Analog 代码分析

内容列表：

一、analog 的说明及结构

在 AG32 芯片中，analog 是个特殊的存在。
在芯片中，包含了 adc/dac/cmp 硬核，但硬核却没有和 mcu 直接相连。需要经过 cpld 的辅助，才能在 mcu 中顺利使用 adc/dac/cmp 的功能。

结构是：　（C 代码）mcu　<==>　cpld　<==>　AD/DA 硬核

analog 实现的就是这里的 cpld 部分。
这部分其实是个很好的学习样例，它为我们展现了 mcu 和 cpld 之间是如何搭配工作的。我们可以参考这部分的框架，更好的写出自己的 cpld 逻辑。
这部分代码在样例程序 examples\analog\ 目录下。

先来看这个 examples\analog 工程：

在 src 里边，是 mcu 这边对 adc 模块的调用。里边需关注 example_analog.c 和 analog_ip.h。
在 logic 里边，是 cpld 部分的实现。里边需关注 analog_ip.v 和 ahb2apb.v。

二、MCU 端的处理

整个 analog 包含了 ADC/DAC/CMP 三部分。三个部分会依次讲到。
先从 ADC 开始。

第一部分：驱动部分

在 analog_ip.h 文件中，先看对设备地址和寄存器的定义：

```
typedef struct
{
  __IO uint32_t CTRL;         // 0x00
  __IO uint32_t STAT;         // 0x04
  __IO uint32_t DATA;         // 0x08
  __IO uint32_t RESERVED[12]; // 0x0C - 0x38
  __IO uint32_t CHNL;         // 0x3C
  __IO uint32_t SEQ[16];      // 0x40 - 0x7C
} ADC_TypeDef;
#define ADC0 ((ADC_TypeDef *) 0x60000000)
#define ADC1 ((ADC_TypeDef *) 0x60001000)
#define ADC2 ((ADC_TypeDef *) 0x60002000)
```

从这里的定义可以看出，每个 ADC 有 6 个寄存器：CTRL、STAT、DATA、RESERVED、CHNL、SEQ。（这 6 个寄存器是 mcu 和 cpld 交互的接口）。
然后，共有 3 个 ADC（ADC0、ADC1、ADC2），访问地址分别是：0x60000000、0x60001000、和 0x60002000。
如果前边看过 user_manual.pdf 就会知道，0x60000000~0x80000000 之间是给 cpld 使用的地址空间。mcu 要访问 cpld，必须要通过以上地址才能触发 cpld 的信号。其实不止是这里的 AD/DA 模块，所有的 cpld 逻辑（包括用户实现的 cpld），如果要与 mcu 交互，也必须使用这个地址区间。

接下来，按照正常推断：
0x60000000 地址，是 ADC0 的 CTRL 寄存器；
0x60000004 地址，是 ADC0 的 STAT 寄存器；
0x60000008 地址，是 ADC0 的 DATA 寄存器；
0x6000003C 地址，是 ADC0 的 CHNL 寄存器；
在 MCU 端，往 0x60000000 地址写一个值，就是写到了“ADC0 的 CTRL 寄存器”。
在 MCU 端，读 0x60000004 地址的值，读到的就是“ADC0 的 STAT 寄存器”。

那么，就会带来几个疑问：
1. 为什么 ADC0 的 CTRL 寄存器地址是 0x60000000，可以是 0x70000000 吗？
2. MCU 端直接读写 0x60000000 这样的地址时，cpld 端会发生什么？
如果看过前边的《AG32 中 cpld 的基础》，就已经有答案了。如果没看过，带着疑问继续往下看。后边也会描述到。

继续看这几个寄存器的意义：

CTRL：写寄存器。mcu 对 ADC 控制的寄存器（里边包含多个控制项）。
STAT：读寄存器。mcu 读 ADC 当前状态的寄存器。
DATA：读寄存器。mcu 读取 ADC 转换完结果的寄存器。
CHNL：写寄存器。mcu 设置一轮采样多少个通道数；
SEQ：写寄存器。mcu 设置一轮采样多个通道时的通道序列。

其中：
CTRL 寄存器有 5 项：
ADC_CTRL_START：控制 ADC 开始启动；
ADC_CTRL_STOP：控制 ADC 停止工作；
ADC_CTRL_CONT：控制 ADC 连续工作；当该项 disable 时，采样一轮就自动停止。否则就会一轮一轮不间断采样，直到 mcu 主动来 STOP 时。
ADC_CTRL_DMAEN：是否使能 DMA；DMA 使能后，采样完**一个通道**就会触发一次 DMA 来搬数据（不是一轮完才触发）；
ADC_CTRL_SCLK_DIV：设置 ADC 的分频系数。是从 BUSCLK 基础上分频的。这里 ADC 的时钟频率应该小于 12M。
STAT 寄存器有 2 项值：
ADC_STAT_EN：是否 ADC 正在工作；
ADC_STAT_EOC：是否 ADC 已经转换完成；
DATA 寄存器的意义：
当 ADC 转换完成后，可以从这个寄存器中获取转换后的值。
1. 当配置 ADC 只读取一个通道时，等转换完后（ADC_STAT_EOC==1）直接来取就可以。
2. 当配置 ADC 一轮读取多个通道时，最好使用 DMA，每转换好一个值，用 DMA 自动搬运一次数据（搬运到 DMA 配置的 buff 中）。
CHNL 寄存器的意义：
配置 ADC 一轮读取多少个通道的值。
这里配置的值为：N-1。如，只读取一个通道这里值为 0；读取 2 个通道这里值为 1...
SEQ[]寄存器的意义：
配合 CHNL 寄存器使用。填充真正要采集的 channal id。
如果只读取 1 个通道的值，则填入 SEQ[0]；
如果读取 2 个通道的值，则填入 SEQ[0]和 SEQ[1]；
如果读取 3 个通道的值，则填入 SEQ[0]和 SEQ[1]、SEQ[2]...

这里多了解下“一轮”和“连续采样”的概念：
如果只配置一个通道采样，则“一轮”就是这一个通道采样完一次。
如果配置了多个通道采样，则“一轮”是把配置的所有通道全部采样完一次。
如果没有配置“连续采样”，则以上采样完一轮 ADC 就自动停止了。
如果配置了“连续采样”，则 ADC 启动后，会一轮一轮采样下去，直到主动去 STOP。
需要注意的是：每一次采样的值，都是放在 DATA 寄存器的。这个寄存器只能存一个值。所以，如果一轮中会采样多个通道，或者配置了连续采样，一般都需要配置 DMA 来自动搬运数据。（不然后边采样到的数据会冲掉前边缓存值）

再结合源码，继续看 analog_ip.h 头文件，只看 ADC 的全部定义（先不看 DAC 和 CMP）。
看完后会发现：

1. 每个 ADC 有 16 个通道；
2. 接下来的 ADC 函数封装，都是围绕以上的寄存器设定而进行的；

   理解了以上寄存器，其实也就理解了 ADC 的驱动函数：

```c
// ADC sample rate is: APB clock frequency / (1 + sclk_div) / 2 / 13
static inline void ADC_Start(ADC_TypeDef *adc, uint32_t sclk_div) { adc->CTRL = ADC_CTRL_
static inline void ADC_StartContinuous(ADC_TypeDef *adc, uint32_t sclk_div) { adc->CTRL =
static inline void ADC_StartDma(ADC_TypeDef *adc, uint32_t sclk_div) { adc->CTRL = ADC_CT
static inline void ADC_Stop(ADC_TypeDef *adc) { adc->CTRL |= ADC_CTRL_STOP; }
static inline void ADC_WaitForEoc(ADC_TypeDef *adc) { while (!(adc->STAT & ADC_STAT_EOC))
static inline void ADC_SetChannel(ADC_TypeDef *adc, ADC_ChannelNumTypeDef channel) { adc-
static inline void ADC_SetSeqLength(ADC_TypeDef *adc, ADC_SeqLengthTypeDef seq_length) {
static inline void ADC_SetSeqChannel(ADC_TypeDef *adc, uint32_t seq, ADC_ChannelNumTypeDe

static inline uint32_t ADC_GetStat(ADC_TypeDef *adc) { return adc->STAT; }
static inline uint32_t ADC_GetData(ADC_TypeDef *adc) { return adc->DATA; }
```

