1　基于 ACM7606 的多通道简易示波器

1.1 背景介绍

模数转换器即 A/D 转换器，或简称 ADC，通常是指一个将模拟信号转变为数字信号的电子元件。通常的模数转换器是将一个输入电压信号转换为一个输出的数字信号。由于数字信号本身不具有实际意义，仅仅表示一个相对大小，故任何一个模数转换器都需要一个参考模拟量作为转换的标准，比较常见的参考标准为最大的可转换信号大小。而输出的数字量则表示输入信号相对于参考信号的大小。

本次实验将基于 TFT LCD 屏显示实验一节的内容，结合 ADI 公司的 16 位 8 通道并行采样 ADC 芯片，完成多通道简易示波器的设计。除了波形显示，用户可以通过触摸屏对采样率、采样通道、触发模式等参数进行设置。

1.1.1 ACM7606 模块简介

ACM7606 数据采集模块使用的是 ADI 公司的 16 位 8 通道同步采样模数转换器 AD7606，模块图如下图 1-1 所示。

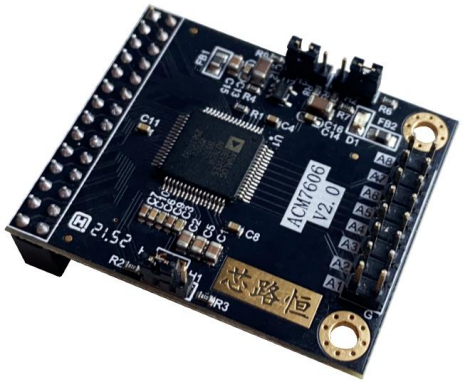

图 1-1ACM7606 模块图

AD7606 是 16 位 8 通道同步采样模数数据采集系统（DAS）。内置模拟输入箝位保护、二阶抗混叠滤波器、跟踪保持放大器、16 位电荷再分配逐次逼近型模数转换器（ADC）、灵活的数字滤波器、2.5V 基准电压源、基准电压缓冲以及高速串行和并行接口。AD7606 采用 5V 单电源供电，可以处理±10V 和±5V 真双极性输入信号。同时所有通道均能以高达 200kSPS 的吞吐速率采样。输入箝位保护电路可以耐受最高达±16.5V 的电压。无论以何种采样频率工作，其模拟输入阻抗均为 1MΩ。采用单电源工作方式，具有片内滤波和高输入阻抗，因此无需驱动运算放大器和外部双极性电源。AD7606 抗混叠滤波器的 3dB 截止频

率为 22kHz；当采样频率为 200Ksps 时，它具有 40dB 的抗混叠抑制特性。

芯片对外提供 SPI 和并行的数字接口。当 AD7606 的 8 个通道全部以 200KPS 的最高速率进行转换时，数据输出速率达到 25.6Mbps，需要使用高性能 MCU 的 SPI 外设才能勉强该速率要求。因此可以使用 16 位并口来进行数据的传输，提高数据传输速率。当 AD7606 应用在 FPGA 系统中的时候，使用 SPI 串行接口和并行接口都能够轻松的满足数据传输的速率需求。当在 FPGA 系统上应用 AD7606 时，可以通过在 FPGA 上设计 AD7606 控制转换逻辑，将转换结果数据直接存储到片上的存储器如 FIFO 或者 RAM 中，也可以存储到 FPGA 片外的存储器如 SRAM 或 SDRAM 中，然后由其他主控芯片如 MCU 或 DSP 读出，或者直接在 FPGA 内部进行数据的运算和处理。当然，由于 FPGA 片上可以设计软核控制器，也可以直接使用软核控制器完成数据的处理和传输工作。

1.1.1.1 功能框图

AD7606 的功能框图如下图 1-2 所示：

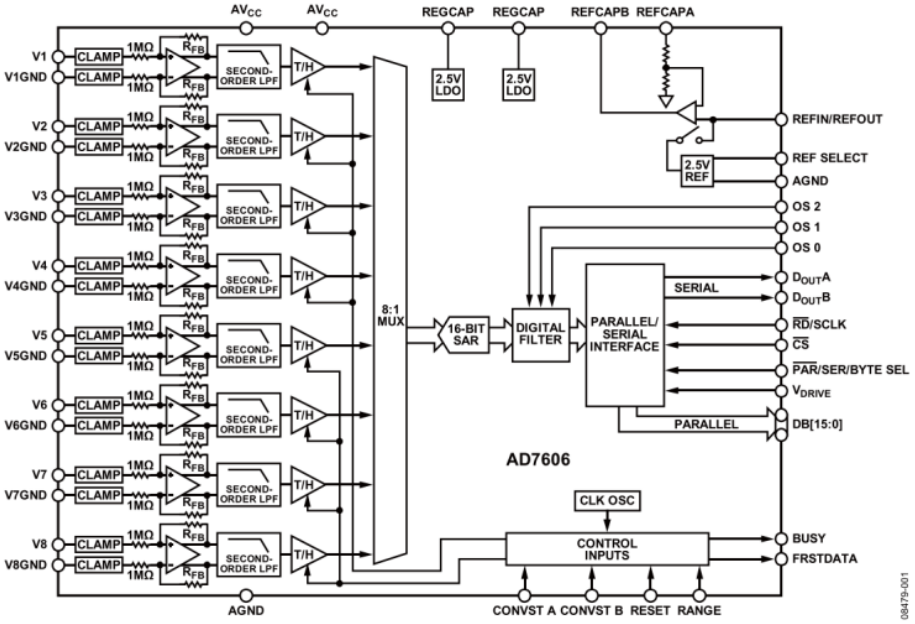

图 1-2 AD7606 功能框图

如上图所示，采集到的数据在经过稳压滤波和采样保持后通过 8 选 1 多路选择器被分别送入到 16 位逐次逼近型 ADC 芯片 AD7606 中进行转换，最后经由数字滤波后输出，当数据以串行模式输出时，数据会从 DoutA、DoutB 中输出，如果数据是以并行方式输出，那么数据将从 DB[15:0]中输出。

1.1.1.2 模拟输入

AD7606 可处理真双极性、单端输入电压。RANGE 引脚的逻辑电平决定所有模拟输入通道的模拟输入范围。如果此引脚与逻辑高电平相连，则所有通道的模拟输入范围为±10V。如果此引脚与逻辑低电平相连，则所有通道的模拟输入范围为±5V。AD7606 的模拟输入阻抗为 1MΩ。这是固定输入阻抗，不随 AD7606 采样频率而变化。AD7606 的输入结构如下图 1-3 所示，其各路模拟输入均含有箝位保护电路。虽然采用 5V 单电源供电，但此模拟输入箝位保护允许输入过压达到±16.5V。

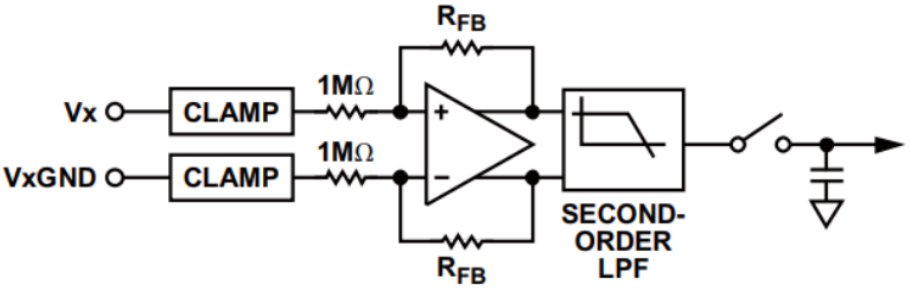

图 1-3 AD7606 模拟输入结构

1.1.1.3 数字滤波器与过采样

AD7606 内置一个可选的数字一阶 sinc 滤波器，在使用较低吞吐率或需要更高信噪比或更宽动态范围的应用中，应使用该滤波器。数字滤波器的过采样引脚 OS[2:0]控制（具体参考 AD7606 数据手册）。OS2 为 MSB 控制位。OS0 为 LSB 控制位，下表 1-1 提供了用来选择不同过采样倍率的过采样位解码。

表 1-1 不同过采样倍率的过采样位解码

OS[2:0]	过采样倍率	5V 范围 SNR(dB)	10V 范围 SNR(dB)	5V 范围 3dB 带宽(kHz)	10V 范围 3dB 带宽(kHz)	最大吞吐量 CONVST 频率(kHz)
000	No OS	89	90	15	22	200
001	2	91.2	92	15	22	100
010	4	92.6	93.6	13.7	18.5	50
011	8	94.2	95	10.3	11.9	25
100	16	95.5	96	6	6	12.5
101	32	96.4	96.7	3	3	6.25
110	64	96.9	97	1.5	1.5	3.125
111	无效					

OS 引脚在 BUSY 下降沿锁存，从而设置下一个转换的过采样倍率，如下图 1-4 所示，如果 OS 引脚选择过采样倍率 8，则下一个 CONVST x 上升沿采集各通道的第一个采样点，一个内部产生的采样信号采集所有通道的其余 7 个样点，然后对这些样点求平均值，以改进 SNR 性能。开启过采样时，CONVST A 和

CONVST B 引脚必须连在一起驱动，转换过程中 BUSY 保持高电平的时间会延长。BUSY 保持高电平的实际时间取决于所选的过采样倍率，过采样倍率越高，则 BUSY 保持高电平的时间或总转换时间越长。

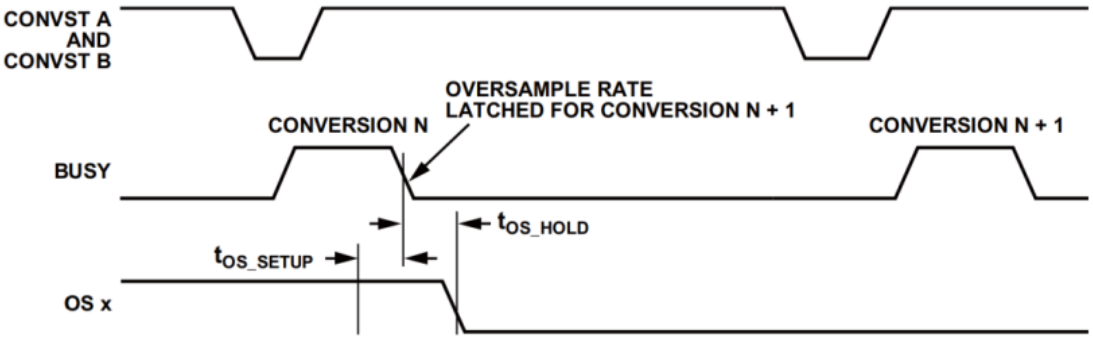

图 1-4 OS 引脚在 BUSY 下降沿锁存时序示意图

1.1.1.4 工作时序

AD7606 根据采样方式不同具有多种驱动时序，本次实验采用的为并行输出（即 8 个 16 位的数据通过 16 根并行线一个接着一个输出），转换后读取模式。其时序图由两部分组成：完成 AD 转换和读取 AD 数据。其中的时间可以参考 ADI 公司的手册。当 CONVST A 和 CONVST B 通道都变为上升沿时，BUSY 信号转变为高电平，代表转换开始，知道 BUSY 的下降沿到来，代表数据已经转换完成，正在锁存至输出数据寄存器中，当$\overline{CS}$变为下降沿时，数据将会被输送到总线上。并行工作模式下，当$\overline{CS}$和$\overline{RD}$都为低电平时，会使能总线，将转换结果输出到并行数据总线上，当 V1 转换结果开始输出之后，FRSTDATA 会随后转变为高电平，表示输出数据总线可以提供 V1 的结果。并行模式下，每次数据的输出为 16 位，对应一个通道。

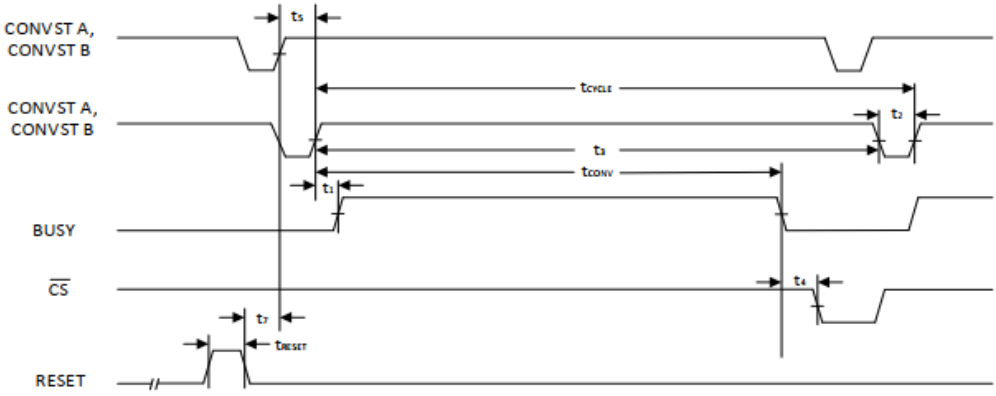

图 1-5 CONVST 时序—转换之后读取

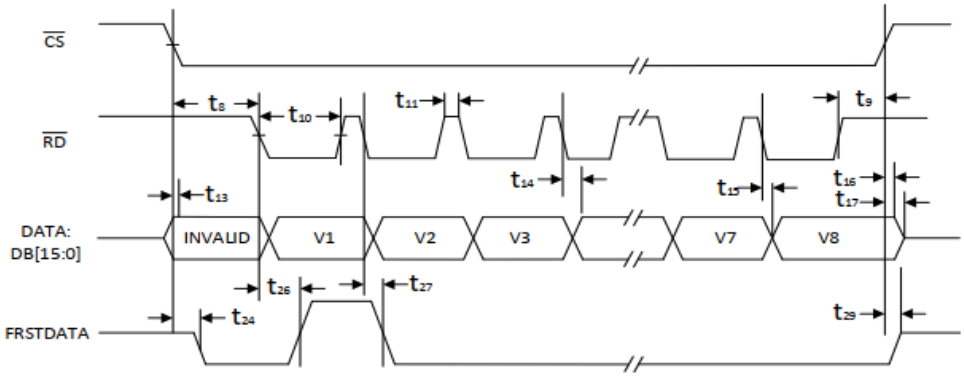

图 1-6 并行模式，独立的$\overline{CS}$和$\overline{RD}$脉冲

1.2 实验介绍

本次设计将会使用到独立模块 ACM7606，该模块通过开发板上 40pin 拓展接口连接，支持 8 通道电压数据采集。用户可以设置采集其中任一通道的数据，自定义的 IP 核会根据数据计算出其最大值、最小值、中间值等特殊值，ADC 采集数据会存储进 RAM 中，最终在触摸屏中显示。用户可以通过触摸屏中的按键设置不同的触发模式、采样频率、采样通道、触发电压等。

1.3 建立 HQFPGA 工程

将 TFT LCD 屏显示实验的工程复制至存放本次实验的文件夹之下，并且对其进行修改，修改方式参考实验二的 2.3 节中的内容，工程名修改为“xist_acm7606_scope”。

