改静态 IP：

```
netsh interface ip set address "以太网名" static 静态 IP 255.255.255.0
```

其中，红色部分为需要用户自己输入的部分，以太网名通常由“以太网”+“空格”+“数字”组成，用户输入时，注意不要忘记空格。

如果想将 IP 恢复到动态 IP 地址，只需要使用以下语句：

```
netsh interface ip set address name="以太网名" source=dhcp
```