到这里，ADC 的 C 驱动部分全部看完。

接下来看 example_analog.c 中对 ADC 的使用举例。

第二部分：MCU 对 ADC 的应用：

在 example_analog.c 中，对 ADC 操作的样例有两个：

1. TestAdc 函数：这个函数中只设置一个通道进行采样。

   流程很简单：设置通道 -> 设置工作频率 -> 启动转换 -> 转换完成后读取数据。

   这个函数里，没有启动 DMA，也没有启动**连续转换**。只是单纯的一个通道采集一次。

   是对 ADC 最基础的应用。
2. TestDacAdc 函数：这个函数中配置了对两个通道进行 DMA 读取 N 轮的数据。

   流程：配置两个通道 -> 配置 DMA 读取 -> 启动 DMA -> 等待 N 轮读完 -> 停止 ADC。

   函数中的几点说明：

   A. 通过以下 3 个函数，先配置出来 2 个采样通道：

   ```c
   // Use ADC0 to sample both DAC0 and DAC1
   ADC_SetSeqLength(ADC0, ADC_SEQ_LENGTH2); // Scan 2 channels
   ADC_SetSeqChannel(ADC0, 0, ADC_CHANNEL4); // Output from DAC0
   ADC_SetSeqChannel(ADC0, 1, ADC_CHANNEL5); // Output from DAC1
   ```

   B. 配置 DMA 读取时，因为要读取 N（N=WAVE_TABLE_SIZE）轮，所以配置长度为：WAVE_TABLE_SIZE * 2。

   这个配置中，一轮是读 2 次数据，而 WAVE_TABLE_SIZE 轮，则共发生 DMA 搬运数据的次数就是 WAVE_TABLE_SIZE * 2 次。

   C. 启动 ADC 采样后，等待的是 DMA 搬运的结束。

   注意，这里不是等待 ADC 的结束（因为 DMA 启动时默认是“连续转换”的，ADC 不会自动结束），这里是等待 DMA 结束后再主动去停止 ADC。

三、cpld 端 analog_ip 描述及对 ahb2apb 的应用

在 examples\analog\logic 下，存在三个.v 文件：example_board.v、ahb2apb.v 和 analog_ip.v。

其中 example_board.v 是绑定工程自动生成的框架.v，不用去关注。只需要知道，这里是 cpld 的最初入口，这里会调用 analog_ip.v 里边的“用户逻辑入口”。
其中 ahb2apb.v 是 cpld 中 AHB 转 APB 的模块，认为是一个“转换库”即可，不需要去修改。
在芯片中，cpld 跟 mcu/ram 一样，是挂载在 AHB 总线上的。mcu 和 cpld 交互时，是通过 AHB 总线先到 cpld 模块的。然后 cpld 要实现的“外设”（如这里的 ADC），一般是低速设备，不能直接挂载在 AHB 下的。这个时候，需要一个 AHB 转 APB 的“桥”。这里的 ahb2apb.v 就是这个“桥”的功能。ahb2apb.v 这个“桥”的实现代码，可以不用太关注，参考样例会使用即可。

analog_ip.v 中才是真正用户需要关心的地方，这里是用户逻辑入口。

其实除了上述的 3 个.v，还有系统的 alta_sim.v。
Alta_sim.v 中会适配跟芯片相关的一些底层逻辑，对“用户层”提供一些“库函数”。
以上模块在《AG32 下 cpld 的使用入门.pdf》中有更详细描述。

在这个功能中，模块 analog_ip 是实现 ADC/DAC/CMP 的入口。
Mcu 对 0x60000000~0x7F000000 的读写操作，都是通过 AHB 到达 ananlog_ip 的接口的。
比如，mcu 要读 0x60000004 的寄存器：
*mcu 端直接 C 语言这样调用：int cpRdReg = *((int *)0x60000004);*
此时，ananlog_ip 接口的 mem_ahb_ 信号组就会被触发。然后在 analog_ip.v 里根据信号做出回应。这块具体详情请参考《AG32 下 cpld 的基础》。
在 analog_ip 中，实例化了 ahb2apb“桥”、实例化了 3 个 ADC 和 2 个 DAC，以及 CMP。
MCU 发过来的读写操作到达 analog_ip 后，会先通过“桥”转成 apb 信号，然后再发送给挂载在 apb 总线上的 ADC/DAC/CMP。
ADC/DAC/CMP 在收到信号时，如果确认是给自己的指令，就执行相对应的操作。
相应操作，最后都转化为对“硬核”的控制动作。

从具体代码看 analog_ip 的模块组成和调用：
```
----------------------------------------------------------------
用户逻辑模块（module analog_ip）//用户模块入口
      桥（ahb2apb）
      3 个 ADC 实例（apb_adc）
      2 个 DAC 实例（apb_dac）
      1 个 CMP 实例（apb_cmp）
endmodule

module apb_adc //adc 模块定义
   xxxx
endmodule

module apb_dac //dac 模块定义
```

xxxx
endmodule

module apb_cmp //cmp 模块定义
xxxx
endmodule

整个结构中看，analog_ip 模块包含了 7 个实例：ahb2apb、3 个 ADC、2 个 DAC、1 个 CMP。

接下来对 analog_ip 进行具体代码的分析。

module analog_ip 代码注解：

```
assign slave_ahb_hsel   = 1'b0;
assign slave_ahb_hready = 1'b1;
assign slave_ahb_htrans = 2'b0;
assign slave_ahb_hsize  = 3'b0;
assign slave_ahb_hburst = 3'b0;
assign slave_ahb_hwrite = 1'b0;
assign slave_ahb_haddr  = 32'b0;
assign slave_ahb_hwdata = 32'b0;
```

上边这部分是对 cpld 做为 master 端使用时的信号线赋值。表示这个模块不会有 master 的行为（都是 MCU 主动来操作 cpld 的）。

```
// assign ext_dma_DMACBREQ   = 4'b0;
assign ext_dma_DMACLBREQ = 4'b0;
assign ext_dma_DMACSREQ  = 4'b0;
assign ext_dma_DMACLSREQ = 4'b0;
assign local_int         = 4'b0;
```

这一段，依然是对不会操作到的对外信号设置初值。

```
parameter ADDR_BITS = 16;
parameter DATA_BITS = 32;
```

这里，定义 地址线位宽 和 数据线位宽 分别为 16 位和 32 位。
后续代码中定义的 wire 连线数组，都以这个常量为宽度来进行定义。
地址总线为 16 位宽，也就是说，cpld 中只识别 0 ~ ‘hFFFF 的地址范围。
比如，mcu 端定义的 0x60001004 这个地址，在 cpld 端目前只使用了 0x1004。
也就是说，mcu 端访问 0x60001004 和访问 0x70001004，对 cpld 来说是一样的，都是 0x1004。

```
parameter PER_BITS = 12;
parameter PER_CNT  = 6;
parameter ADC0_ADDR = 'h0000;
parameter ADC1_ADDR = 'h1000;
parameter ADC2_ADDR = 'h2000;
parameter DAC0_ADDR = 'h3000;
parameter DAC1_ADDR = 'h4000;
parameter CMP0_ADDR = 'h5000;
```