1.4 FPGA 侧代码设计

本次实验，FPGA 侧代码的基本框架如下图 1-7 所示。

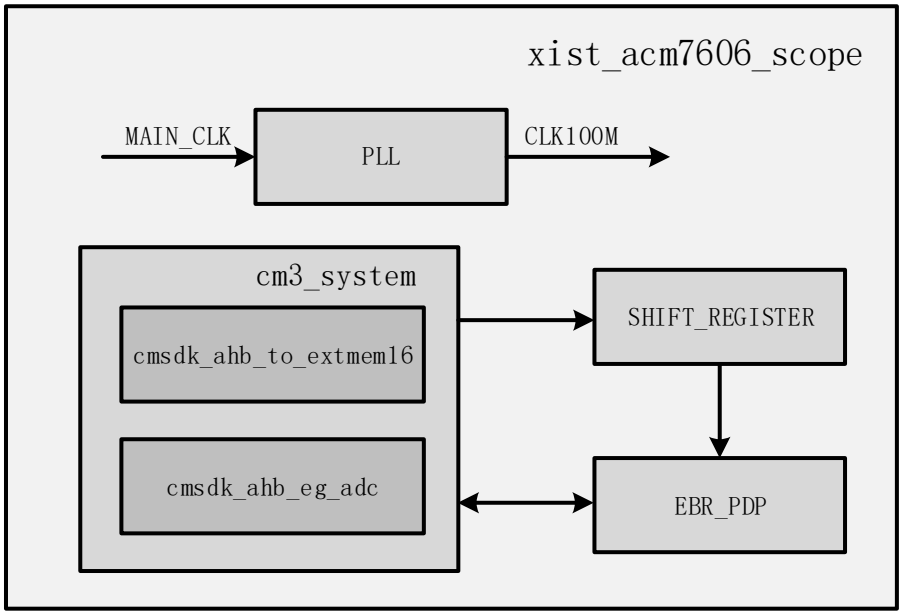

图 1-7 FPGA 侧代码基本框架图

下面将对各个模块进行简要说明。

1. PLL 模块：PLL IP 模块，输入时钟 MAIN_CLK，由开发板上的晶振提供，为 25M 的时钟，输出 100M 的时钟给到其他模块使用。
2. cm3_system 模块：CM3 微控制器模块，用来初始化 CM3 内核。
3. cmsdk_ahb_to_extmem16 模块：用于驱动 TFT 屏，该模块的讲解请参看实验七“TFT LCD 屏显示实验”一节的内容。
4. cmsdk_ahb_eg_adc 模块：本次实验需要新增加的一个自定义 IP，通过该 IP 实现对 ACM7606 的驱动以及设计多个模块实现对触发模式、采样频率、采样通道、触发电压的输出或者控制，在实验十“在 FPGA 侧为 CPU 添加数码管自定义 IP 实验”一节中，我们讲解了如何自定义 IP，这里，将其工程目录下的“cmsdk_ahb_eg_slave”文件夹复制至本工程中，修改文件名为“cmsdk_ahb_eg_adc”，方便我们区分，然后将该文件下的文件添加至工程中，添加文件的方式参看实验一，然后根据本次实验，对其进行修改，使其实现对 ACM7606 的控制，并且输出各项参数给到 CPU 进行处理。
5. SHIFT_REGISTER 模块：移位寄存器 IP，将 ACM7606 输入的 16 位数据移位 256 个时钟周期进行输出，这样做的目的是在触发之后示波器能够显示在触发之前的数据，该模块的配置界面如下图 1-8 所示。

图 1-8 SHIFT_REGISTER 模块配置界面示意图

6. EBR_PDP 模块：RAM IP 核，用来存储 ACM7606 采集到的数据，其读写位宽都为 16，地址深度设置为 65536，配置界面如下图 1-9 所示。

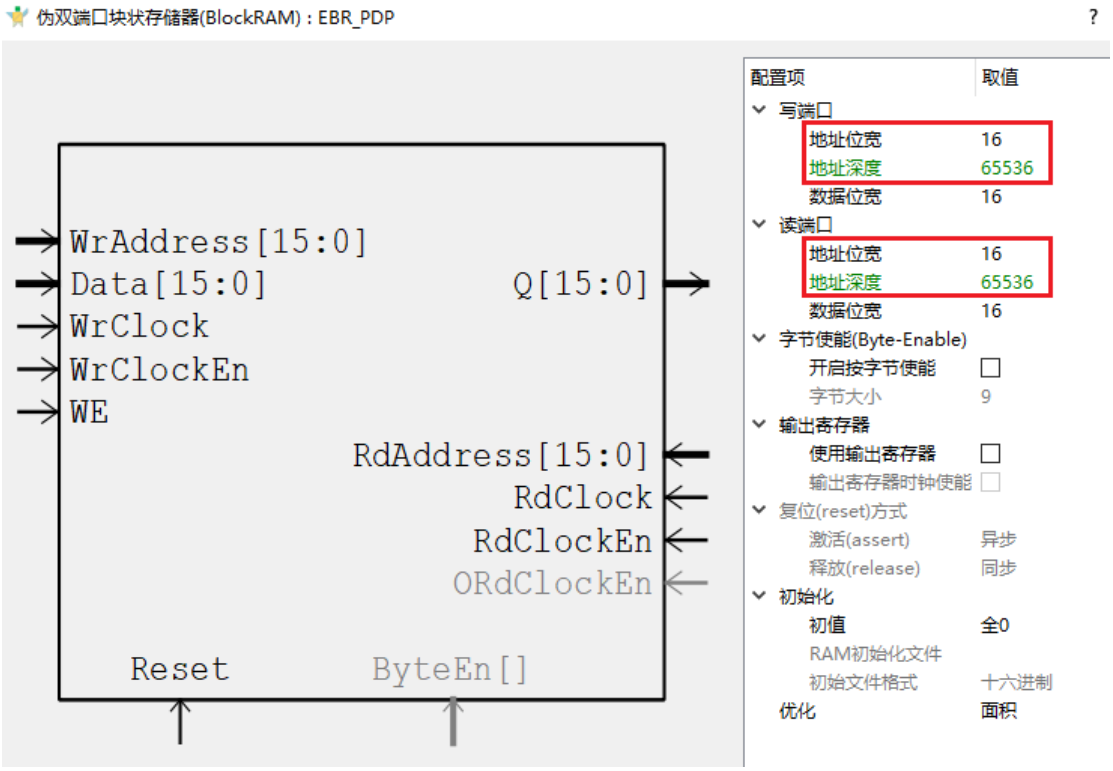

图 1-9 RAM IP 界面配置图

本次实验需要我们自己重新设计模块只有 cmsdk_ahb_eg_adc 模块，下面将对该模块的实现进行讲解，该模块的基本框架如下图 1-10 所示。

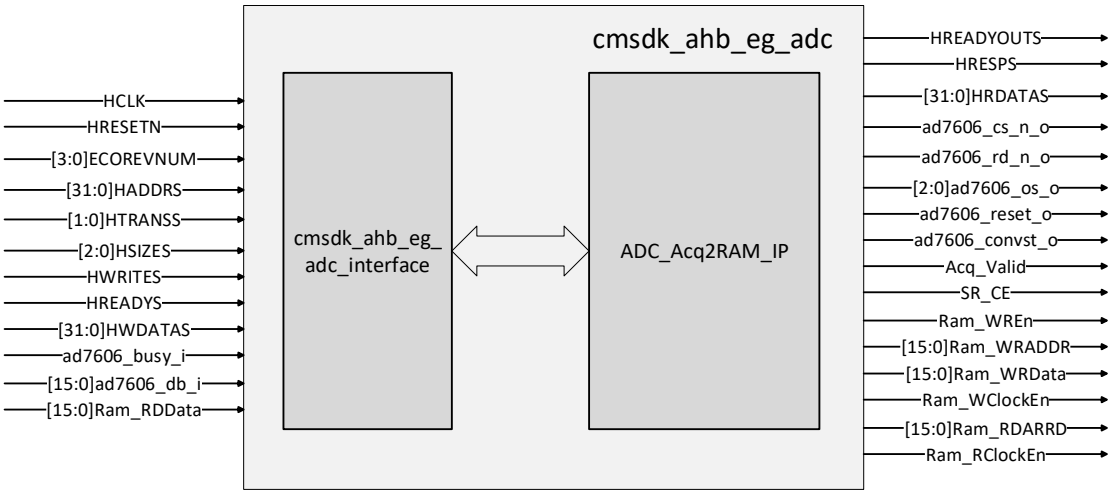

图 1-10 cmsdk_ahb_eg_adc 模块基本框图

上图中的 cmsdk_ahb_eg_adc_interface 模块在实验九中已经作过介绍，这里将不再进行赘述了，本章主要讲解 ADC_Acq2RAM_IP 模块。

1.4.1 ADC_Acq2RAM_IP

ADC_Acq2RAM_IP 模块主要实现对 ACM7606 的驱动以及设计多个模块实现对触发模式、采样频率、采样通道、触发电压的输出或者控制，该模块的基本框图如下图 1-11 所示。

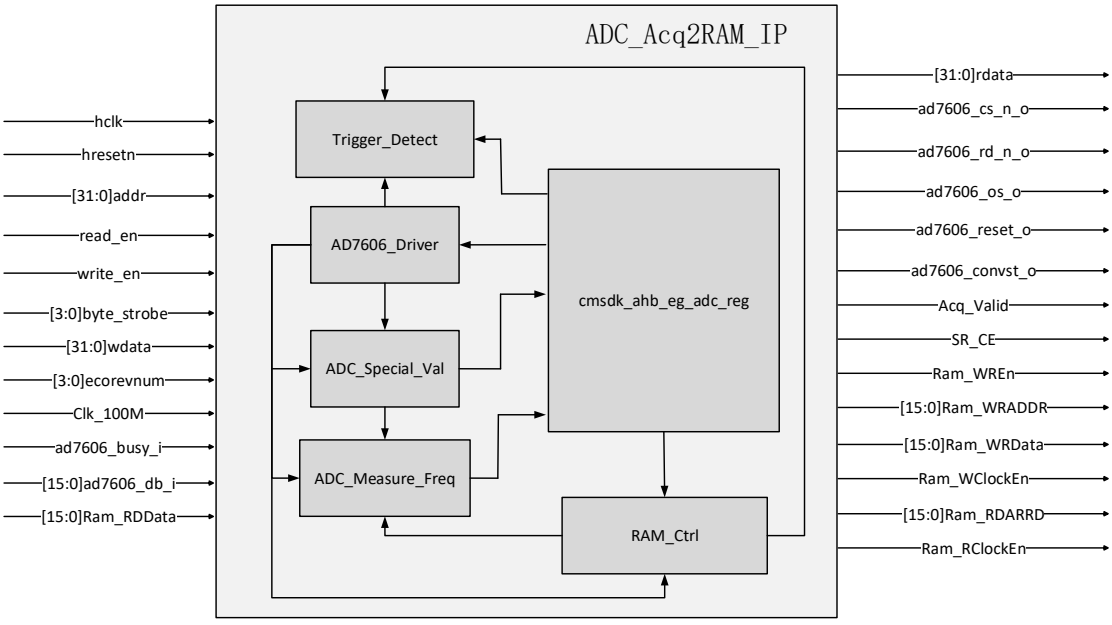

图 1-11 ADC_Acq2RAM_IP 模块基本框图

下面将对该 IP 中使用到的模块进行说明。

1.4.1.1 cmsdk_ahb_eg_adc_reg

cmsdk_ahb_eg_adc_reg 模块将 cmsdk_ahb_eg_adc_interface 输出的信号进行

处理，得到控制 AD7606 采样通道、采样速率、启动采样的信号以及将采样的信号频率、电压值等进行输出，最终交由 CM3 进行处理，该模块的基本结构如下图 1-12 所示。

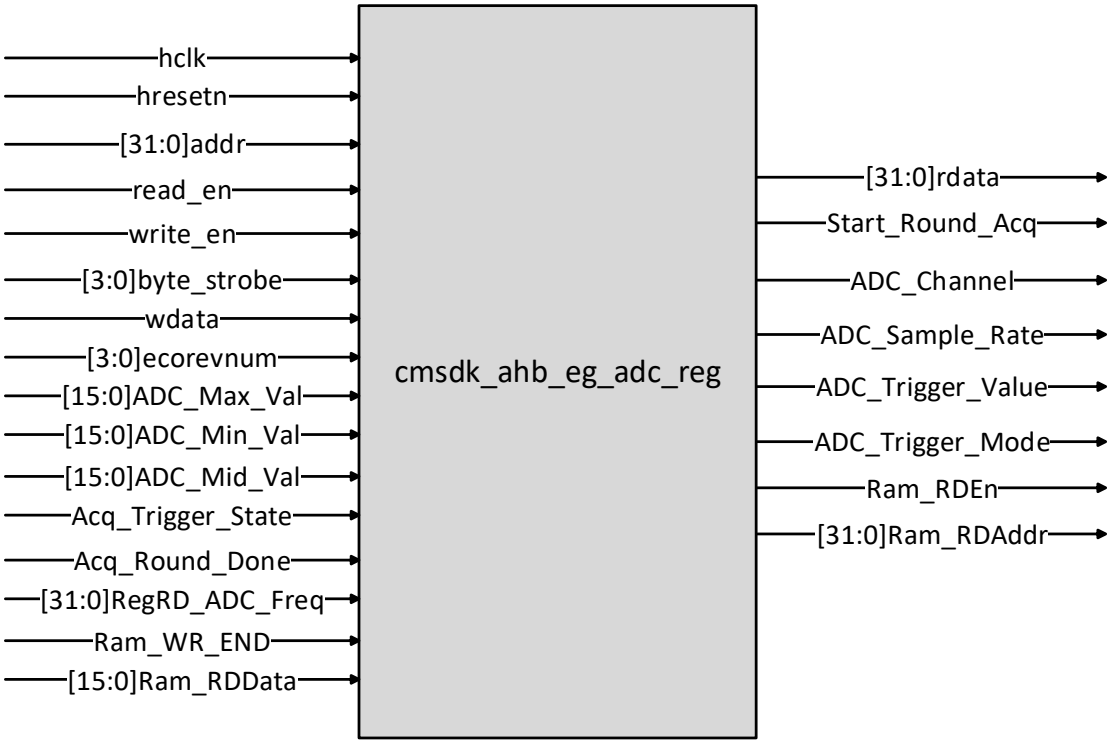

图 1-12 cmsdk_ahb_eg_adc_reg 模块信号基本结构图

对于该模块的信号说明如下所示。

表 1-2 cmsdk_ahb_eg_adc_reg 模块信号说明表

信号名称	I / O	信号意义
hclk	I	为所有总线传输提供基准频率，所有信号的时序都和 HCLK 的上升沿有关
hresetn	I	总线复位信号，低电平有效，用于复位系统和总线。
addr[31:0]	I	地址信号的设定，32 位的地址信号
read_en	I	总线的读使能
write_en	I	总线的写使能
byte_strobe[3:0]	I	传输数据字节位数的选择，由 hsizes 信号和 haddrs 的低 2 位共同控制
wdata[31:0]	I	写数据总线，数据从主机传送至从机
ecorevnum[3:0]	I	工程变更顺序修正位
ADC_Max_Val[15:0]	I	AD7606 采集信号电压的最大值
ADC_Min_Val[15:0]	I	AD7606 采集信号电压的最小值
ADC_Mid_Val[15:0]	I	AD7606 采集信号电压的中值
Acq_Trigger_State	I	AD7606 触发状态控制信号
Acq_Round_Done	I	AD7606 一轮采集完成信号
RegRD_ADC_Freq	I	AD7606 采集信号频率值
Ram_WR_END	I	RAM 的写完成信号

Ram_RDData[15:0]	I	RAM 的读数据信号
rdata[31:0]	O	读数据总线，数据从从机传输至主机
Start_Round_Acq	O	AD7606 开始一轮开始标志信号
ADC_Channel[2:0]	O	AD7606 通道选择信号
ADC_Sample_Rate[10:0]	O	AD7606 采样分频值
ADC_Trigger_Value[15:0]	O	AD7606 触发有效信号
ADC_Trigger_Mode[1:0]	O	AD7606 触发模式信号
Ram_RDEn	O	RAM 的读使能信号
Ram_RDAddr[15:0]	O	RAM 的读地址的信号