常量 PER_CNT=6，表示这里定义的“外设”个数为 6 个：3 个 ADC + 2 个 DAC + 1 个 CMP。
常量 PER_BITS=12，表示 cpld 的外设的“片内”寻址空间为 12 位。即 2^12=0x1000（4K）。
也就是说，每个 ADC 可用的寄存器数量有 1K 个。

接着看 6 个“外设”的地址，每个外设间的地址跨度刚好就是 4K 大小。
mcu 端访问 ADC1（0x60001000），对应的就是这里的 ADC1_ADDR（‘h1000）。
注意：ADC1_ADDR 常量并没有在代码中出现，而是在 mem_ahb_haddr[ADDR_BITS-1:0]赋值给 ahb_haddr，只取 16 位时体现出来的。经过 ahb2apb“桥”之后，最终这个 16 位地址给到 apb_paddr。*即：mcu 访问 0x60001000，到 apb_paddr 拿到的就是 0x1000*。而后续的“片选”、“寄存器偏移”正是从 apb_paddr 来开始的。

```
wire                   apb_psel;
wire                   apb_penable;
wire                   apb_pwrite;
wire [ADDR_BITS-1:0]   apb_paddr;
wire [DATA_BITS-1:0]   apb_pwdata;
wire [3:0]             apb_pstrb;
wire [2:0]             apb_pprot;
wire                   apb_pready  = 1'b1;
wire                   apb_pslverr = 1'b0;
reg  [DATA_BITS-1:0]   apb_prdata;
```

这里定义的是 apb 的信号。
就是 mcu 过来的信号，先到 ahb，再经过“ahb2apb 桥”转换后的 apb 信号。
apb 的各信号意义不再描述，不熟悉的自行百度。
注意这里的地址线（apb_paddr）是 16 位宽，读和写的信号线是 32 位宽。

```
assign apb_clock = bus_clock;
```

这里是设置 apb_clock 为 bus clock。
而 bus_clock，就是在 VE 里定义的定义的 BUSCLK 字段，如图：

```
SYSCLK 100
BUSCLK 50
```

如果 VE 里没有定义 BUSCLK，则 bus_clock 和 sys_clock 同频。
（目前样例程序中，并没有定义 BUSCLK。如果提高主频 SYSCLK 后发现 ADC 跑不起来，可以单独定义该 BUSCLK，确保分频后的 ADC 小于 12M）。

```
ahb2apb #(ADDR_BITS, DATA_BITS) ahb2apb_inst(
    .reset          (!resetn                    ),
    .ahb_clock      (sys_clock                  ),
    .ahb_hmastlock(1'b0                         ),
    .ahb_htrans     (mem_ahb_htrans             ),
    .ahb_hsel       (1'b1                       ),
    .ahb_hready     (mem_ahb_hready             ),
    .ahb_hwrite     (mem_ahb_hwrite             ),
    .ahb_haddr      (mem_ahb_haddr[ADDR_BITS-1:0]),
    .ahb_hsize      (mem_ahb_hsize              ),
    .ahb_hburst     (mem_ahb_hburst             ),
    .ahb_hprot      (4'b0011                    ),
    .ahb_hwdata     (mem_ahb_hwdata             ),
    .ahb_hrdata     (mem_ahb_hrdata             ),
    .ahb_hreadyout (mem_ahb_hreadyout           ),
    .ahb_hresp      (mem_ahb_hresp              ),
    .apb_clock      (apb_clock                  ),
    .apb_psel       (apb_psel                   ),
    .apb_penable    (apb_penable                ),
    .apb_pwrite     (apb_pwrite                 ),
    .apb_paddr      (apb_paddr                  ),
    .apb_pwdata     (apb_pwdata                 ),
    .apb_pstrb      (apb_pstrb                  ),
    .apb_pprot      (apb_pprot                  ),
    .apb_pready     (apb_pready                 ),
    .apb_pslverr    (apb_pslverr                ),
    .apb_prdata     (apb_prdata                 )
);
```

这段代码，是 ahb2apb 的实例，“桥”。

ahb 的信号经过“桥”的转换后，都到 apb_xxx 来给“外设”使用。

而 ADC/DAC/CMP，也正是基于 apb_xxx 而实现的。

所以说：ADC/DAC/CMP 是挂载在 apb 线上的外设，而不是直接挂载在 ahb 线上的外设。

```
wire [PER_CNT-1:0] select = 1 << (apb_paddr[ADDR_BITS-1:PER_BITS]);
```

这里定义了 6 位宽的 select（分别对应 3 个 ADC、2 个 DAC、1 个 CMP）。

可以认为是“片选”（6 位中只能有 1 位是 1，也就是当前“选中”要操作的外设）。

代码解读上，apb_paddr[ADDR_BITS-1:PER_BITS] 是取了 apb_paddr 的高 4 位（即：0x60001000 中为 1 的那个数值），该值就是前边描述的“片选”。从这个片选能够区分，mcu 过来的信号，到底是要访问 6 个外设中的哪个。

这里的等号赋值，是设置 select 的 6 位中的某一位为 1。

```
wire                    per_psel[PER_CNT];
wire                    per_penable[PER_CNT];
wire                    per_pwrite[PER_CNT];
wire [PER_BITS-1:0]     per_paddr[PER_CNT];
wire [DATA_BITS-1:0]    per_pwdata[PER_CNT];
wire [DATA_BITS-1:0]    per_prdata[PER_CNT];
```

常量 PER_CNT 就是 6（表示 3 路 ADC、2 路 DAC、1 路 CMP）。

那么，这里定义了 6 路外设，各自的几个信号：

per_psel：该外设是否被选中；

per_penable：该外设是否被使能；

per_pwrite：该外设写信号线（0：读；1：写）；

per_paddr：要操作的外设的片内偏移（16 位）；

per_pwdata：要写的数据（32 位）；

per_prdata：要读的数据（32 位）；
注意，这些信号在后续实例化外设时，会被分别连接到 6 个外设上。
这些信号线都是同源（来自于 apb_xxx 信号组）且并联起来的，然后由 select[i]片选来决定哪个外设最终 psel 为 1.
想象下：apb_xxx 信号组出来后，每个信号线分成了 6 支，分别接入到 6 个外设。
也就是说，当 mcu 的读或写命令过来时，信号会同时到达 6 个外设。但 6 个外设中只有一个被片选选中。那么被片选选中的这个外设，才真正开始执行动作。

```
wire [PER_CNT-1:0] dma_req;
wire [PER_CNT-1:0] dma_clr;
```

DMA 的两个信号线（CMP 没做 DMA 功能）。
dma_req：当数据准备好，并且 dma 开启时，cpld 通过这个信号线来通知 DMA；
dma_clr：当 dma 搬完数据后，DMA 通过这个信号来通知 cpld。

```
// EXT_DMA0: ADC0
// EXT_DMA1: ADC1
// EXT_DMA2: ADC2 + DAC1 (shared)
// EXT_DMA3: DAC0
assign ext_dma_DMACBREQ  = dma_req[3:0] | (dma_req[4] << 2);
assign dma_clr           = {ext_dma_DMACCLR[2], ext_dma_DMACCLR[3:0]};
```