在该模块中根据需求，设置相关寄存器用于驱动其他模块开始工作，并将采集信号的电压值、频率等传输给 CM3 处理，相关寄存器说明如表 1-3。

表 1-3 寄存器配置表说明

类	寄存器名称	寄存器偏移地址	寄存器说明
ADC 相关寄存器	slv_reg0	0x20	接收从 CM3 传输过来的控制信号，其中[2:0]代表 AD7606 通道选择信号 ADC_Channel，[13:3]代表 AD7606 采样分频信号 ADC_Sample_Rate，[29:14]代表 AD7606 触发有效信号 ADC_Trigger_Value，[31:30]代表 AD7606 触发模式信号 ADC_Trigger_Mode。
	slv_reg1	0x24	将 AD7606 采集信号的最大值和最小值传输给 CM3，其中[15:0]代表采集信号电压的最小值 ADC_Min_Val，[31:16]代表采集信号电压的最大值 ADC_Max_Val。
	slv_reg2	0x28	将 AD7606 采集信号的中值、采样触发状态、一次采样完成信号传输给 CM3，其中[15:0]代表 AD7606 采集信号的中值 ADC_Mid_Val，[17:16]代表采样触发状态信号 Acq_Trigger_State，[19:18]代表一次采样完成信号 Acq_Round_Done。
	slv_reg3	0x2C	将 AD7606 采样信号的频率值传输给 CM3
RAM 相关寄存器	ram_reg0	0x18	CM3 控制的 RAM 的读地址信号，[15:0]代表 RAM 的读地址信号，高 16 位暂时未用到。
	ram_reg1	0x1C	RAM 输出的对应地址的 16 位数据，[15:0]代表从 RAM 读到对应数据，高 16 位暂时未用到，当读该寄存器的时候，同时产生 RAM 的读使能信号 Ram_RDEn。
	ram_reg2	0x30	输出给 CM3 的 RAM 一次写数据完成信号，产生该信号之后，方可开始读取数据

根据上表中的寄存器说明，接下来就需要通过代码实现这些寄存器需要实现的功能。

1. 根据地址信号 addr 以及写使能信号 write_en，得到相关寄存器写选择信号 wr_sel，根据表 1-3 中内容，定义相关寄存器信号如下所示。

```
wire[6:0]   wr_sel;
// Address decoding for write operations
```

```
assign wr_sel[0] = ((addr[11:2]==10'b0000001000)&(write_en)) ? 1'b1: 1'b0;  //寄存器 0：0x20
assign wr_sel[1] = ((addr[11:2]==10'b0000001001)&(write_en)) ? 1'b1: 1'b0;  //寄存器 1：0x24
assign wr_sel[2] = ((addr[11:2]==10'b0000001010)&(write_en)) ? 1'b1: 1'b0;  //寄存器 2：0x28
assign wr_sel[3] = ((addr[11:2]==10'b0000001011)&(write_en)) ? 1'b1: 1'b0;  //寄存器 3：0x2C
assign wr_sel[4] = ((addr[11:2]==10'b0000000110)&(write_en)) ? 1'b1: 1'b0;  //寄存器 4：0x18
assign wr_sel[5] = ((addr[11:2]==10'b0000000111)&(write_en)) ? 1'b1: 1'b0;  //寄存器 4：0x1C
assign wr_sel[6] = ((addr[11:2]==10'b0000001100)&(write_en)) ? 1'b1: 1'b0;  //寄存器 4：0x30
```

2. 根据 wr_sel 信号，进行写寄存器的操作，根据字节传输位数选择信号 byte_strobe，对应不同的位写入至寄存器中，代码如下所示：

```
always @(posedge hclk or negedge hresetn)
begin
if (~hresetn)
    begin
    slv_reg0 <= {32{1'b0}}; // reset data register to 0x00000000
    slv_reg1 <= {32{1'b0}};
    slv_reg2 <= {32{1'b0}};
    slv_reg3 <= {32{1'b0}};
    ram_reg0 <= {32{1'b0}};
    end
else if (wr_sel[0])//寄存器 0
    begin
    if (byte_strobe[0])
        slv_reg0[ 7: 0] <= wdata[ 7: 0];
    if (byte_strobe[1])
        slv_reg0[15: 8] <= wdata[15: 8];
    if (byte_strobe[2])
        slv_reg0[23:16] <= wdata[23:16];
    if (byte_strobe[3])
        slv_reg0[31:24] <= wdata[31:24];
    end
else if (wr_sel[1])//寄存器 1
    begin
    if (byte_strobe[0])
        slv_reg1[ 7: 0] <= wdata[ 7: 0];
    if (byte_strobe[1])
        slv_reg1[15: 8] <= wdata[15: 8];
    if (byte_strobe[2])
        slv_reg1[23:16] <= wdata[23:16];
    if (byte_strobe[3])
        slv_reg1[31:24] <= wdata[31:24];
    end
else if (wr_sel[2])//寄存器 2
    begin
    if (byte_strobe[0])
        slv_reg2[ 7: 0] <= wdata[ 7: 0];
    if (byte_strobe[1])
        slv_reg2[15: 8] <= wdata[15: 8];
```

```
    if (byte_strobe[2])
        slv_reg2[23:16] <= wdata[23:16];
    if (byte_strobe[3])
        slv_reg2[31:24] <= wdata[31:24];
    end
else if (wr_sel[3])//寄存器 3
    begin
    if (byte_strobe[0])
        slv_reg3[ 7: 0] <= wdata[ 7: 0];
    if (byte_strobe[1])
        slv_reg3[15: 8] <= wdata[15: 8];
    if (byte_strobe[2])
        slv_reg3[23:16] <= wdata[23:16];
    if (byte_strobe[3])
        slv_reg3[31:24] <= wdata[31:24];
    end
else if(wr_sel[4]) //RAM 的读
    begin
    if (byte_strobe[0])
        ram_reg0[ 7: 0] <= wdata[ 7: 0];
    if (byte_strobe[1])
        ram_reg0[15: 8] <= wdata[15: 8];
    if (byte_strobe[2])
        ram_reg0[23:16] <= wdata[23:16];
    if (byte_strobe[3])
        ram_reg0[31:24] <= wdata[31:24];
    end
end
```

3. 进行寄存器的读操作，当产生 read_en 信号之后，根据 addr 信号不同，按照表 1-3 中的内容，将得到相关信号值赋值给 rdata，最终通过 AHB 总线传输给 CM3 进行处理，代码如下所示。

```
//读寄存器操作
always @ (read_en or addr or slv_reg0 or slv_reg1 or slv_reg2 or slv_reg3 or
ram_reg0 or ram_reg1)
 begin
   case (read_en)
     1'b1:
     begin
       if (addr[11:6] == 8'h00) begin
         case(addr[5:2])
           4'b1000:rdata=slv_reg0;    //0xA0000000
           4'b1001:rdata={ADC_Max_Val,ADC_Min_Val};   //0xA0000004
           4'b1010:rdata={Acq_Round_Done,Acq_Trigger_State,ADC_Mid_Val};
           4'b1011:rdata=RegRD_ADC_Freq;
           4'b0110:rdata=ram_reg0;
           4'b0111: begin rdata =  Ram_RDData; Ram_RDEn = 1'd1; end
```

```
            4'b1100: rdata = Ram_WR_END;
            default: rdata = {32{1'bx}};
          endcase
        end
        else begin
          rdata = {32'h00000000}; // default
        end
      end
    1'b0:
      begin
        rdata =  {32{1'b0}};
      end
    default:
      begin
        rdata =  {32{1'bx}};
      end
    endcase
      end
```

4. 根据表 1-3 中的内容，将写入寄存器 slv_reg0 和 ram_reg0 的值分配给不同的控制信号，代码如下所示。

```
 //ADC 模块控制信号
assign ADC_Channel = slv_reg0[2:0];
assign ADC_Sample_Rate = slv_reg0[13:3];
assign ADC_Trigger_Value = slv_reg0[29:14];
assign ADC_Trigger_Mode = slv_reg0[31:30];
//RAM 控制信号
assign Ram_RDAddr = ram_reg0[15:0];
```

1.4.1.2 AD7606_Driver

AD7606 控制器驱动模块 ad7606_driver，该控制器实现了对 AD7606 型 8 通道 16 位 ADC 的数据转换控制并输出。使用该控制器时，用户无需关心 AD7606 的具体控制时序，一切都在控制器内部完成，用户只需要像使用并行 ADC 一样取用数据即可，该模块的基本框图如下图 1-13 所示。

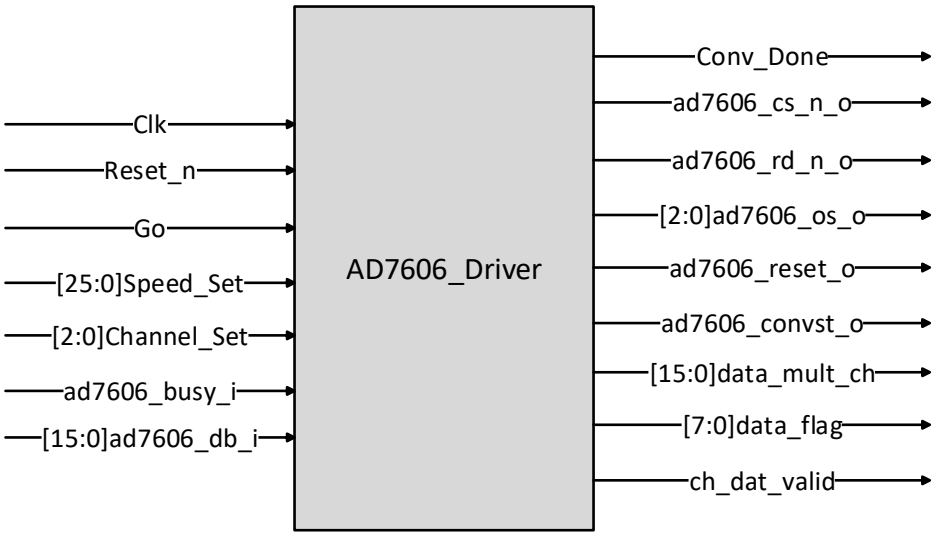

图 1-13 ad7606_driver 模块的基本框图

对该模块的信号说明如下表 1-4 所示。

表 1-4 ad7606_driver 模块信号说明表

信号名称	I/O	信号意义
Clk	I	时钟信号，为了让采样速率准确，要求为 100MHz
Reset_n	I	复位，低电平复位
Go	I	采样使能信号，为高电平就使能采样，低电平则在已经开始的一轮采样结束后，停止下一次采样。
Speed_Set[25:0]	I	采样速率控制端口，Speed_Set = 100000000/speed - 1
Channel_Set[2:0]	I	采样通道设置，采集从 0 通道到设定通道 Channel_Set 的值
ad7606_busy_i	I	ad7606 转换状态标志信号，为高电平则表明 ad7606 当前仍处于转换状态，结果没有更新，如果此时读取，读取的结果就还是前一次的采样转换结果。需要待该信号变为低电平之后，再读取 ad7606 中的数据
ad7606_db_i[15:0]	I	ad7606 的 16 位数据线，读取时，输出对应通道的转换结果
Conv_Done	O	一次采样完成标志信号，单时钟周期脉冲信号。每次 8 个通道结果都输出后，产生一个高脉冲信号。
ad7606_cs_n_o	O	ad7606 芯片选中控制信号，可以从 AD7606 中读取转换结果时，需要使该信号为低电平
ad7606_rd_n_o	O	ad7606 转换结果读取信号，该信号的下降沿，AD7606 将特定通道的采样结果送到 16 位数据线上，供外部读取。外部可以在 rd_n 信号的上升沿读取 16 位数据线上的结果
ad7606_os_o[2:0]	O	ad7606 过采样控制信号，使用过采样可以进一步提高 ad7606 的采样精度，使用过采样会降低 ad7606 的有效转换速率，关于过采样的功能和使用方法，可以参考 ad7606 的 datasheet，默认为 0，则表示不使用过采样。能够运行在最高的转换速率（200Ksps）
ad7606_reset_o	O	ad7606 的复位信号，复位 ad7606 内部各个功能单元的工作状态
ad7606_convst_o	O	ad7606 转换开始信号，该信号的上升沿启动 ad7606 内部的采样转换逻辑开始新一轮的采样转换
data_mult_ch[15:0]	O	多通道数据输出端口，该通道 16 位，在不同的时刻，输出不同通道的转换结果，使用时，与 data_flag 信号配合，data_flag 的哪一位出现高脉

		冲，则代表当前 data_mult_ch 的值为该通道的转换结果。该端口设计的目的是用于往 FIFO、RAM 等存储器中存储结果时使用。
data_flag[7:0]	O	转换结果有效标志信号，因为 AD7606 有 8 个通道，转换结果是依次输出，并非同时的，所以设置 8 个 Flag 信号，每个通道的结果就绪之后，就产生一个 Flag 信号，通知外部可以取用。另外，如果只关心其中的部分通道， 则只需要关心 data_flag 中对应的位即可
ch_dat_valid	O	输出通道有效信号

该控制器在工作时会根据主机的指令对采样频率进行修改，当信号转换完成后便对 ADC 写控制器发出 data_flag 信号，控制其将转换完成数据 data_mult_ch 写入 FIFO 或者 RAM 的同时，也指出了该信号来自哪个通道。通过这两个信号，我们可以实现让 ADC 以特定的采样速率多次采样一个或多个通道的数据。每当 data_flag 中任意一位为 1，则将 data_mult_ch 中的值写入 FIFO 或者 RAM 中。由于 FIFO 和 RAM 等存储器，只有一个数据输入接口，所以这个时候用一个 data_mult_ch 端口分时输出不同通道的采样结果，就比使用 data1~data8 这 8 个 16 位的数据端口分别输出各自通道的采样结果要方便。

例如，将通道 1、2、5、8 的采样结果存入 FIFO。就可以使用下面的写法：

```
module adc_wr_fifo(
  input Clk,
  input Reset_n,
  input [7:0]data_flag;
  input [15:0]data_mult_ch;
  output reg fifo_wrreq,
  output reg [15:0]fifo_data
);
always@(posedge Clk or negedge Reset_n)
if(!Reset_n)begin
  fifo_wrreq <= 0;
  fifo_data <= 0;
else if(data_flag == 8'b1001_0011)begin
  fifo_wrreq <= 1'd1;
  fifo_data <= data_mult_ch;
end
else begin
  fifo_wrreq <= 0;
  fifo_data <= fifo_data;
end
endmodule
```

1.4.1.3 ADC_Special_Val

ADC_Special_Val 模块用于得到 ADC 采集电压的最大、最小和中值，该模块的基本结构如下图 1-14 所示。

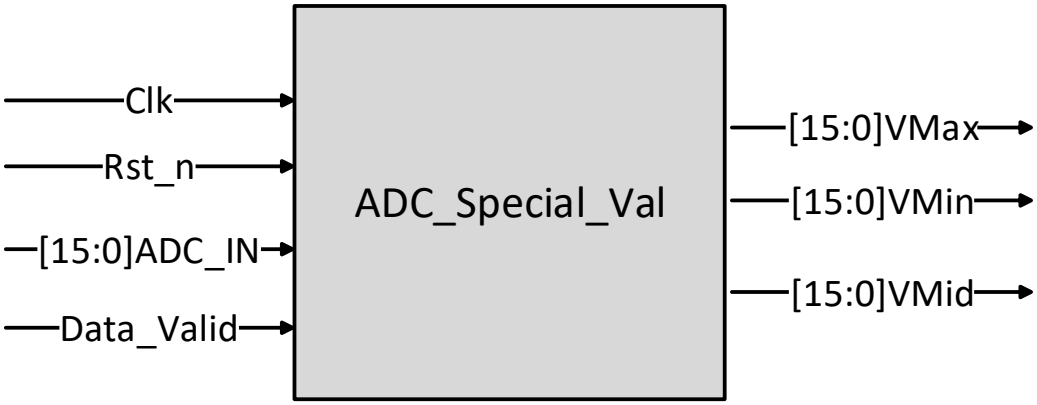

图 1-14 ADC_Special_Val 模块的基本框图

该模块的信号说明如表 1-5 所示。

表 1-5 ADC_Special_Val 模块的信号说明表

信号名称	I/O	信号意义
Clk	I	模块的时钟信号 100M
Rst_n	I	模块的复位信号
ADC_IN[15:0]	I	ADC 采集得到的 16 位数据
Data_Valid	I	ADC 采集数据有效信号
VMaxt[15:0]	O	采集信号的最大值
VMin[15:0]	O	采集信号的最下值
VMid [15:0]	O	采集信号的中值