这里和上边的 dma 两个信号（向外请求 req 和向内清除 clr）是关联的。
ext_dma_DMACBREQ：这个就是 ahb 线上，cpld 对外 dma 请求的信号线。当 5 个设备（不含 cmp）有 dma 请求信号时，都会通过 ext_dma_DMACBREQ 传递出去给 DMA。
注意：如上边注释所标注，ADC2 和 DAC1 是共用的。
所以才有表达式：ext_dma_DMACBREQ = dma_req[3:0] | (dma_req[4] << 2) 里边的或。
（注意，数组序列为 0:ADC0 1:ADC1 2:ADC2 3:DAC0 4:DAC1 5:CMP）
dma_clr：这个信号组，源于 ahb 总线上 ext_dma_DMACCLR 过来的信号的重组。
重组表达式，与 ext_dma_DMACBREQ 相反，把 4 个信号拆分为 5 个信号。
注：由于 DMA 中 ADC2 和 DAC1 的共用和 CMP 的缺失，信号从 AHB 过来的，跟这里的数组存在一个重组的过程。上边的两个表达式就是正向和反向的重组表达式。

```
genvar i;
generate
  for (i = 0; i < PER_CNT; i = i + 1) begin : gen_per
    assign per_psel[i]    = apb_psel    & select[i];
    assign per_penable[i] = apb_penable & select[i];
    assign per_pwrite[i]  = apb_pwrite;
    assign per_paddr[i]   = apb_paddr[PER_BITS-1:0];
    assign per_pwdata[i]  = apb_pwdata;
    if (i < 3) begin : gen_adc
      apb_adc adc_inst(
    end else if (i < 5) begin : gen_dac
      apb_dac dac_inst(
    end else begin : gen_cmp
      apb_cmp cmp_inst(
    end
  end
endgenerate
```

上边的过程，就是实例化 6 个外设。
想象一下：生成 6 个外设模块，并把前边定义好的 6 组信号（从 apb_xxx 过来的）分别接入到 6 个外设中去。

```
reg [PER_CNT-1:0] pr_select; // These extra registers
always @ (posedge apb_clock or negedge resetn) begin
  if (!resetn) begin
    pr_select <= 'b0;
  end else if (apb_psel && !apb_penable) begin
    pr_select <= select;
  end
end
```

这里 pr_select 是定义了一组额外的片选寄存器。

pr_select 仅同步记录 select 信号组（相当于 select 的缓存）；

pr_select 仅用于后边紧跟的 apb_prdata 处理。

```
integer j;
always @ (*) begin
  apb_prdata = 0;
  for (j = 0; j < PER_CNT; j = j + 1) begin
    apb_prdata = apb_prdata | (pr_select[j] ? per_prdata[j] : 0);
  end
end
```

这里是把当前选中的外设的读信号 per_prdata 的值，记录到 apb_prdata 寄存器中。（以供 ahb2apb 桥，继续传递给 ahb_prdata，最终在读周期内传递给 mcu）

这里的 for 循环中，其实 6 次执行中，只有 1 次 pr_select[i]是有效的（也就是当前片选选中的那个外设），只有这一次会取到 per_prdata 有效的值。

所以，这里的 apb_prdata 总是会实时缓存“当前使能的外设的读信号的值”。

也就是说，这段代码保证了：mcu 来读 6 个外设中的一个的数据时，**数据能从外设 read 信号线传递到 apb 的 read 信号线**，继而再向上传递。

到这里，analog_ip 模块已经描述完了。

可以看到：

Mcu 写数据时，信号通过 AHB 到来后，先通过 ahb2apb 转成 apb_xxx 信号，然后再分 6 组传给 6 个外设（片选决定了哪个使能）；

Mcu 读数据时，信号通过 AHB 到来后，仍是先通过 ahb2apb 转成 apb_xxx 信号，然后再分 6 组传给 6 个外设（片选决定了哪个使能），只是 apb_prdata 会实时获取外设的 prdata，再往回传递。

扩展 1：

如果不需要 3 个 ADC、2 个 DAC、1 个 CMP 里边的全部（比如只需要一个 1 个 ADC），那么，就把不需要的实例去掉即可。

这里是 for 循环生成实例的，也可以不用 for 循环直接实例化的。

扩展 2：

如果要实现自己的“apb 外设”，只需要跟这里的 adc/dac/cmp 一样，生成一个实例，挂载在 apb 下边即可。挂载在 apb 下边后，跟 adc 一样，把必要的信号引入到自己的外设，然后在自己的模块逻辑中影响这些信号即可。

唯一要注意的是，自己新增外设的地址，跟上述的分开即可（自己新增外设也可以建立像上边“片选”的方式）。

网盘上有个样例《6.UartTx 例程》，展示了如何自建一个模块挂载在 apb 下，可以参考。

四、CPLD 端对 ADC 的处理

接下来，看具体的 ADC 模块。

前边提到过，芯片里本身是包含了 ADC 硬核的。但 mcu 并没有和硬核直接相连，而是通过这里的 cpld 来连接的。
那么，硬核 ADC 部分的信号接口，从这里的调用看，就是 alta_adc 部分：

```
alta_adc adc_inst(
    .enb   (!adc_en ),
    .sclk  (sclk    ),
    .insel (chnl_sel),
    .stop  (stop    ),
    .db    (adc_db  ),
    .eoc   (adc_eoc )
);
```

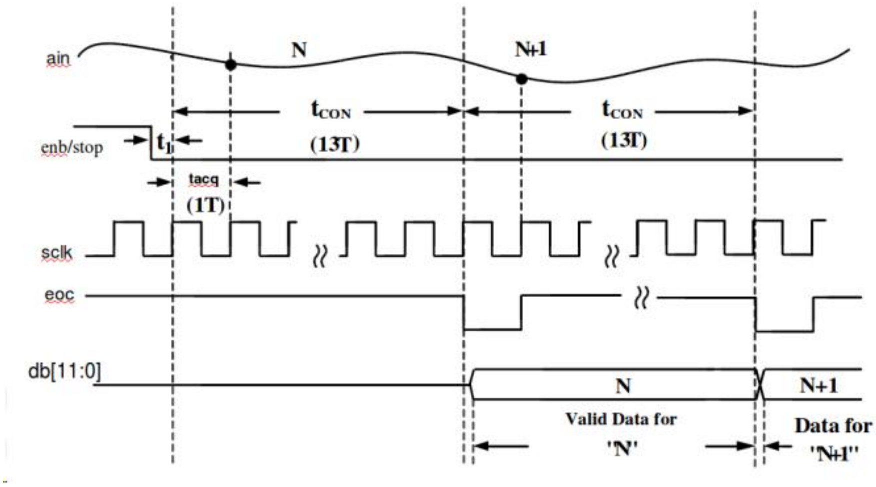
关于 alta_adc 的接口定义，从 alta_sim.v 中可以看到：

```
module alta_adc (
    input          enb,
    input          sclk,
    input          stop,
    input   [4:0]  insel,
    output  [11:0] db,
    output         eoc
);
```

先看这里的接口很有意义，因为 cpld 里绕来绕去，最终要调用的就是这个硬核接口。
也就是说，从 mcu 到 cpld 里的全部操作，最终都是要转换为对这里接口的操作的。采样的数据也来源于这里。

先看以上接口的几个信号的意义：（这部分可以从 Reference Manual 手册上获取）
enb：设置 ADC 是否使能 （0: enable, 1: disable）
stop：是否开启 ADC 采样（0: disable，1: enable 即，关闭 adc）
Insel：采样哪个通道；
db：（output）12 位采样到的数据；
eoc：（output）采样完成的信号，一个 clk 周期。
使用上，insel 设定好，输入 sclk，enb 和 stop 都拉低时，进入 ADC 采样。在采样结束后，会输出 eoc（宽度为一个时钟周期）的低电平，在这个周期的下降沿开始，外部可以读取硬核内转换后的值（这个值会保留到下次采样结束）。

Reference Manual 手册（文件内搜索 ADC）上对 ADC 时序描述如下：



接下来，继续看 ADC module，看模块内如何处理 apb_xxx 信号的输入，将信号转换后操作上边的 alta_adc。

在具体代码分析时，注意里边的 3 个周期：

1. apb 的 clk 时钟周期；
2. apb 分频后传给 adc 的 clk 时钟周期；
3. adc 中采样一次数据的时间（采样一次完整的 ADC 数据，需要 12+1 个 adc_clk 周期）。