在获取 AD7606 采集电压的特殊值时，首先需要知道 AD7606 的输出编码方式为二进制补码的方式，也就是说用 0/1 来区分一个数的符号，为 0 则代表正数，为 1 代表复数，其电压对应的二进制值如下图 1-15 所示。

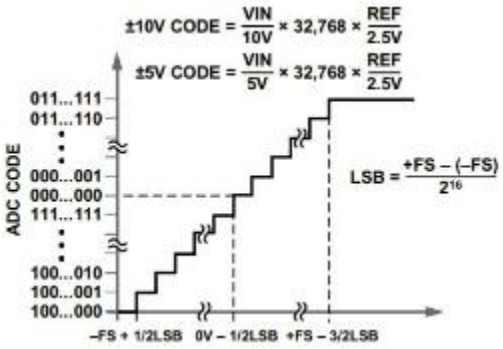

图 1-15 电压对应二进制数值图

从上图可以看出，AD7606 采集的数值最高位对应的就是符号位，我们在获取其采集电压的最大、最下值时，需要根据是正数还是复数进行判断。

首先我们定义一个计数器，计时达到 1S 时，重新开始计数，代码如下所示：

```
parameter UPDATERATE = 99999999;//计时一秒
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
```

```
    counter <= 0;
else if(counter >= UPDATERATE)
    counter <= 0;
else
    counter <= counter + 1'b1;
```

然后需要得到采集电压的最大值，当计数达到 1S 时，使电压最大值 VMax_r 为-32767（1111 1111 1111 1111）；当输入的电压 ADC_IN 最高位 ADC_IN[15]小于 VMax_r 最高位 VMax_r[15]并且数据有效时，代表此时输入的 ADC_IN 为正数，VMax_r 为负数，最大值 VMax_r 为 ADC_IN；当输入电压 ADC_IN 最高位 ADC_IN[15]等于 VMax_r 最高位 VMax_r[15]时，代表此时两个数都是正数或者复数，这时只需要比较两个值的大小，大的那个对应的就是电压最大值，否则电压的最大值保持不变，代码如下所示：

```
always@(posedge Clk)
if(counter >= UPDATERATE)
    VMax_r <= -16'd32767;//-32767
else if((ADC_IN[15] < VMax_r[15]) && Data_Valid)//IN 为正，r 为负，IN>r
    VMax_r <= ADC_IN;
else if((ADC_IN[15]==VMax_r[15]) && (ADC_IN > VMax_r) && Data_Valid)
    VMax_r <= ADC_IN;
else
    VMax_r <= VMax_r;
```

然后需要得到采集电压的最小值，当计数达到 1S 时，使电压最小值 VMin_r 为-32767（1111 1111 1111 1111）；当输入的电压 ADC_IN 最高位 ADC_IN[15]大于 VMin_r 最高位 VMin_r [15]并且数据有效时，代表此时输入的 ADC_IN 为负数，VMax_r 为正数，最小值 VMin_r 为 ADC_IN；当输入电压 ADC_IN 最高位 ADC_IN[15]等于 VMin_r 最高位 VMin_r [15]时，代表此时两个数都是正数或者复数，这时只需要比较两个值的大小，小的那个对应的就是电压最小值，否则，电压的最小值保持不变，代码如下所示：

```
always@(posedge Clk)
if(counter >= UPDATERATE)
    VMin_r <= 16'd32767;
else if((ADC_IN[15] > VMin_r[15]) && Data_Valid)//IN 为负，r 为正，IN<r
    VMin_r <= ADC_IN;
else if((ADC_IN[15]==VMin_r[15]) && (ADC_IN < VMin_r) && Data_Valid)
    VMin_r <= ADC_IN;
else
    VMin_r <= VMin_r;
```

最后，当计数达到 1S 后，将最大值和最小值进行输出，否则保持不变，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)begin
```

```
    VMin <= 0;
    VMax <= 0;
end
else begin
    if(counter >= UPDATERATE)begin
        VMin <= VMin_r;
        VMax <= VMax_r;
    end
    else begin
        VMin <= VMin;
        VMax <= VMax;
    end
end
```

得到最大、最下值之后，首先将其相加，然后除以 2，便得到其中间值，代码如下所示：

```
assign VMid_r = $signed(VMax) + $signed(VMin);
assign VMid = VMid_r[16:1];
```

上述代码中的$signed（）作用是将 VMax 和 VMin 作为有符号数进行相加，在进行运算的时候就会按照有符号数进行扩位，在高位补符号位，然后取 VMid _r[16:1]，这样就得到了 16 位的有符号的电压中值数据，举个例子，当最大值为+29408（0111001011100000），最小值为-32520（1111111100001000），两者相加得到-29160(1011100011101000)，取高 16 位得到-14580（1011100011110100），也就是将-29160 除以 2 的数值，也就是采集电压的中值。

1.4.1.4 ADC_Measure_Freq

ADC_Measure_Freq 模块用来得到采集信号的频率值，该模块的基本结构框图如图 1-16 所示：

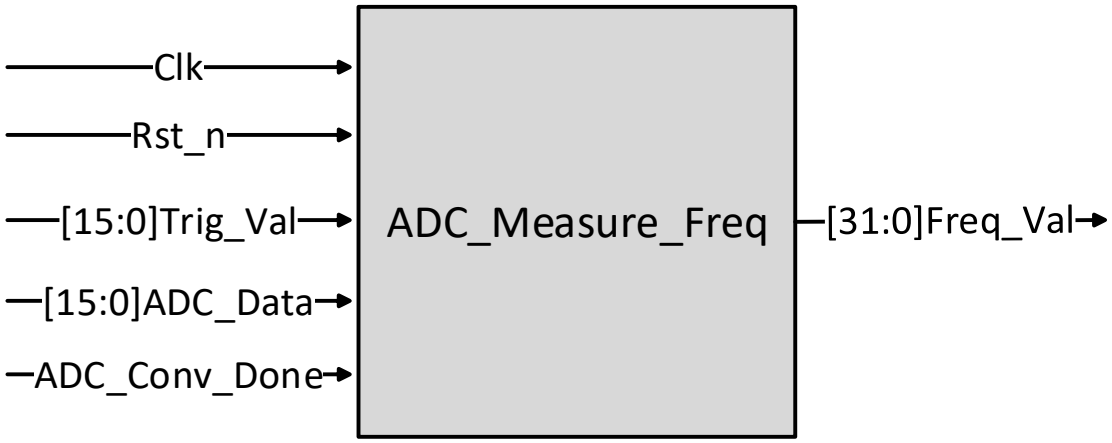

图 1-16 ADC_Measure_Freq 模块的基本结构框图

该模块的信号说明如下表 1-6 所示。

表 1-6 ADC_Measure_Freq 模块的信号说明表

信号名称	I/O	信号意义
Clk	I	模块的时钟信号 100M
Rst_n	I	模块的复位信号
Trig_Val [15:0]	I	频率测量触发值
ADC_Data[15:0]	I	ADC 采集到的 16 位数据
ADC_Conv_Done	I	ADC 单次采集完成信号
Freq_Val [31:0]	O	采集信号的频率测量值

首先设计一个计数器，根据设定的计数值来控制采样时间，这里设定计数值为 99999999，也就是计数 1S，代码如下所示：

```
parameter TIME_CNT_VAL = 99999999; //设定计数值来控制采样时间
//计数 1 秒
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Time_Cnt <= 0;
    else if(Time_Cnt >= TIME_CNT_VAL)
        Time_Cnt <= 0;
    else
        Time_Cnt <= Time_Cnt + 1;
end
```

保存触发设定值，防止中途被修改，每次测完一秒后才允许被修改，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Trig_Val_r <= 0;
    else if(Time_Cnt >= TIME_CNT_VAL)
        Trig_Val_r <= Trig_Val;
    else
        Trig_Val_r <= Trig_Val_r;
end
```

保存上一次 ADC 采集到的值，代码如下所示：

```
//保存上一次 ADC 测量值
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n) begin
        ADC_Data_Pre <= 0;
    end
    else if(ADC_Conv_Done) begin
        ADC_Data_Pre <= ADC_Data;
    end
    else begin
        ADC_Data_Pre <= ADC_Data_Pre;
    end
end
```

对 1S 内的触发点的个数进行计数，以此测得频率。当上次采样的电压值 ADC_Data_Pre 小于触发值 Trig_Val_r、本次采样的电压值 ADC_Data 大于 Trig_Val_r、上次采样电压值 ADC_Data_Pre 小于本次采样电压值 ADC_Data、单次采样完成，满足以上条件，触发周期计数加 1，这里以电压中值为触发，使用采集电压的中值作为触发值，是因为在中值两边的电压变化最明显，测得的频率最准，这里以正弦波为例，触发条件如下图 1-17 所示。

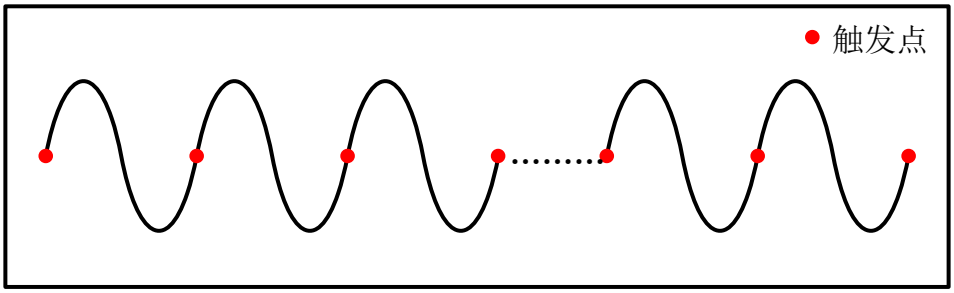

图 1-17 正弦波触发条件波形图

记录一次触发周期的时间，也就是满足触发条件所需时间，从上图可以看出，两次触发之间需要的时间刚好是一个周期的时间，代码如下所示：

```
//记录一次触发周期的时间
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Cycle_Cnt <= 0;
    else if((ADC_Data_Pre<=Trig_Val_r) && (ADC_Data>=Trig_Val_r) &&
(ADC_Data_Pre<ADC_Data) && ADC_Conv_Done)
        Cycle_Cnt <= 0;
    else
        Cycle_Cnt <= Cycle_Cnt + 1;
    end
```

保存上一次触发周期的时间，用于比较两次触发周期的时间，防止出现误触发的现象，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Cycle_Cnt_Pre <= 0;
    else if((ADC_Data_Pre <= Trig_Val_r) && (ADC_Data >= Trig_Val_r)
&& (ADC_Data_Pre < ADC_Data) && ADC_Conv_Done)
        Cycle_Cnt_Pre <= Cycle_Cnt;
    else
        Cycle_Cnt_Pre <= Cycle_Cnt_Pre;
end
```

记录 1S 内累加的触发计数，以此测得频率，如果当前周期计数小于等于上一个计数周期的一半，则此次触发异常，触发计数无效，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Freq_Add <= 0;
    else if(Time_Cnt >= TIME_CNT_VAL)
        Freq_Add <= 0;
    else if((ADC_Data_Pre <= Trig_Val_r) && (ADC_Data >= Trig_Val_r)
&& (ADC_Data_Pre < ADC_Data) && ADC_Conv_Done) begin
        if(Cycle_Cnt <= Cycle_Cnt_Pre[31:1])
            Freq_Add <= Freq_Add;
        else
            Freq_Add <= Freq_Add + 1;
    end
    else
        Freq_Add <= Freq_Add;
end
```

最后将计数 1 后测出的频率值进行输出，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if(!Rst_n)
        Freq_Val <= 0;
    else if(Time_Cnt >= TIME_CNT_VAL)
        Freq_Val <= Freq_Add;
    else
        Freq_Val <= Freq_Val;
end
```

1.4.1.5 RAM_Ctrl

本次实验将 AD7606 驱动模块 AD7606_Driver 的采样率设置为其最高采样频率 200Ksps，如果想要改变其采样频率，可以通过对采样数据进行抽取重采样的方法实现，比如希望以 40Ksps 的采样速率采样，则只需要每间隔 50 个采样数据取一个结果存储或使用，其他 49 个数据直接舍弃，这样就能实现 40Ksps 的采样率了。RAM_Ctrl 模块就是实现该功能的，该模块通过输入的采样分频值，最终产生数据有效标志，将需要数据写入至 RAM 中，该模块的基本结构如图 1-18 下所示。

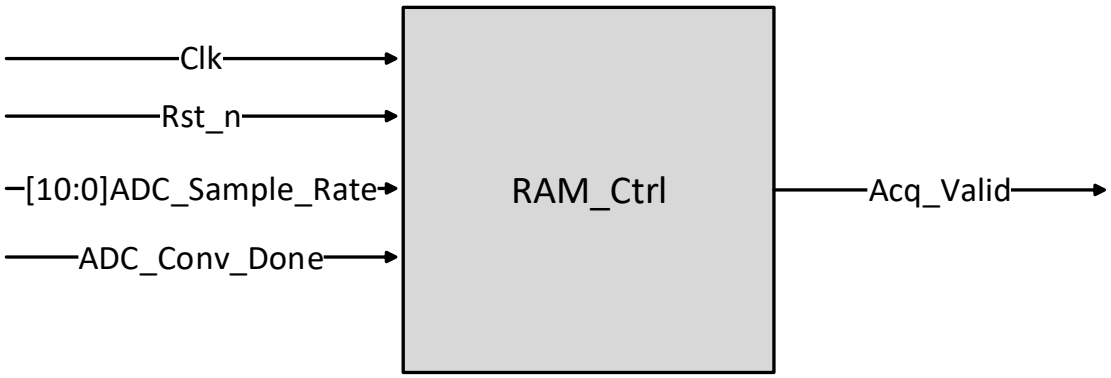

图 1-18 RAM_Ctrl 模块的基本结构图

该模块的信号说明如下表 1-7 所示。

表 1-7 RAM_Ctrl 模块信号说明表

信号名称	I/O	信号意义
Clk	I	模块的时钟信号 100M
Rst_n	I	模块的复位信号
ADC_Sample_Rate [10:0]	I	采样分频值
ADC_Conv_Done	I	ADC 单次采集完成信号
Acq_Valid	O	采集有效标志信号

首先设计一个采样分频计数器，根据采样分频值，当产生单次采集完成之后，计数器进行计数，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if (!Rst_n)
        Acq_Div_Cnt <= 0;
    else if(Acq_Div_Cnt >= ADC_Sample_Rate)
        Acq_Div_Cnt <= 0;
    else if(ADC_Conv_Done)
        Acq_Div_Cnt <= Acq_Div_Cnt + 1;
    else
        Acq_Div_Cnt <= Acq_Div_Cnt;
end
```

然后产生数据有效标志，当采样分频计数器达到设定值时，产生数据有效标志信号 Acq_Valid，否则，清除该标志信号，代码如下所示：

```
always@(posedge Clk or negedge Rst_n)
begin
    if (!Rst_n)
        Acq_Valid <= 0;
    else if(Acq_Div_Cnt >= ADC_Sample_Rate)
        Acq_Valid <= 1;
    else
        Acq_Valid <= 0;
end
```

1.4.1.6 Trigger_Detect

触发检测模块（Trigger_Detect）根据设置的不同触发模式得到的采集触发值，最终输出传输有效标志信号，通过该信号，产生 RAM 的写使能信号，将数据写入至 RAM 中，该模块的基本结构如下图 1-19 所示。