其实这 3 个周期也是理解以下代码的基础。不理清这 3 个层面，代码是无法完全理解的。后边讲解中，这 3 个周期会经常出现。

进入 module apb_adc 的模块：

```
module apb_adc (
  input          stop,
  output         dma_req,
  input          dma_clr,
  input          apb_clock,
  input          apb_resetn,
  input          apb_psel,
  input          apb_penable,
  input          apb_pwrite,
  input   [11:0] apb_paddr,
  input   [31:0] apb_pwdata,
  output  [31:0] apb_prdata
);
```

相比前边模块，这里 ADC 模块的输入输出信号线已经很少了。几个信号线看名字，跟前边也是相仿的，不再描述。

```
parameter SCLK_BIT = 16;
parameter SEQ_MAX = 16;
```

常量 SCLK_BIT，定义用多少位记录 ADC 的分频数；
常量 SEQ_MAX，定义最多支持多少个通道按序列采样；

```
parameter ADDR_CTRL = 'h00;
parameter ADDR_STAT = 'h04;
parameter ADDR_DATA = 'h08;
parameter ADDR_CHNL = 'h3C;
parameter ADDR_SEQ0 = 'h40;
```

对应到 MCU 端使用的 ADC 寄存器列表。mcu 端如下：

```
typedef struct
{
  __IO uint32_t CTRL;           // 0x00
  __IO uint32_t STAT;           // 0x04
  __IO uint32_t DATA;           // 0x08
  __IO uint32_t RESERVED[12]; // 0x0C - 0x38
  __IO uint32_t CHNL;           // 0x3C
  __IO uint32_t SEQ[16];        // 0x40 - 0x7C
} ADC_TypeDef;
```

```
wire is_seq_addr = apb_paddr[11:6] == (ADDR_SEQ0 >> 6);
wire [3:0] seq_idx = apb_paddr[5:2];
```

第一个表达式里，

这里(ADDR_SEQ0 >> 6) 的常量代入值后为 0x40>>6，计算后为 1。

apb_paddr[11:6] 是取 apb_paddr 的 6~11 位。比如 mcu 的 0x60001008，这里取到 0；

而 0x60001040~0x6000107F 之间，这里取到 1。

从 mcu 寄存器定义看，0x60001040 正是 adc1->SEQ 数组开始的地址。

所以，is_seq_addr 这个值的意义，是当前动作是否为读/写通道（adc channal）；

第二个表达式里，

seq_idx = apb_paddr[5:2]，为读取地址后再除以 4。

用值带入下，读取地址是 0x60001040 时，seq_idx 为 0；读取地址是 0x60001044 时，seq_idx 为 1；...

所以，seq_idx 这个值的意义，是要读 adc->SEQ 数组里的哪个位置。后续用做数组下标。

```
reg adc_en;
reg apb_eoc;
reg [11:0] apb_db;
wire adc_eoc;
wire [11:0] adc_db;
wire eoc_rising = !adc_eoc && !apb_eoc;
wire apb_data_phase = apb_psel && apb_penable;
```

这里是定义几个寄存器和信号。

请注意这个.v 代码中的寄存器命名方式。adc_开头的，都是跟硬核 ADC 相关。

adc_en：adc 硬核是否使能（使能情况下 ADC 才正常工作）。

注意：该寄存器是直接连接到硬核接口的。用来控制硬核 ADC 是否使能。

当 mcu 端调用 ADC_Start 时，改变 ctrl_adc_start 后，会改变该值为 1；

当 mcu 端调用 ADC_Stop 时，或者 ADC 一轮采样完后，会改变该值为 0；

adc_eoc：信号线，连接 ADC 硬核的 eoc 信号；

这个信号平时为高电平，在采样完后有一个 adc clk 周期的低电平。

adc_db：信号线，12 位宽，连接 ADC 硬核的 db 信号；

当采样完数据，adc_eoc 变低后，该信号线上的数据有效，可以往外读取。

apb_eoc：寄存器，用于记录 adc_eoc 状态。

它和 adc_eoc 信号宽度一样都是 1 个 adc_clk 周期，但变化比 adc_eoc 晚一个节拍。

并且，它和 adc_eoc 高低刚好是反的。它正常为低，触发时为高。

这个寄存器值为 1，可以表示采样完成（并且数据已经放到 apb_db 寄存器）。

apb_db：寄存器，用于记录 adc_db 的值；

即：缓存 ADC 采样到的值。直到下个数据来时，才冲掉上个缓存值。

每次采样完，在 adc_eoc 变化的周期里，就会把 adc_db 的值写入到这个寄存器。

eoc_rising：eoc 上升沿，用以标记硬核 ADC 一次数据刚采样完。

这个标记的长度也是 1 个 adc_clk 周期。就是 adc_eoc 已经变低但 apb_eoc 还没变高的那个 clk 周期，跟 adc_eoc 是重合（但高低是反的）。

apb_data_phase：当前 ADC 是否被选中并且使能（是否可正常操作该外设）。

当该信号为 1 时，该 ADC 外设可以处理 apb 过来的读写命令。

```
reg ctrl_adc_start;          // Write 1 to set.
reg ctrl_adc_stop;           // Write 1 to stop
reg ctrl_adc_cont;           // Continous mode
reg ctrl_adc_dmaen;          // DMA enable
reg [SCLK_BIT-1:0] ctrl_sclk_div; // Sclk is
`define ADC_CTRL_REG {ctrl_sclk_div, 12'b0,
```

ctrl_adc_start、ctrl_adc_stop、ctrl_adc_cont、ctrl_adc_dmaen、ctrl_sclk_div：这几个寄存器就是记录 mcu 端设置过来的值。

其中，ctrl_adc_start 会自动被置零（在 stop 被设置时，或在采样完成时<非连续模式下>）。

ADC_CTRL_REG 定义了 CTRL_REG 寄存器的数据格式，对应 MCU 端的 adc->CTRL。

```
reg stat_adc_eoc;
`define ADC_STAT_REG {stat_adc_eoc, adc_en}
```

stat_adc_eoc：表示一轮采样是否结束。

它和 apb_eoc 的区别是：

apb_eoc 是实时记录真正硬核采样的状态的，只不过比 adc_eoc 晚一个节拍；每采样一个通道就会产生一次信号。

stat_adc_eoc 是给 MCU 反馈状态用的寄存器。当一轮数据（可能是一个数据也可能是多个数据，看配置的通道号而定）全部采样完，才会产生一次变化。（搜关键字 stat_adc_eoc）看下文中的更多描述。

ADC_STAT_REG 定义了 STAT_REG 寄存器的数据格式，对应 MCU 端的 adc->STAT。

```
reg seq_last;
reg [3:0] seq_length;
reg [4:0] seq_reg [SEQ_MAX];
reg [3:0] seq_cnt;
wire [4:0] chnl_sel;
`define ADC_CHNL_REG {chnl_sel, seq_cnt, seq_length}
```

seq_last：是否为一轮采样中的最后一个；

seq_length：记录 mcu 设置过来的一轮的采样通道数；

seq_reg：记录 mcu 设置过来的一轮采样的通道编号数组，数组长度为 16；

seq_cnt：当前这轮已经采样了多少个（类似：运行中的局部变量）；

chnl_sel：当前正在采样的是哪个通道；

ADC_CHNL_REG 定义了 CHNL_REG 寄存器的数据格式，对应 MCU 端的 adc->CHNL。

```
wire seq_done = apb_eoc && seq_last;
wire adc_stop = ctrl_adc_stop || (seq_done && !ctrl_adc_cont);
```

seq_done：是否一轮中的最后一个通道的数据已经采样完；
（因为 apb_eoc 表示一次采样结束，seq_last 表示一轮中的最后一次采样）
adc_stop：adc 是否为停止状态（当 mcu 发来了 stop 命令，或，非连续模式下的一轮采样完）

```
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    ctrl_adc_start <= 1'b0;
    ctrl_adc_stop  <= 1'b0;
    ctrl_adc_cont  <= 1'b0;
    ctrl_adc_dmaen <= 1'b0;
    ctrl_sclk_div  <= 'b0;
  end else if (apb_data_phase && apb_pwrite && apb_paddr == ADDR_CTRL) begin
    ctrl_adc_start <= apb_pwdata[0] | ctrl_adc_start;
    ctrl_adc_stop  <= apb_pwdata[1] | ctrl_adc_stop;
    ctrl_adc_cont  <= apb_pwdata[2];
    ctrl_adc_dmaen <= apb_pwdata[3];
    ctrl_sclk_div  <= apb_pwdata[31:32-SCLK_BIT];
  end else begin
    if (adc_stop) begin
      ctrl_adc_start <= 1'b0;
      ctrl_adc_stop  <= 1'b0;
    end
  end
end
```