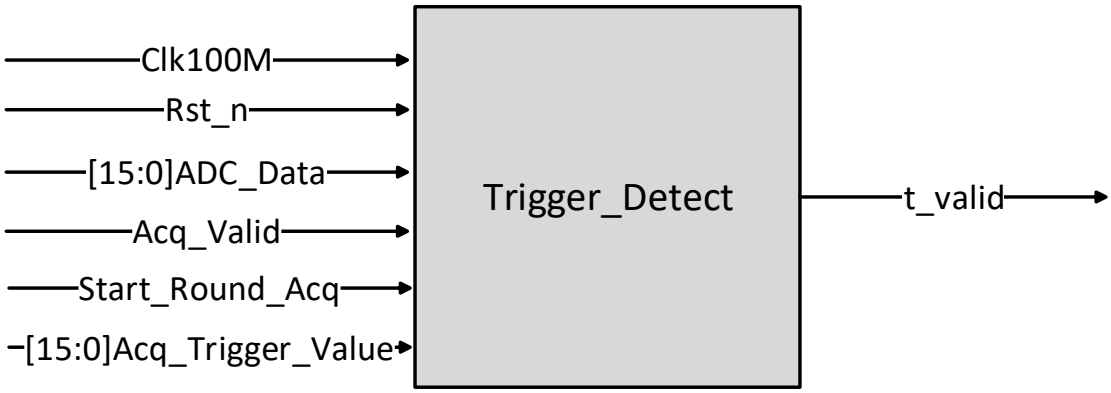

图 1-19 Trigger_Detect 模块的基本结构框图

对该模块的信号说明如下表 1-8 所示。

表 1-8 Trigger_Detect 模块的信号说明表

信号名称	I/O	信号意义
Clk100M	I	模块的时钟信号 100M
Rst_n	I	模块的复位信号
ADC_Data [15:0]	I	ADC 采集的 16 位的数据信号
Acq_Valid	I	采集有效标志信号
Start_Round_Acq	I	启动一轮采集标志信号
Acq_Trigger_Value[15:0]	I	采集触发值
t_valid	O	传输有效标志信号

首先设置 8 个 16 位的寄存器，当产生采集有效标志信号时，对 ADC 输出的 16 位的寄存器进行移位寄存，代码如下所示：

```
reg signed [15:0]ADC_Data_Reg[7:0];//寄存器
always@(posedge Clk100M)
if (Acq_Valid) begin
    ADC_Data_Reg[7] <= ADC_Data;
    ADC_Data_Reg[6] <= ADC_Data_Reg[7];
    ADC_Data_Reg[5] <= ADC_Data_Reg[6];
    ADC_Data_Reg[4] <= ADC_Data_Reg[5];
    ADC_Data_Reg[3] <= ADC_Data_Reg[4];
    ADC_Data_Reg[2] <= ADC_Data_Reg[3];
    ADC_Data_Reg[1] <= ADC_Data_Reg[2];
    ADC_Data_Reg[0] <= ADC_Data_Reg[1];
end
```

设置一个累加器，产生采集有效标志信号的时候，将 8 个 16 位的寄存器进

行累加，代码如下所示：

```
reg signed [18:0]ADC_Data_Reg_Add; //累加器
always@(posedge Clk100M)
begin
    if(Acq_Valid)
    ADC_Data_Reg_Add <= ADC_Data_Reg[7] + ADC_Data_Reg[6] +
                        ADC_Data_Reg[5] + ADC_Data_Reg[4] +
                        ADC_Data_Reg[3] + ADC_Data_Reg[2] +
                        ADC_Data_Reg[1] + ADC_Data_Reg[0];
end
```

然后得到 ADC 输出数据的平均值，也就是累加器的[18:3]的数据，相当于除以 8，代码如下所示：

```
wire signed [15:0]ADC_Average_Val; //平均值
assign ADC_Average_Val = ADC_Data_Reg_Add[18:3];
```

当产生数据有效信号的时候，将得到的 ADC_Average_Val 数据进行寄存，这样在产生下一次的 ADC_Average_Val 信号的时候，使其保持不变，就相当于保存上次的 ADC 采集的数据平均值信号，代码如下所示：

```
always@(posedge Clk100M or negedge Rst_n)
begin
    if (!Rst_n)
        ADC_Average_Val_Pre <= 0;
    else if(Acq_Valid)
        ADC_Average_Val_Pre <= ADC_Average_Val;
    else
        ADC_Average_Val_Pre <= ADC_Average_Val_Pre;
end
```

设计一个状态机，包含空闲状态和等待触发状态，如下所示：

```
localparam
    IDLE = 2'b01,//空闲态
    WAIT = 2'b10;//等待触发态
```

下面对这个状态的实现进行说明。

1. IDLE

空闲状态，当在该状态的时候，传输完成信号 Trigger_Done 为低电平，当产生开始信号的，进入等待触发状态，该状态的代码如下所示：

```
IDLE: begin
    Trigger_Done <= 0;
    if(Start_Signal)
        State <= WAIT;
    else
        State <= IDLE;
end
```

当产生启动开始信号 Start_Round_Acq 之后，将开始信号 Start_Signal 拉高，

进入等待触发状态之后，将该信号拉低，否则保持不变，代码如下所示：

```
always@(posedge Clk100M or negedge Rst_n)
begin
    if(!Rst_n)
        Start_Signal <= 0;
    else if(Start_Round_Acq)
        Start_Signal <= 1;
    else if(State == WAIT)
        Start_Signal <= 0;
    else
        Start_Signal <= Start_Signal;
end
```

2. WAIT

当处于等待触发状态的时候，如果上一次 ADC 输出数据的平均值 ADC_Average_Val_Pre 小于触发值 Acq_Trigger_Value 同时本次 ADC 输出数据的平均值 ADC_Average_Val 大于触发值 Acq_Trigger_Value，此时代表触发完成，将触发完成信号 Trigger_Done 拉低，并且进入空闲状态，该状态的代码如下所示：

```
WAIT: begin
    if((ADC_Average_Val_Pre<=Acq_Trigger_Value)&&(ADC_Average_Val>=
Acq_Trigger_Value)) begin
        Trigger_Done <= 1;
        State <= IDLE;
    end else
        Trigger_Done <= 0;
    end
```

在本次设计中，设置的一轮采集的长度 ONE_ROUND_LENGTH 为 1024，也就是 1024 个 ADC 输出的数据个数，根据采集长度，设计一个 Data_Valid，当产生触发完成信号 Trigger_Done 之后，将 Data_Valid 信号拉高，当计数器的值达到采集长度的时候，将 Data_Valid 信号拉低，代码如下所示：

```
always@(posedge Clk100M or negedge Rst_n)
begin
    if (!Rst_n)
        Data_Valid <= 0;
    else if(Trigger_Done)
        Data_Valid <= 1;
    else if(Cnt >= ONE_ROUND_LENGTH)
        Data_Valid <= 0;
end
```

当 Data_Valid 为高电平的时候，并且 Acq_Valid 信号有效的情况下，计数器 Cnt 进行计数，当计数值达到 1024 个时候，将计数器清零，然后重新计数，代码如下所示：

```
always@(posedge Clk100M or negedge Rst_n)
begin
    if (!Rst_n)
        Cnt <= 0;
    else if(Data_Valid)
        begin
            if(Cnt >= ONE_ROUND_LENGTH)
                Cnt <= 0;
            else if(Acq_Valid)
                Cnt <= Cnt + 1;
            else
                Cnt <= Cnt;
        end
    else
        Cnt <= 0;
end
```

最终当 Data_Valid 有效并且 Acq_Valid 有效的情况下，输出传输有效信号 t_valid，代码如下所示：

```
assign t_valid = Data_Valid && Acq_Valid;
```

至此，ADC_Acq2RAM_IP 中涉及的模块都已经介绍完了，接下来只需要将各个模块的端口连接以及产生 RAM 端口信号，关于各个模块的端口连接这里将不做介绍，请读者自行查看源代码，下面将介绍 RAM 端口信号的产生。

1.4.1.7 RAM 端口信号的产生

RAM 的写使能信号以及时钟使能信号，就是 Trigger_Detect 模块输出的传输有效信号 trans_valid，写入的数据就是 AD7606_Driver 输出的数据 ADC_Data，代码如下所示：

```
assign Ram_WREn = trans_valid;
assign Ram_WRData = ADC_Data;
assign Ram_WClockEn = Ram_WREn;
```

当产生写使能 Ram_WREn 信号的时候，产生写地址信号，知道地址信号增加至一次传输的长度的时候，又返回从地址 0 中写入数据，代码如下所示：

```
//产生写地址信号
always@(posedge Clk_100M or negedge hresetn)
if(!hresetn)
    Ram_WRADDR <= 16'd0;
else if(Ram_WREn) begin
    if(Ram_WRADDR >= ONE_ROUND_LENGTH - 1'd1)
        Ram_WRADDR <= 1'd0;
    else
        Ram_WRADDR <= Ram_WRADDR + 16'd1;
end
```

```
else
        Ram_WRADDR <= Ram_WRADDR;
```

当产生启动一轮采集信号 Start_Round_Acq 之后，将 RAM 的写完成信号 Ram_WR_END 拉低，当地址增加至传输长度之后，说明此时 RAM 写完成，将 Ram_WR_END 拉高，代码如下所示：

```
//产生 RAM 的写完成标志信号
always@(posedge Clk_100M or negedge hresetn)
if(!hresetn)
    Ram_WR_END <= 1'd0;
else if(Start_Round_Acq)
    Ram_WR_END <= 1'd0;
else if(Ram_WRADDR >= ONE_ROUND_LENGTH - 1'd1)
    Ram_WR_END <= 1'd1;
else
    Ram_WR_END <= Ram_WR_END;
```

综上所述，完成了 ADC_Acq2RAM_IP 模块的设计，接下来就需要将该模块例化至 cmsdk_ahb_eg_adc 模块中，这里将不再进行说明，请读者自行查看源代码。

1.4.2 cm3_system 模块设计

在 cm3_system 模块中例化 cmsdk_ahb_to_extmem16 模块和 cmsdk_ahb_eg_adc 模块，并将 AHB0 连接至 cmsdk_ahb_to_extmem16 模块，AHB1 连接至 cmsdk_ahb_eg_adc 模块，代码过长，这里模块的例化请查看源代码。

至此，FPGA 侧重点部分的代码就讲解完成了，关于完整的代码文件请读者自行查看源文件。

1.5 物理管脚约束

本次实验新增 AD7606 的管脚，将该模块的引脚约束添加至自己工程中的.upc 文件中，引脚约束如下所示：

```
#----AD7606 引脚分配-----------
phycst.pin.set {ad7606_cs_n_o} L3
phycst.pin.set {ad7606_rd_n_o} C8
phycst.pin.set {ad7606_busy_i} M2
phycst.pin.set {ad7606_db_i[0]} A3
phycst.pin.set {ad7606_db_i[1]} A5
phycst.pin.set {ad7606_db_i[2]} B5
phycst.pin.set {ad7606_db_i[3]} B7
phycst.pin.set {ad7606_db_i[4]} A7
phycst.pin.set {ad7606_db_i[5]} E9
phycst.pin.set {ad7606_db_i[6]} B9
```

```
phycst.pin.set {ad7606_db_i[7]} D10
phycst.pin.set {ad7606_db_i[8]} C9
phycst.pin.set {ad7606_db_i[9]} C10
phycst.pin.set {ad7606_db_i[10]} A11
phycst.pin.set {ad7606_db_i[11]} G11
phycst.pin.set {ad7606_db_i[12]} C11
phycst.pin.set {ad7606_db_i[13]} G12
phycst.pin.set {ad7606_db_i[14]} D4
phycst.pin.set {ad7606_db_i[15]} A4
phycst.pin.set {ad7606_os_o[0]} C6
phycst.pin.set {ad7606_os_o[1]} C4
phycst.pin.set {ad7606_os_o[2]} C7
phycst.pin.set {ad7606_reset_o} B2
phycst.pin.set {ad7606_convst_o} D6
```

1.6 编译设计

保存完修改后的顶层文件和物理约束文件之后，点击全部运行，生成本次实验需要的 FPGA 侧的 bin 文件，操作如图 1-20 所示。

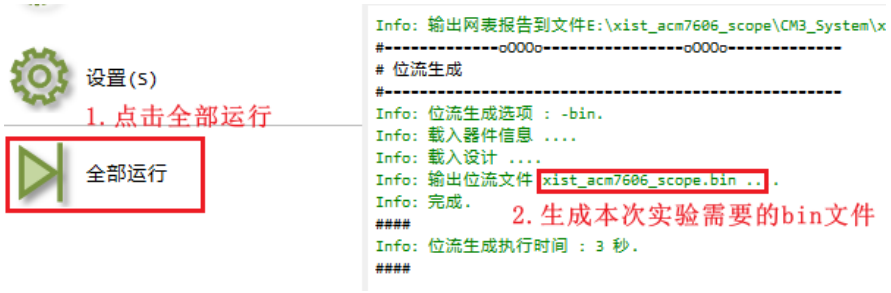

图 1-20 编译运行整个文件

1.7 建立 MDK 工程

将 TFT LCD 屏显示实验的 MDK 工程复制至本次工程的文件下，并对其进行修改，操作方式参考 2.8 节中的内容。

1.8 软件设计

本次实验需要新增的库文件，都在 CM3_Softerware\HARDWARE 文件夹下，读者在做本次实验的，需要将对应的库添加到工程中，这里以 ADC_Acq2ram 库为例，添加文件的方式如下图 1-21 所示。然后在弹出的对话框中添加文件夹，如下图 1-22 所示。

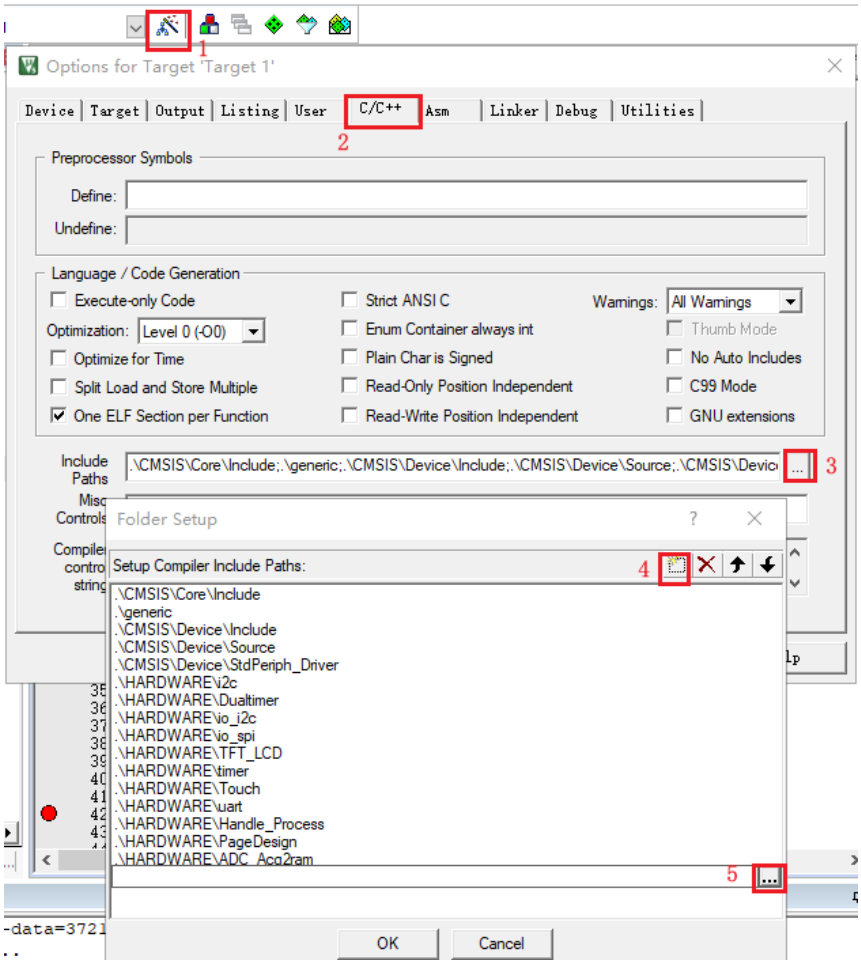

图 1-21 添加文件步骤

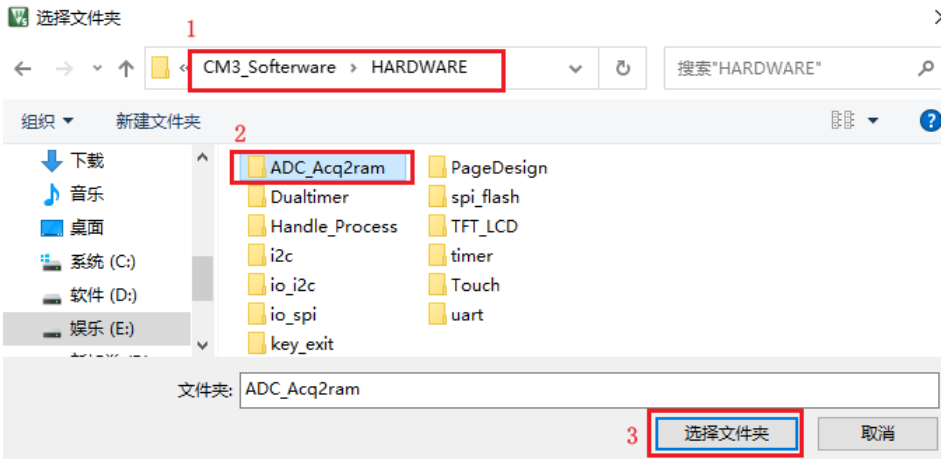

图 1-22 选择文件夹

将所需的文件按照上述方式进行添加，下面将对库文件中的部分重要函数进行说明。

1.8.1 部分库函数实现说明

1.8.1.1 初始化页面和控制器函数

Init_Homepage 函数用来初始化页面和控制器，函数中主要包含 LCD 屏的初始化、I2C 的初始化、触摸初始化、定时器初始化、以及一些背景、按键、显示波形窗口的绘制等，函数的实现的代码如下所示：

```
//初始化页面
void Init_Homepage()
{
    TFTLCD_Init();   //LCD 屏初始化
    Touch_IIC_Init();//I2C 初始化
    GT9147_Init();   //触摸初始化

    //定时器初始化，每 10ms 触发一次
    Timer_Init(CM3DS_MPS2_TIMER0,SystemCoreClock/100,SystemCoreClock/100);
    TIM_EXTI_Init(CM3DS_MPS2_TIMER0,ENABLE);

    LCD_Clear(LCD_BLACK);   //设置整体背景为黑色
    POINT_COLOR = LCD_WHITE;//笔画设定为白色
    BACK_COLOR = LCD_BLACK; //笔画背景设定为黑色

    LCD_ShowString(3,20,180,16,16,0,(uint8_t*)"www.corecourse.cn");//显示网址
    LCD_Set_Window(20,50,96,96);//开窗
    LCD_DisplayPic(20,50,9216,gImage_logo);//画 LOGO

    Draw_Box(ADC_Wave.Window, 0);
    Get_Waveform_Window_Parameters(&ADC_Wave);
    Draw_Waveform_Windows(ADC_Wave, Sample_Rate[Sample_Set]);//KHz

    Draw_Normal_Button(Button_RUN);
    Draw_Normal_Button(Button_AUTO);
    Draw_Normal_Button(Button_CH_INC);
    Draw_Normal_Button(Button_CH_DEC);
    Draw_Normal_Button(Button_SA_INC);
    Draw_Normal_Button(Button_SA_DEC);
    Draw_Normal_Button(Button_Tri_Mode);
    Draw_Normal_Button(Wave_Slider);
    Fill_Box(Slider,LCD_BLUE,5);

    sprintf(str,"%.3fV",Get_ADC_Vmid());   //得到 ADC 采集到的电压值
    strcpy(Button_TriggerVal.Text[0],str);
    Draw_Normal_Button(Button_TriggerVal);

    sprintf(str,"CH%d",CH_Set);
    Draw_Normal_Text(Text_CHANNEL, str);
    sprintf(str,"%dKHz",(int)Sample_Rate[Sample_Set]);
    Draw_Normal_Text(Text_SAMPLE, str);
```

```
    //先采集一次，设置 ADC 寄存器相关值
    Set_ADC_Mode(CH_Set, Sample_Rate[Sample_Set], Trigger_Value, Trigger_Mode);
    ADC_DataTransfer(P_ADC_Data,ADC_DATA_LENGTH);
    //测量点为 Y 轴线
    Measure_Point = ADC_Wave.Wave_Area.X1 + ADC_Wave.Wave_Area.Width/2;
    //触发点为 X 轴线
    Trigger_Point = ADC_Wave.Wave_Area.Y1 + ADC_Wave.Wave_Area.Height/2;
}
```

关于绘图和初始化 TFT、触摸的部分将不做介绍，下面将对该函数中比较重要的几个点进行介绍。

1. 定时器初始化

在函数中，初始了定时 0 进行工作，每 10ms 触发一次，通过计时器控制每个进程的触发间隔，启动对应的标志位，各个时间的标志位和各个进程标志位如下所示：

```
//各个时间标志位
static uint8_t Flag_20ms = 0;
static uint8_t Flag_30ms = 0;
static uint8_t Flag_50ms = 0;
static uint8_t Flag_100ms = 0;
static uint8_t Flag_200ms = 0;
static uint8_t Flag_500ms = 0;

//定义各个进程的标志位
uint8_t Flag_DrawWave = 0;      //绘制波形，10ms 一次
uint8_t Flag_DrawGrid = 0;      //绘制网格，20ms 一次
uint8_t Flag_TouchScan = 0;     //触摸扫描，30ms 一次
uint8_t Flag_Refresh_Val = 0;   //刷新电压和频率数值，500ms 一次
```

关于标志位的使用将在后面进行介绍，得到需要控制的标志位之后，在定时器中断函数中实现对标志位的控制，代码如下所示：

```
void TIMER0_Handler(void)
{
    /*********添加定时器 0 的中断服务函数**********/
    //以下通过计时器控制每个进程的触发间隔，启动对应的标志位
    Flag_DrawWave = 1;
    if(Flag_20ms >= 1) {
        Flag_20ms = 0;
        Flag_DrawGrid = 1;
    }
    else
        Flag_20ms++;
    if(Flag_30ms >= 2) {
        Flag_30ms = 0;
        Flag_TouchScan = 1;
    }
```

```
    else
        Flag_30ms++;
    if(Flag_50ms >= 4) {
        Flag_50ms = 0;

    }
    else
        Flag_50ms++;
    if(Flag_100ms >= 9) {
        Flag_100ms = 0;
    }
    else
        Flag_100ms++;

    if(Flag_200ms >= 19) {
        Flag_200ms = 0;
    }
    else
        Flag_200ms++;
    if(Flag_500ms >= 49) {
        Flag_500ms = 0;
        Flag_Refresh_Val = 1;
    }
    else
        Flag_500ms++;
    /*****************************************/
    TIM_ClearIT(CM3DS_MPS2_TIMER0);
}
```

2. 设置 ADC 相关寄存器模式以及获取相关寄存器中的电压值、频率值

关于 ADC 寄存器的相关控制都在 ADC_Acq2ram 库中，接下来我们将讲解该库中的部分函数。

在介绍该库中函数的实现之前，首先介绍 ADC_Acq2ram 库中涉及的寄存器，如下表 1-9 所示，关于寄存器地址的设定请参看前面“cmsdk_ahb_eg_adc_reg”模块的设计。

表 1-9 ADC 寄存器说明表

寄存器名称	寄存器地址	寄存器意义
REG_ADC_CONTROL	ADC_IP_BASEADDR+0x20	ADC 控制寄存器 [2:0] ADC 通道选择 Channel [13:3]ADC 采样分频 Sample_Div [29:14] ADC 触发电压值 Trigger_Val [31:30] ADC 触发模式 Trigger_Mode
REG_ADC_MINMAX	ADC_IP_BASEADDR+0x24	ADC 采集电压的最大最小值寄存器 [15:0]ADC 采样电压的最小值 [31:16]ADC 采样电压的最大值

REG_ADC_STATE	ADC_IP_BASEADDR+0x28	ADC 采集电压的中值、采样触发状态、一次采样完成信号寄存器 [15:0] ADC 采集信号中值 ADC_Mid_Val [17:16]采样触发状态信号 Acq_Trigger_State [19:18]一次采样完成信号 Acq_Round_Done。
REG_ADC_FREQ	ADC_IP_BASEADDR+0x2C	ADC 采样频率寄存器，该寄存器中存储采样信号的频率值

注：寄存器地址中的 ADC_IP_BASEADDR 为 AHB1 的地址 0xC0000000

读寄存器的操作就是读取对应地址中的数据，实现方式如下所示：

```
uint32_t GET_ADC_REG(uint32_t address)
{
    uint32_t *data;
    data = (unsigned int *)address;
    return *data;
}
```

写寄存器的操作就是向寄存器中地址中写入值，实现方式如下所示：

```
void SET_ADC_REG(uint32_t *address,uint32_t data)
{
    *address = data;
}
```

了解读写寄存器以及寄存器的含义之后，我们需要先设置 ADC 控制寄存器中的值，也就是向 ADC_IP_BASEADDR+0x20 地址中的对应位写入值，代码如下所示：

```
void Set_ADC_Mode(uint32_t Channel,uint32_t Sample_Rate,
        uint32_t Trigger_Val,uint32_t Trigger_Mode)
{
    uint32_t Sample_Div;
    Sample_Div = 200/Sample_Rate;//采样分频=原始采样率 200K/设定采样率
    SET_ADC_REG((uint32_t *)REG_ADC_CONTROL, Channel|(Sample_Div<< 3)
        | (Trigger_Val << 14) | (Trigger_Mode << 30));
    }
```

在 Init_Homepage 函数中，我们也使用到了该函数，如下所示：

```
Set_ADC_Mode(CH_Set, Sample_Rate[Sample_Set], Trigger_Value, Trigger_Mode);
```

对上述变量说明如下所示：

表 1-10 Set_ADC_Mode 函数变量说明表

变量名称	变量含义	变量值获取
CH_Set	通道设置	一共有 8 个通道可供选择（0~7），默认通道 0，可以通过触摸切换
Sample_Rate	采样分频设置	采样分频值=原始采样率 200K/设定采样率，可以通过触摸设置
Trigger_Value	触发电压值	默认触发值为 0V，可以通过设置
Trigger_Mode	触发模式设置	0 表示自动触发，1 表示手动触发，2 表示单次触发，通过触摸设置

设置完 ADC 的控制器之后，我们需要读取电压、频率相关寄存器中的值，读完之后，将这些值显示在 TFT LCD 显示屏上，这里以读取电压最大值为例，

通过 Get_ADC_Vmax 函数实现读取电压的功能，电压最大值对应 GET_ADC_REG 的高 16 位，读取完成后，将得到的数值转换成电压值，转换公式如下所示：

$$V = \frac{V_{Tmp}}{32768} \times 5$$

上述公式中的V_{Tmp}是从寄存器中读取出来的值，V 是实际电压值，也就是显示屏上需要显示的值，由此，我们便得到了 Get_ADC_Vmax 函数的实现方式，如下所示：

```
float Get_ADC_Vmax()
{
    uint32_t Vmax_Val;
    int16_t Vmax_Tmp;
    float Vmax;
    Vmax_Val = (GET_ADC_REG(REG_ADC_MINMAX) >> 16);
    Vmax_Tmp = Vmax_Val;    //无符号转有符号
    Vmax = Vmax_Tmp/32768.0*5.0;
    return Vmax;
}
```

获取电压的最小值、频率值、中值的方式都可以参考上述方法实现。

3. 获取 RAM 中存储的数据

读取数据之前，首先对 RAM 相关寄存器进行说明，如下表 1-11 所示，这些寄存器的基地址都是 AHB1 的地址。

表 1-11 RAM 相关寄存器说明表

寄存器名称	寄存器偏移地址	寄存器意义
RD_ADDR_OFFSET	0x18	RAM 地址寄存器
RD_DATA_OFFSET	0x1C	RAM 数据寄存器
RD_WR_END_OFFSET	0x30	RAM 写完成寄存器

通过上表，我们便可以向 RAM 地址寄存器中写入地址，然后读取数据寄存器中存储的对应地址中的数据，这样我们就得到了读取 RAM 中存储数据的方式，也就是 ADC_DataTransfer 函数的实现方式，如下所示，其中长度为 1024。

```
uint32_t ADC_DataTransfer(uint16_t *BuffAddr, uint32_t Length)
{
    int i = 0;
    for (i = 0; i < Length; i++){
        //将写入的数据从 FPGA RAM 回读
        AHB_SlaveWrite((uint32_t *)(CM3DS_MPS2_TARGEXP1_BASE+RD_ADDR_OFFSET), i);
        BuffAddr[i] = AHB_SlaveRead(CM3DS_MPS2_TARGEXP1_BASE + RD_DATA_OFFSET);
    }
}
```

Init_Homepage 函数中还涉及许多函数，包括画按键、显示电压值、频率值字符串等操作函数，这里将不做介绍，请读者自行查看源代码

1.8.1.2 单次触发函数

Handle_Single_Trigger 主要用来处理单次触发事件，在该函数中，首先通过 ADC_DataTransfer 函数读取 RAM 中存储的数据，当处于单次触发时，恢复按钮的颜色，绘制按钮，触发取消则不再 STOP，否则执行 STOP 操作，其实其中涉及的都是按钮颜色背景的切换，比较容易理解，这里就不再进行详细说明，函数的实现代码如下所示：

```
void Handle_Single_Trigger()
{
    ADC_DataTransfer(P_ADC_Data,ADC_DATA_LENGTH);
    //按钮恢复原色
    if(Trigger_Mode == 0x03) {
        Button_TriggerVal.TextColor = LCD_BLACK;
        Button_TriggerVal.BackColor = LCD_GREEN;
    } else {
        Button_TriggerVal.TextColor = LCD_WHITE;
        Button_TriggerVal.BackColor = LCD_GRAY;
    }
    Draw_Normal_Button(Button_TriggerVal);

    Trigger_Enable_Press();

    if(!Cancel_Trigger)//触发取消则不再 STOP
        Perform_STOP();//执行 STOP 操作
    Cancel_Trigger = 0;      //清标志
}
```

1.8.1.3 启动下一次传输

启动下一次传输函数（Handle_Round_Done）首先通过 Set_ADC_Mode 函数设置 ADC 的控制寄存器，然后通过 ADC_DataTransfer 函数读取 RAM 中存储的数据，代码如下所示：

```
void Handle_Round_Done()
{
    //开启下一次传输
    Set_ADC_Mode(CH_Set, Sample_Rate[Sample_Set], Trigger_Value, Trigger_Mode);
    ADC_DataTransfer(P_ADC_Data,ADC_DATA_LENGTH);
}
```