记录 mcu 设置过来的寄存器的值。
当复位时，这几个寄存器清 0；
当 mcu 设置 CTRL 寄存器时，cpld 中对这几个寄存器进行赋值；
当 adc 要停止时（不管是以上哪种原因），重新设置 start 和 stop 寄存器的值，等待 mcu 的新指令。

```
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    apb_eoc <= 1'b0;
  end else begin
    apb_eoc <= !adc_eoc;
  end
end
```

参考上边对 apb_eoc 和 adc_eoc 意义的描述。
这里就是设置 apb_eoc 比 adc_eoc 慢一个 clk，且高低反转的地方。

```
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    apb_db <= 12'b0;
  end else if (adc_en && eoc_rising) begin
    apb_db <= adc_db;
  end
end
```

围绕 apb_db 的处理。
apb_db 里记录了当前的采样值。
在 eoc_rising 标记使能时，从 adc_db 信号线上读采样值，存储在 apb_db 寄存器上。
这里的 eoc_rising 是在 adc_eoc 变化时（apb_eoc 变化前）就开始变化了。

```verilog
reg adc_restart;
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    adc_restart <= 'h0;
  end else if (apb_data_phase && apb_pwrite && (apb_paddr == ADDR_CHNL || is_seq_addr))
    adc_restart <= 1'b1;
  end else if (adc_restart) begin
    adc_restart <= 1'b0;
  end
end
```

围绕 adc_restart 的处理。

判断式二，当（该外设使能 且 mcu 来写时 且 （写的地址是通道数 或 写地址是通道号））时，adc_restart 设置为 1。

判断式三，当 adc_restart 被置 1 后，下个周期就恢复为 0。

也就是说，当 mcu 来写通道数/通道号时，adc_resart 寄存器会被置 1。然后下个 clk 置回 0。

即：当该 ADC 被重置通道时，adc_restart 有一个 apb_clk 的周期被置 1。

adc_restart 存在的意义，是后续来自动重置硬核的 adc_en 信号。

```verilog
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    stat_adc_eoc <= 1'b0;
  end else if (apb_data_phase && apb_pwrite && apb_paddr == ADDR_STAT && apb_pwdata[0] == 1'b0)
    stat_adc_eoc <= 1'b0;
  end else if (apb_data_phase && !apb_pwrite && apb_paddr == ADDR_DATA) begin
    stat_adc_eoc <= 1'b0;
  end else if (seq_done) begin
    stat_adc_eoc <= 1'b1;
  end
end
```

这里是围绕寄存器 stat_adc_eoc 的处理。

stat_adc_eoc 是反馈给 mcu，表示是否转换完成的标记。

判断式二，当（该外设使能 且 mcu 来写时 且 写地址是状态 且 写内容是 adc_en 赋 0）

即：mcu 端通过 adc->STAT 写 ADC_STAT_EN 位为 0 时。

判断式三，当（该外设使能 且 mcu 来读时 且 读地址是 DATA）

即：mcu 端通过 adc->DATA 来读 ADC 的值时。

判断式四，当（一轮中的最后一个通道已经采样完）

也就是说，stat_adc_eoc 这个寄存器只有在一轮中的最后一个通道已经采样完时，会被置 1，然后一旦有 mcu 读 ADC 数据的动作，或者 MCU 来重置 adc_en 的动作，stat_adc_eoc 被自动重置为 0。

```verilog
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    seq_length <= 'h0;
  end else if (apb_data_phase && apb_pwrite && apb_paddr == ADDR_CHNL)
    seq_length <= apb_pwdata[3:0];
  end
end
```

这里的处理，就是把 mcu 设置过来的通道数（adc->CHNL）写入 seq_length 寄存器。

```
genvar i;
generate
  for (i = 0; i < SEQ_MAX; i = i + 1) begin : gen_seq
    always @ (posedge apb_clock or negedge apb_resetn) begin
      if (!apb_resetn) begin
        seq_reg[i] <= 'h0;
      end else if (apb_data_phase && apb_pwrite && is_seq_addr && seq_idx == i)
        seq_reg[i] <= apb_pwdata[4:0];
      end
    end
  end
endgenerate
```

当 mcu 端设置通道时（adc->SEQ[seq] = channel），触发这里的赋值操作。
寄存器 seq_reg 数组存储的是 mcu 设置进来的通道号（最多 16 个）。

```
wire adc_seq_next;
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    seq_last <= 1'b0;
  end else if (apb_eoc) begin
    seq_last <= 1'b0;
  end else if (adc_seq_next && seq_cnt == seq_length) begin
    seq_last <= 1'b1;
  end
end
```

这里是对寄存器 seq_last 的处理。
判断式二，apb_eoc 表示当前次采样是否完成。
判断式三，adc_seq_next 的意义是“ADC 采样已经进行一半”，seq_cnt == seq_length 是指当前正在操作的通道就是这一轮的最后一个通道。
所以，seq_last 意义是：当前操作的通道是否为本轮的最后一个通道。
adc_seq_next 是采样中的一个时间点，搜关键字 adc_seq_next 看下文的详细描述。

```
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    seq_cnt <= 0;
  end else if (!ctrl_adc_start || seq_last) begin
    seq_cnt <= 0;
  end else if (adc_seq_next) begin
    seq_cnt <= seq_cnt + 1;
  end
end
```

这里是设置寄存器 seq_cnt。寄存器 seq_cnt 的意义是当前正在采样的通道 id。
判断式二，当 ADC 停止或者一轮已经处理完时，seq_cnt 被设置为 0；
判断式三，当一轮还没有处理完且“可以处理接下来的事情时”，seq_cnt 累加；

```
wire [31:0] ctrl_read = apb_paddr == ADDR_CTRL ? `ADC_CTRL_REG : 0;
wire [31:0] stat_read = apb_paddr == ADDR_STAT ? `ADC_STAT_REG : 0;
wire [31:0] chnl_read = apb_paddr == ADDR_CHNL ? `ADC_CHNL_REG : 0;
wire [31:0] data_read = apb_paddr == ADDR_DATA ? apb_db        : 0;
wire [31:0] seq_read  = is_seq_addr ? seq_reg[seq_idx] : 0;
```

这里是定义出来反馈给 mcu 各个寄存器的数据，对应 mcu 的几个寄存器。
这种表达方式较为便捷。

```
reg [31:0] prdata;
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    prdata <= 32'h0;
  end else if (apb_psel && !apb_penable && !apb_pwrite) begin
    prdata <= ctrl_read | stat_read | chnl_read | data_read | seq_read;
  end
end
assign apb_prdata = prdata;
```