1.8.1.4 刷新波形窗口

刷新波形窗口（Refresh_WaveWindow）函数首先读波形数据、然后执行刷新电压值、刷新时间显示等操作，函数的实现代码如下所示：

```
void Refresh_WaveWindow()
{
    Read_Wave_Data();//读波形数据
    Draw_Waveform(ADC_Wave,Pre_Wave_Data,Wave_Data);    //画波形
    if(Wave_Run) {
        if(Trigger_Mode)
            POINT_COLOR=LCD_ORANGE;     //画笔颜色
        else
            POINT_COLOR=LCD_BRRED;      //画笔颜色
            //画横标线
            Draw_Mark_Line(0,Trigger_Point, ADC_Wave.Wave_Area.X1,
                    ADC_Wave.Wave_Area.X1 + ADC_Wave.Wave_Area.Width);
    } else if(!Wave_Run) {
        POINT_COLOR=LCD_CYAN;      //画笔颜色
        //画竖标线
        Draw_Mark_Line(1,Measure_Point, ADC_Wave.Wave_Area.Y1,
                ADC_Wave.Wave_Area.Y1 + ADC_Wave.Wave_Area.Height);

        //刷新电压显示
        Point_Voltage=Wave_Data[Measure_Point - ADC_Wave.Wave_Area.X1]*5.0/32768;
        if(Point_Voltage >= 0)
            sprintf(str,"Voltage = +%.3fV",Point_Voltage);
        else
            sprintf(str,"Voltage = %.3fV",Point_Voltage);
        Draw_Normal_Text(Text_Point_V, str);

        //刷新时间显示
        Point_Time=(float)((Measure_Point-ADC_Wave.Wave_Area.X1)-
250)/Sample_Rate[Sample_Set];
        if(Point_Time >= 0)
            sprintf(str,"Time = +%.3fms",Point_Time);
        else
            sprintf(str,"Time = %.3fms",Point_Time);
        Draw_Normal_Text(Text_Point_T, str);
    }
    POINT_COLOR=LCD_BLACK;      //画笔颜色
}
```

上述函数中主要都是关于绘图的说明，这里将不做介绍，其中比较重要的就是读取波形数据 Read_Wave_Data 函数的实现，函数波形的显示一共分为两页显示，一页最多显示 500 个数据，首先保存上一次的波形数据，然后根据界面上的滑动按钮，切换 Wave_Data 保存的数据的起始地址，也就是设置的 P_ADC_Data 中的起始第一个需要显示的数据，代码如下所示：

```
void Read_Wave_Data()
{
    uint16_t i ;
```

```
    memcpy(Pre_Wave_Data,Wave_Data,ADC_DATA_LENGTH*2);
    for(i = 0 ;i<ADC_DATA_LENGTH;i++)
    {
        Wave_Data[i] = P_ADC_Data[i+ADC_Wave_Offset];
    }
}
```

综上所述，对于函数库中比较重要的部分，也就是数据处理部分做了详细介绍，其中关于触摸以及画图的部分请用户自行查看源代码进行理解。

1.8.2 main 文件代码实现

在 main 函数中首先初始化主页和控制器，然后循环处理事件 Handle_Events，Handle_Events 事件首先读取 RD_WR_END_OFFSET 寄存器的值，判断是否写完，为 1 则代表写完，为 0 则没有，然后就是在 Handle_Events 函数执行一系列事件，需要执行的事件，如下所示：

1. 如果开启了单次触发，触发成功则 STOP，取消触发则恢复原状；
2. 每轮传输完成则开启下一轮传输；
3. 检测到 Flag_DrawWave 标志信号，定时刷新波形窗口，10ms 一次，取消定时中断函数中标记的 Flag_DrawWave；
4. 检测到 Flag_TouchScan 标志信号，定时检测触摸，30ms 一次，取消定时中断函数中标记的 Flag_TouchScan；
5. 检测到 Flag_DrawGrid 标志信号，定时刷网格背景，20ms 一次，取消定时中断函数中标记的 Flag_DrawGrid；
6. 检测到 Flag_Refresh_Val 标志信号，定时刷测量值，500ms 一次，取消定时中断函数中标记的 Flag_Refresh_Val；

综上所述，便可以得到 Handle_Events 函数的实现代码，如下所示：

```
void Handle_Events(void)
{
    ADC_ROUND_DONE = AHB_SlaveRead(CM3DS_MPS2_TARGEXP1_BASE +
RD_WR_END_OFFSET);
    //如果开启了单次触发，触发成功则 STOP，取消触发则恢复原状
    if(Single_TriggerFlag && (ADC_ROUND_DONE || (Cancel_Trigger))) {
        Single_TriggerFlag = 0; //清标志
        Handle_Single_Trigger();//处理单次触发事件
    }
    //每轮传输完成则开启下一轮传输
    if(ADC_ROUND_DONE && Wave_Run && (!Single_TriggerFlag)) {
        Handle_Round_Done();
    }
    //定时刷新波形窗口，10ms 一次
    if(Flag_DrawWave) {
```

```
        Flag_DrawWave = 0;
        Refresh_WaveWindow();    //刷新波形窗口
    }
    //定时检测触摸，30ms 一次
    if(Flag_TouchScan) {
        Flag_TouchScan = 0;
    Touch_Scan();                    //触摸扫描
    }
    //定时刷网格背景，20ms 一次
    if(Flag_DrawGrid) {
        Flag_DrawGrid = 0;
        Draw_Grid_Background(ADC_Wave); //绘制网格背景
    }
    //定时刷测量值，500ms 一次
    if(Flag_Refresh_Val && Wave_Run) {
        Flag_Refresh_Val = 0;
        Refresh_Measure_Val();          //刷新测量值
    }
}
```

在 main 函数中，调用 Handle_Events 即可，main 中代码如下所示：

```
int main()
{
    int i = 0;
    Init_Homepage();//初始化主页和控制器
    //循环处理事件
    while(1) {
        Handle_Events();
    }
    return 0;
}
```

至此，本次实验的代码就设计完成了，接下来便可以进行板级验证了。

1.9 板级验证

1.9.1 实验所需硬件

(1) AC208 开发板
(2) FPGA 下载器：XIST USB Cable
(3) CM3 仿真器：DAP Link
(4) 电源线一根
(5) TFT LCD 屏一块
(6) ACM7606 数据采集模块
(7) 信号发生器

1.9.2 硬件连接

JTAG 和 DAP Link 的硬件连接参考实验一。

在连接 TFT LCD 屏的时候，由于开发板显示接口有 2*18 共 36 个接口座，而 TFT LCD 屏只有 34 个接口，所以在连接的时候，请保持靠下插接的方式，空出主板上显示扩展接口的 1、2 两个脚，即对应的开发板后面显示的 GND 和 3V3 不要接。需要空出的接口如下图 1-23 所示。

图 1-23 TFT LCD 屏连接时需要空出的引脚

TFT LCD 屏正确的连接方式如下图 1-24 所示。

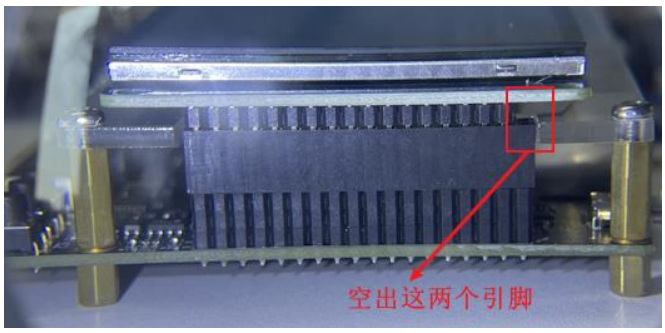

图 1-24 TFT LCD 显示屏正确连接示意图

ACM7606 模块连接至 AC208 开发板上最下面一排 40pin 的排针，靠右连接，也就是靠近数码管的一边，整体的硬件连接图如下图 1-25 所示。

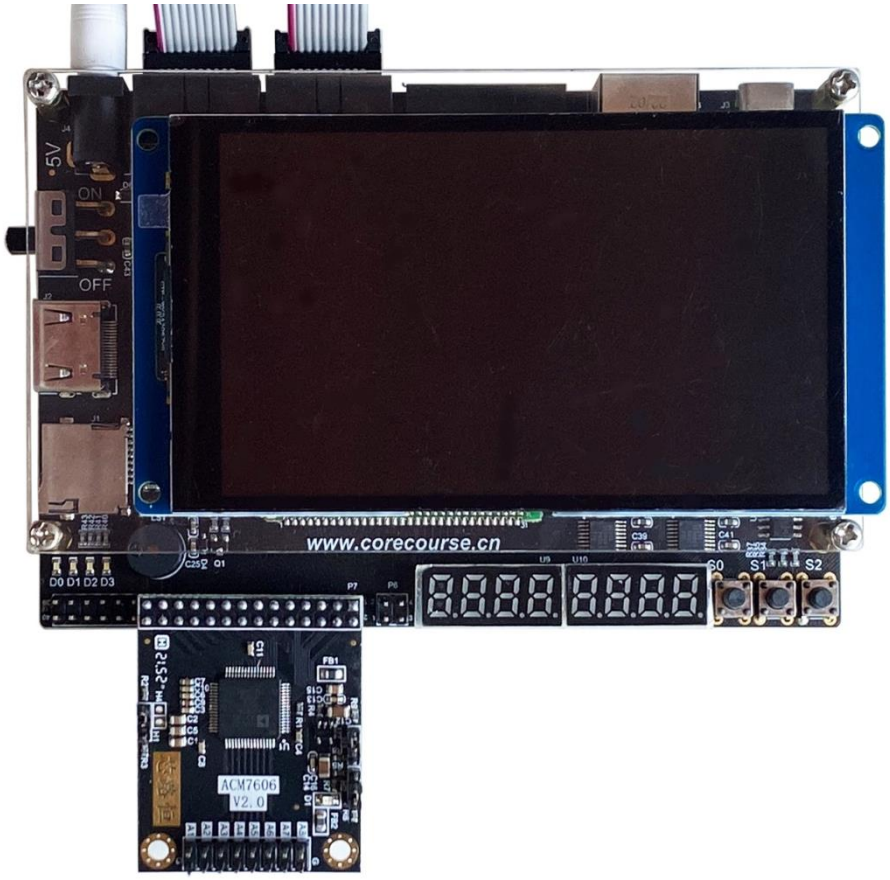

图 1-25 系统整体硬件连接图

1.9.3 下载文件至开发板

下载文件的方式参考实验一。

1.9.4 功能演示

程序下载成功之后，初始界面如下图 1-26 所示（未连接信号源）。

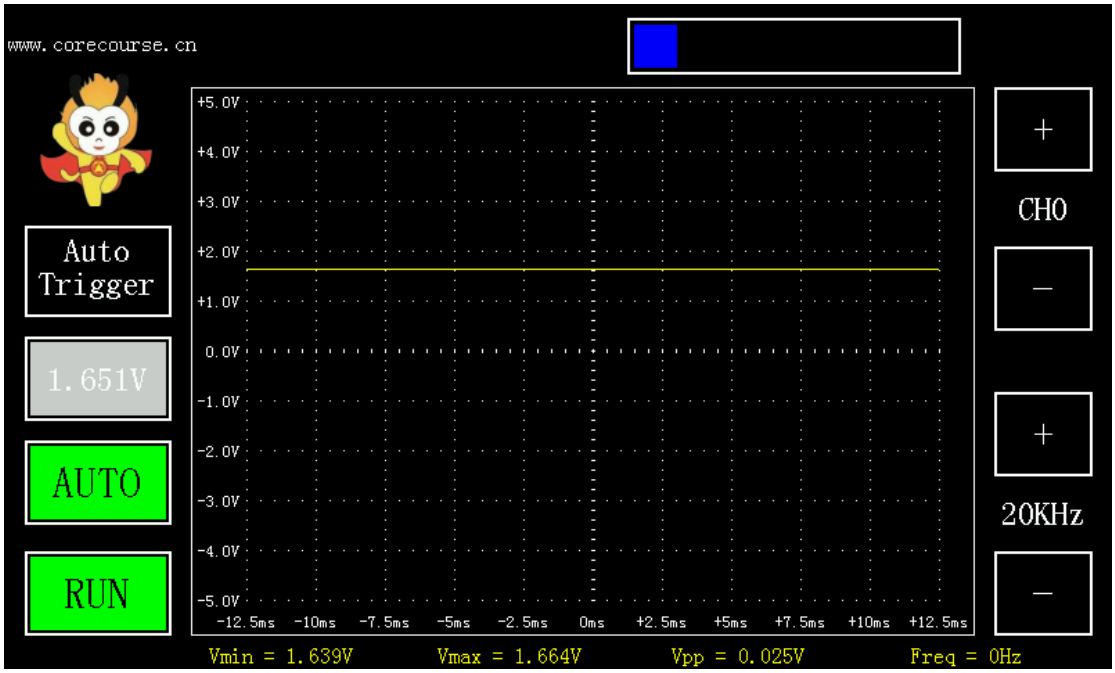

图 1-26 初始界面图

然后给 AD7606 的通道 1 输入一个 200hz，Vpp=5V 的正弦波，显示如下图 1-27，从图中最下一行的显示中，可以看到，其采集电压最大、最小值都为 2.5V 左右，频率为 200hz，与我们输入的信号一致。

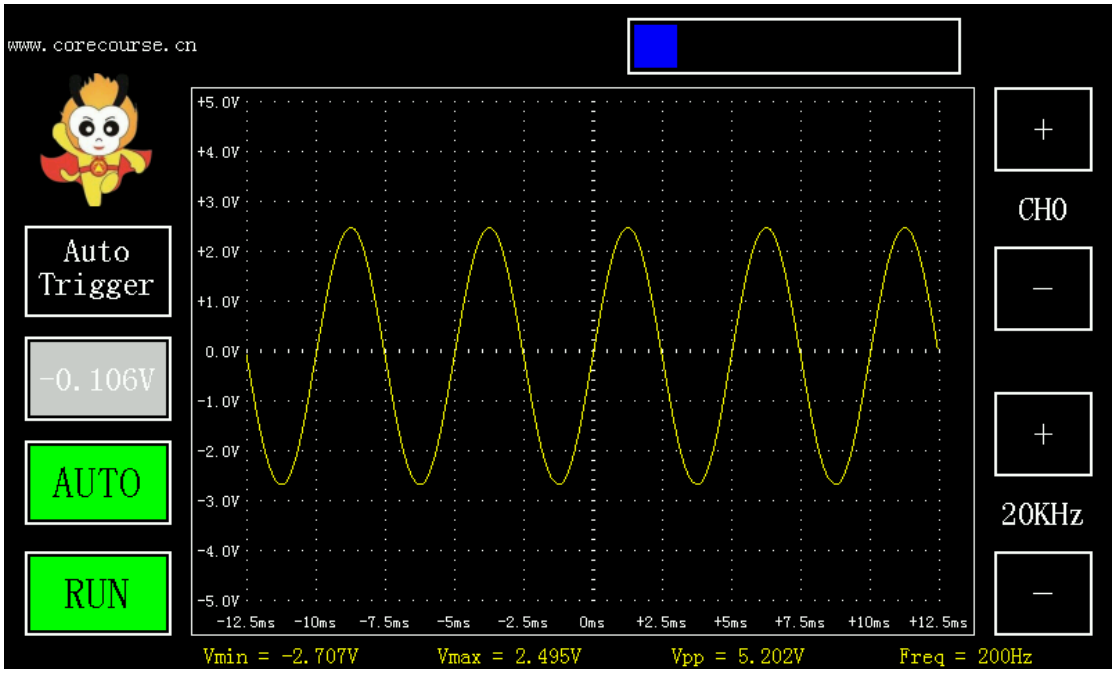

图 1-27 输入信号源之后，显示波形图

下面将对主界面中设计的一些功能进行说明。

1.9.4.1 主界面说明

主界面如下图 1-28 所示。

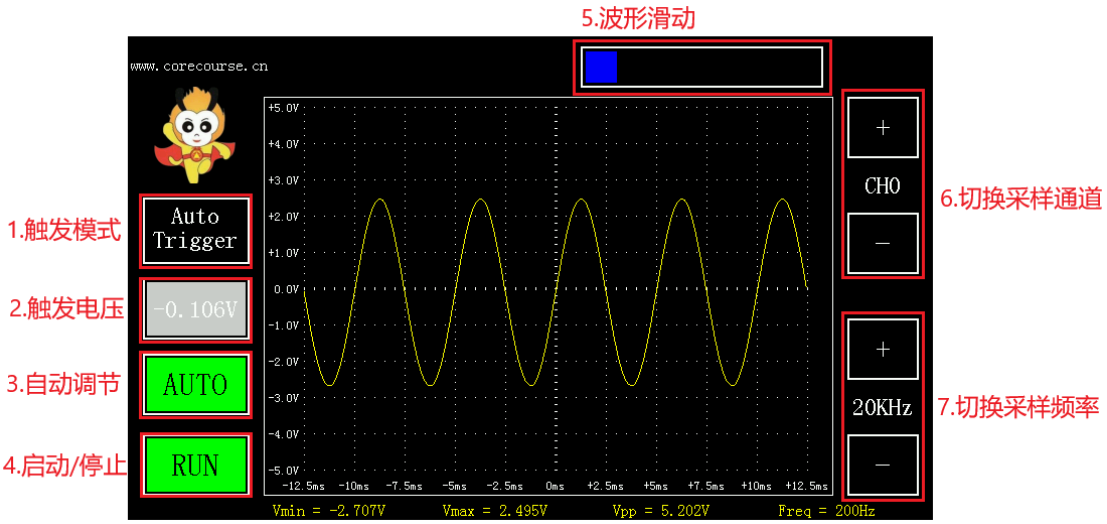

图 1-28 主界面示意图

1. 触发模式：可选自动触发、普通触发、单次触发三种模式；
2. 触发电压：手动模式下不可更改，在其他模式下，可通过滑动波形窗口的橙色标线来选择当前的触发电压；
3. 自动调整：会根据当前输入的波形频率，自动调整到最佳的采样率，并将触发模式切换为自动模式；
4. 启动/停止：启动/停止波形采集；
5. 波形移动：本工程存储了长度为 1024 的波形，但窗口只显示 500，拖动滑块即可移动波形；
6. 切换采样通道：点击“+”、“-”可切换 CH0~CH7 共 8 个采样通道；
7. 选择采样率：点击“+”、“-”可切换 1KHz~1000KHz 共 10 种采样率；
8. 显示输入电压的最值、峰峰值、频率值。

1.9.4.2 测量界面介绍

测量界面如下图 1-29 所示，在点击“RUN”按钮后，“RUN”会变为“STOP”表示停止波形采集，并进入测量界面在测量界面会有一条青色竖标线，触摸可以移动标线，在上方会显示标线与波形交点处的电压值、相对时间值

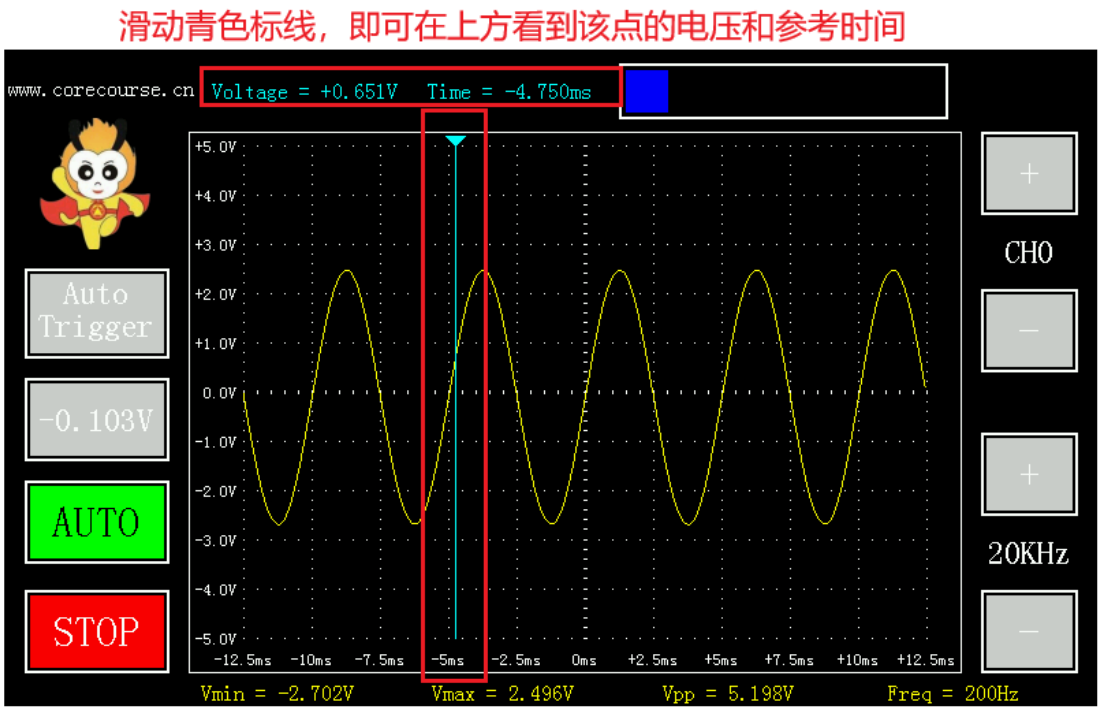

图 1-29 测量界面示意图

1.9.4.3 触发模式介绍

本工程支持三种触发方式：自动触发、普通触发、单次触发

1. 自动触发：触发值自动选择为输入电压的中值，如下图 1-30 所示。

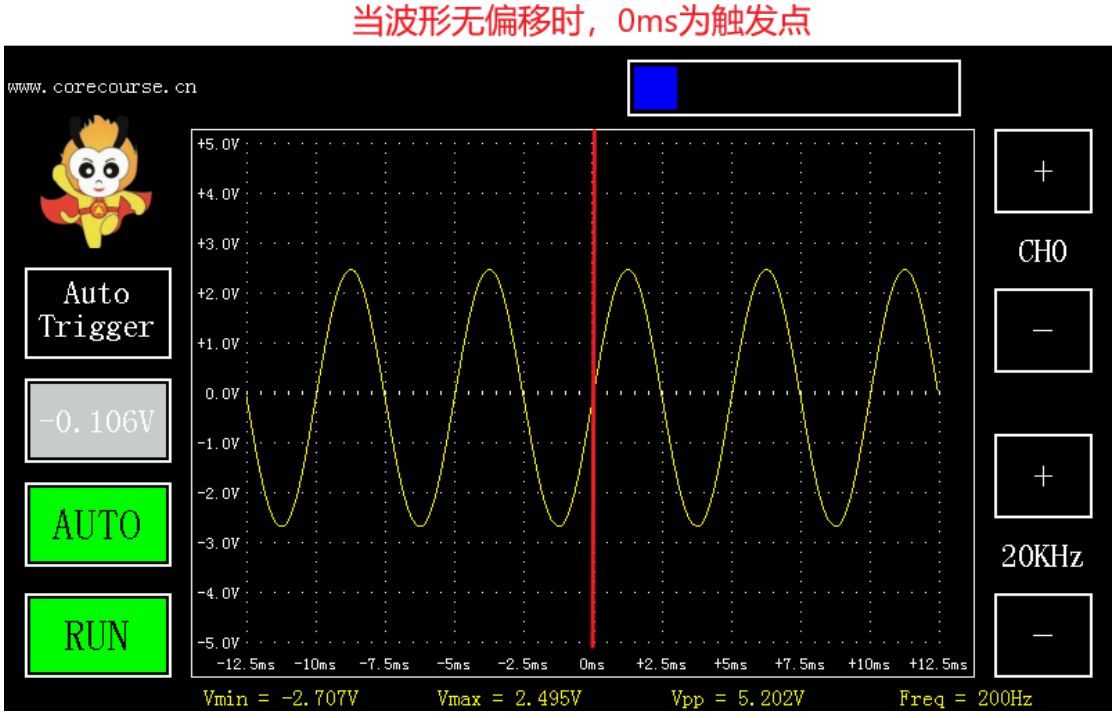

图 1-30 自动触发显示示意图

2. 普通触发：触发值由用户自己选择。如下图 1-31，点击触发模式切换按键，切换到普通触发模式，然后设置触发电压为 0.917V，可以看到波形触发点变为 0.917V。

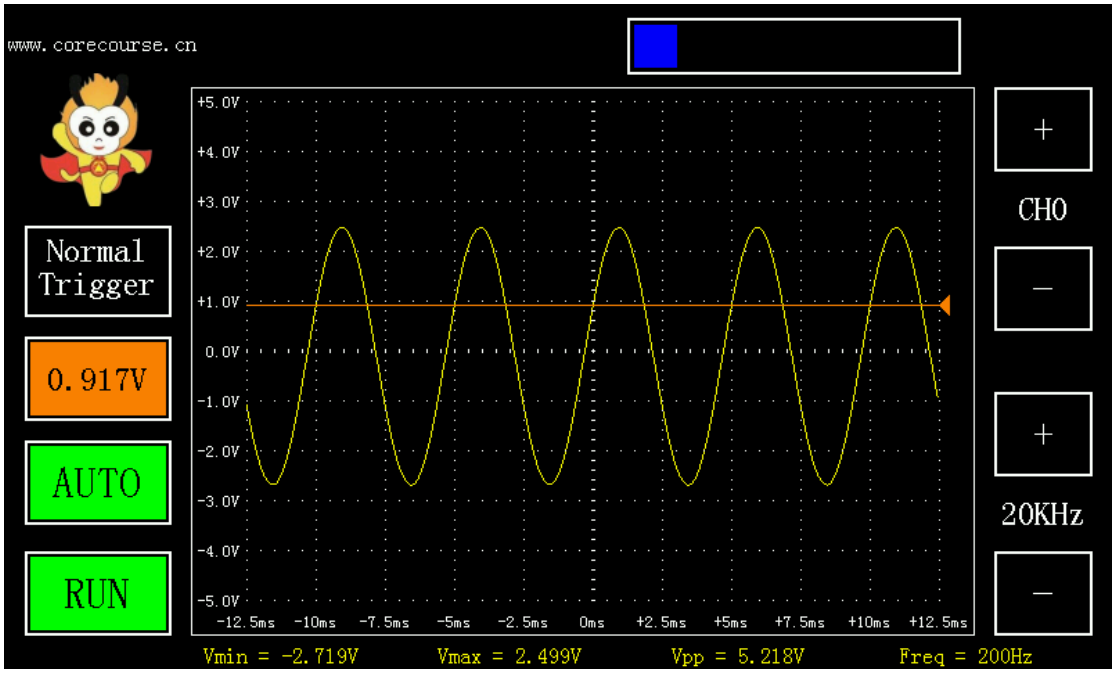

图 1-31 普通触发示意图

3. 单次触发：点击触发模式切换按键，切换到单次触发模式，“触发电压”按钮会变为绿色可触摸，然后设置触发电压为 0.750V，点击“0.750V”启动单次触发，在等待触发时，“触发电压”按钮为红色，再次点击可退出等待，如下图 1-32 所示。

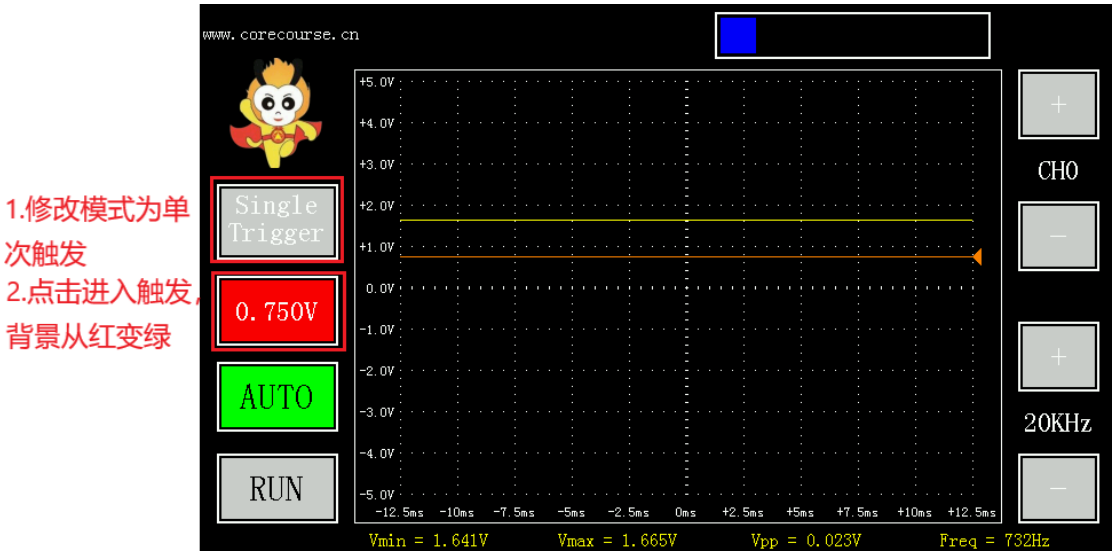

图 1-32 单次触发波形图

触发完成之后，会进入停止状态，方便观测波形，如下图 1-33 所示。

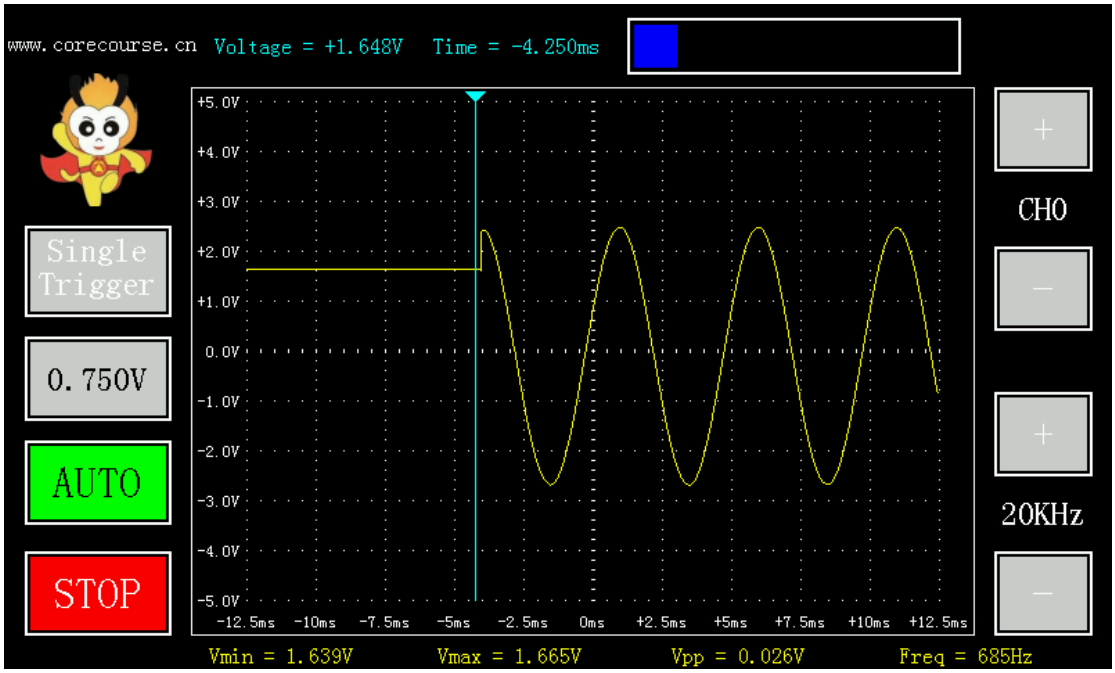

图 1-33 触发完成波形图

1.10 思考与总结

本章设计以 TFT_LCD 实验为基础，结合 AD7606 模块完成了多通道数据采集系统的搭建，用户在使用时可以直接通过触摸 LCD 屏上对应按钮，实现不同功能的切换/控制以及采集通道的选择。

在进入单次触发之后，我们可以看到波形并不是从设定的触发电压之后开始显示波形，而是在设定的触发电压之前就已经有波形数据了，这是因为我们在程序中添加了一个移位寄存器，将 AD 采集的数据进行移位之后，才送到 RAM 中进行显示，所以这里才能看到在触发之后，还会显示触发之前的数据。