这里是 mcu 来读寄存器数据时，数据反馈的组织表达式。

这里利用了：一次读取只能有一个地址。所以，一串“或”表达式里，只会有一个有值。

```
reg [3:0] adc_state;
// Pause sclk_en when DMA is in action
wire sclk_en = ctrl_adc_start && (ctrl_adc_cont || !stat_adc_eoc) &&
               (!dma_req && !dma_clr || adc_state < 10);
reg [SCLK_BIT-1:0] sclk_counter;
reg sclk;
```

寄存器 adc_state 的值，表示在当前这次采样中的第几个 adc_clk 周期（ADC 采样一次数据需要 12 个 adc_clk 周期，然后 eoc 还要额外一个 adc_clk 周期）。

寄存器 sclk，这是要输入到 ADC 硬核的 clk 时钟。输入给硬核的时钟，是在当前 apb_clk 基础上分频。分频的参数由 mcu 来设置为 ctrl_sclk_div。

寄存器 sclk_counter，是处理 apb 到 adc 分频用的。从 0 ->累加到分频值（产生一次 sclk 反转）->归 0 ->累加到分频周期（产生一次 sclk 反转）...，不断循环。

信号线 sclk_en，则表示是否给 ADC 硬核产生 clk 波形。

信号线 sclk，是直连硬核的 clk 信号线。会输出给硬核 ADC 的时钟波形。

```
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    sclk_counter <= 0;
    sclk <= 1'b0;
  end else if (!adc_en) begin
    sclk_counter <= 0;
    sclk <= 1'b0;
  end else if (sclk_en) begin
    if (sclk_counter == ctrl_sclk_div) begin
      sclk_counter <= 0;
      sclk         <= !sclk;
    end else begin
      sclk_counter <= sclk_counter + 1;
    end
  end
end
```

这段处理的就是 apb 到 adc 的 clk 的分频。

adc_clk 是从 apb_clk 上分频的，adc_clk 的宽度是 apb_clk 宽度的 N 倍（N=分频数，即 N=ctrl_sclk_div）

sclk 会连接硬核的 clk 时钟线。这里的高低反转，就是为硬核 ADC 产生 clock 时钟。

```
wire sclk_rising = sclk_en && !sclk && sclk_counter == ctrl_sclk_div;
```

信号 sclk_rising，如果值为 1，表示 adc_clk 的上升沿发生了。

注：adc_clk 是从 apb_clk 上分频的，adc_clk 的宽度是 apb_clk 宽度的 N 倍（N=分频数）。

这个信号会在 adc_clk 从低翻转到高时发生，宽度为 1 个 apb_clk 周期。

```
assign adc_seq_next = adc_state == 4 && sclk_rising;
```

adc_state 是计数当前正在一次 ADC 采样中的第几个 adc_clk。

那么 (adc_state==4)则表示，在 ADC 采样中的第四个 adc_clk 时的这个时间点。

所以，adc_seq_next 表示在 ADC 采样中的第四个 adc_clk 时且 adc_clk 处于上升沿的这个时间点。这个宽度为 1 个 apb_clk 周期。

adc_seq_next 这个本身没有意义，只是表示个时间点，具体要看用到它时的上下文语境。一般它被用于“可以处理接下来的事情了”。

```
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    adc_state <= 0;
  end else if (!adc_en || apb_eoc) begin
    adc_state <= 0;
  end else if (sclk_rising) begin
    adc_state <= adc_state + 1;
  end
end
```

这里是处理 adc_state 的。

寄存器 adc_state 的值，表示当前正在一次 ADC 采样中的第几个 adc_clk。（注：一个完整的 ADC 的采样周期，是 13 个 adc_clk）

判断式三中的 sclk_rising 是 adc_clk 发生上升沿时会被置 1 的（宽度为 1 个 apb_clk）。

```
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    adc_en <= 1'b0;
  end else if (adc_stop || adc_restart) begin
    adc_en <= 1'b0;
  end else if (ctrl_adc_start) begin
    adc_en <= 1'b1;
  end
end
```

这里处理的 adc_en，表示 adc 硬核是否使能（使能情况下 ADC 才正常工作）。

注意：该寄存器是直接传递到硬核接口的。用来控制硬核 ADC 是否使能。

处理的逻辑：

当 mcu 端调用 ADC_Start 时，改变 ctrl_adc_start 后，会改变该值为 1；

当 mcu 端调用 ADC_Stop 时，或者 ADC 一轮采样完后，会改变该值为 0；

```
reg dma_reg;
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    dma_reg <= 1'b0;
  end else if (!ctrl_adc_dmaen || dma_clr) begin
    dma_reg <= 1'b0;
  end else if (ctrl_adc_dmaen && eoc_rising) begin
    dma_reg <= 1'b1;
  end
end
assign dma_req = dma_reg;
```

这里是处理 DMA 的地方。

如果设置了 DMA 使能，则每个 ADC 采样完后的上升沿，都会触发一次 DMA 请求。

等 DMA 来读取数据后，发送 dma clear 信号时，清除该 DMA 请求信号。

cpld 对外的 DMA 信号线只有两条：对外请求 req 和对内清除 clr。处理过程就在这里。

到这里，ADC 模块的代码也全部分析完毕。
回顾一下 cpld 里做的任务，包括：存储 mcu 设置过来的参数、给硬核产生分频后的 clk、硬核采样完后读取结果并存储下来、如果设置多个通道采样则自动切换通道、如果设置 DMA 则采样完一个通道后发出 DMA 请求、如果设置连续采样则一轮一轮持续下去、为 MCU 准备数据。
硬核 ADC 是不管那么多的，只要满足硬核的两个条件（enable 和 clk），硬核就会用设置给它的通道号进行数据采样。那么，连续采样/多通道采样，中间怎么变动通道，怎么判断持续，这些便都是 cpld 里实现的了。

便于理解整个交互流程，可以设想一个场景：配置 2 个通道采样，并设置 DMA 读取，设置为不连续采样。其工作过程：

1. mcu 端设置采样通道； ----cpld 对应写寄存器
2. mcu 端配置 DMA；----cpld 对应写寄存器
3. mcu 端启动 ADC 采样（ADC_START）；----cpld 对应写寄存器
4. cpld 端根据配置的分频数产生 adc_clk 给到硬核，硬核开始采样；
5. 当硬核采样到第 4 个 adc_clk 时，会检测到接下来还有通道要采样，会切换通道号；
6. 第一个通道采样完后，硬核产生 adc_eoc 的信号；
7. cpld 根据这个信号，将数据读到 apb_db 中，并在下个 clk 里产生跟随的 apb_eoc 信号；
8. 数据到 apb_db 后，触发 DMA 信号，通知 DMA 来读取数据；
9. 硬核用第 5 步切换的通道号继续开始采样；
10. 当硬核采样到第 4 个 adc_clk 时，cpld 会检测到接下来没有通道要采集了，设置各标记；
11. 第二个通道采样完后，硬核产生 adc_eoc 的信号；
12. cpld 根据这个信号，将数据读到 apb_db 中，并在下个 clk 里产生跟随的 apb_eoc 信号；
13. 数据到 apb_db 后，触发 DMA 信号，通知 DMA 来读取数据；
14. 一轮已经全部采样完，adc_en 被设置为 disable，硬核停止工作；
15. 设置 stat_adc_eoc 为 1，并结束 cpld 工作。等待下次启动。

五、CPLD 端对 DAC 的处理

DAC 的处理和 ADC 相似，更简单些。

前边提到过，芯片里本身是包含了 ADC/DAC/CMP 硬核的。但 mcu 并没有和硬核直接相连，而是通过这里的 cpld 来连接的。
那么，硬核 ADC 部分的信号接口，从这里的调用看，就是 alta_dac 部分：

```
alta_dac dac_inst(
   .enb    (!ctrl_dac_en    ),
   .bufenb(!ctrl_dac_bufen),
   .din    (dac_din         ),
   .stop   (stop            ),
   .dout   (dac_dout        )
);
```

从 alta_sim.v 中看硬核 DAC 的接口定义：

```
module alta_dac (
  input enb, bufenb, stop,
  input [9:0] din,
  output dout
);
```

先看以上接口的几个信号的意义：（这部分可以从 Reference Manual 手册上获取）
enb：设置 DAC 是否使能 （0: enable, 1: disable）
bufenb：是否使能输出缓冲（运算放大器）（0: enable, 1: disable）
　　不管是否开启这项，都可以输出。但开启的话，会使输出阻抗很低。
　　这项是独立使用的，跟其他逻辑无关。
stop：是否开启 DAC 转换（0: disable，1: enable 即，关闭 adc）
din：10 位输出数据的值；
dout：（output）信号输出到的 IO。
使用上，din 数据准备好，不管输出缓冲是否开启，enb 和 stop 都拉低时，就进入 DAC 转换。DAC 转换完后，就会处于保持状态。如果有新的数据写进来，则改变完 din 后，输出电压自动变化。一直到关闭 DAC 转换为止（enb 拉高）。

代码角度看，DAC 比 ADC 简单上很多。
MCU 的 DAC 代码不再讲解，几个函数也都是调用寄存器的接口。
这里直接进入 CPLD 的讲解。

继续看 DAC module，看模块内如何处理 apb_xxx 信号的输入，以及将信号转换后操作上边的 alta_dac。

```
module apb_dac (
  input          stop,
  output         dma_req,
  input          dma_clr,
  input          apb_clock,
  input          apb_resetn,
  input          apb_psel,
  input          apb_penable,
  input          apb_pwrite,
  input   [11:0] apb_paddr,
  input   [31:0] apb_pwdata,
  output  [31:0] apb_prdata
);
```

这里是 DAC 模块的接口，跟 ADC 模块接口相同。

```
  parameter SCLK_BIT = 16;
```

常量 SCLK_BIT，用于记录分频寄存器的位数；

```
  parameter ADDR_CTRL = 'h00;
  parameter ADDR_DATA = 'h04;
```

对应 MCU 端的寄存器定义的偏移。

```
wire apb_data_phase = apb_psel && apb_penable;
```

apb_data_phase：当前 DAC 是否被选中并且使能（是否可正常操作该外设）。

当该信号为 1 时，该 DAC 外设可以处理 apb 过来的读写命令。

```
reg ctrl_dac_en, ctrl_dac_bufen, ctrl_dac_dmaen;
reg [SCLK_BIT-1:0] ctrl_sclk_div; // Sclk is divided by (ct
`define DAC_CTRL_REG {ctrl_sclk_div, 13'b0, ctrl_dac_dmaen,
```

记录 mcu 端设置过来的几个寄存器值；

注意，这里的 ctrl_dac_en 和 ctrl_dac_bufen 会连接到硬核 DAC 上。

```
reg [9:0] dac_din;
```

要转换的数据的寄存器，10 位宽度。

这个寄存器连接到硬核 DAC 的数据信号线上。

```
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    ctrl_dac_en    <= 'b0;
    ctrl_dac_bufen <= 'b0;
    ctrl_dac_dmaen <= 'b0;
    ctrl_sclk_div  <= 'b0;
  end else if (apb_data_phase && apb_pwrite && apb_paddr == ADDR_CTRL)
    ctrl_dac_en    <= apb_pwdata[0];
    ctrl_dac_bufen <= apb_pwdata[1];
    ctrl_dac_dmaen <= apb_pwdata[2];
    ctrl_sclk_div  <= apb_pwdata[31:16];
  end
end
```

这里就是 MCU 端设置（dac->CTRL）时，直接写进来值的地方。

这里的寄存器，记录 MCU 过来的信息；

```
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    dac_din <= 'b0;
  end else if (apb_data_phase && apb_pwrite && apb_paddr == ADDR_DATA)
    dac_din <= apb_pwdata[9:0];
  end
end
```

这里记录 MCU 端发送数据（dac->DATA）时的数据值。

```
wire [31:0] ctrl_read = apb_paddr == ADDR_CTRL ? `DAC_CTRL_REG : 0;
wire [31:0] data_read = apb_paddr == ADDR_DATA ? dac_din       : 0;
```

这里是要返回给 MCU 读取寄存器值的定义；

```
reg [31:0] prdata;
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    prdata <= 32'h0;
  end else if (apb_psel && !apb_penable && !apb_pwrite) begin
    prdata <= ctrl_read | data_read;
  end
end
assign apb_prdata = prdata;
```

当 MCU 读取 CTRL 或 DATA 时，就是这里处理并返回的。

这里利用了：一次读取只能有一个地址。所以，ctrl_read 和 data_read 只会有一个有值。

```
reg [SCLK_BIT-1:0] sclk_counter;
wire sclk_pulse = sclk_counter == ctrl_sclk_div;
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    sclk_counter <= 'b0;
  end else if (!ctrl_dac_dmaen || sclk_pulse) begin
    sclk_counter <= 'b0;
  end else begin
    sclk_counter <= sclk_counter + 1;
  end
end
```

sclk_counter 是分频计数。

dac_clk 是从 apb_clk 上分频出来的，分频数是 MCU 端设置下来的 ctrl_sclk_div。而这里的 sclk_counter 是在每个 apb_clk 到来时累加 1，加到 ctrl_sclk_div 后重置为 0.

sclk_pulse 信号，是在每次 sclk_counter 累加到 ctrl_sclk_div 时，产生的 1 个 apb_clk 时长的高电平。可以认为 sclk_pulse 是一次 DAC 数据转换完成的脉冲。

```
reg dma_reg;
always @ (posedge apb_clock or negedge apb_resetn) begin
  if (!apb_resetn) begin
    dma_reg <= 1'b0;
  end else if (!ctrl_dac_dmaen || dma_clr) begin
    dma_reg <= 1'b0;
  end else if (ctrl_dac_dmaen && sclk_pulse) begin
    dma_reg <= 1'b1;
  end
end
assign dma_req = dma_reg;
```

这里是对 DMA 的处理。

条件判断三，当 dma 使能，并且分频脉冲到来时，产生 DMA 信号去通知 DMA 搬运下一个数据。

到这里，DAC 的 cpld 代码也已经分析完了。

可以看到，DAC 部分的逻辑，都是直来直往的，没有太多的内部控制。

DAC 的分频，也是配合 DMA 的需要才设置的分频（一个分频时长，产生一次 dma 数据的搬运）。如果不用 DMA，是不需要设置分频的。

DAC 开启后，不会自动终止，需要 MCU 端来停止。

六、CPLD 端对 CMP 的处理

芯片硬核中包含 1 个 CMP，这个 CMP 又分为两路。两路相互独立，可以分开来用。

CPLD 中对 CMP 的处理，相比 DAC 更加简单。
CPLD 中只是对数据进行一个简单的传递：当 MCU 写参数时，cpld 把这些数据存储下，并传递到硬核 CMP；当 MCU 读设置的参数（和比较结果）时，cpld 返回比较结果。
仅此而已。
所以，cpld 代码本身没有什么，这里只需要理解硬核 CMP 的接口部分。
具体使用上，基本可以认为 MCU 的接口，就是对应硬核的接口了。

在 cpld 里调用硬核的地方：

```
alta_cmp cmp(
    .enb1   (!cmp_en1  ),
    .enb2   (!cmp_en2  ),
    .hyst1  (cmp_hyst1 ),
    .hyst2  (cmp_hyst2 ),
    .mode1  (cmp_mode1 ),
    .mode2  (cmp_mode2 ),
    .stop   (stop      ),
    .imsel1(cmp_imsel1),
    .imsel2(cmp_imsel2),
    .ipsel1(cmp_ipsel1),
    .ipsel2(cmp_ipsel2),
    .out1   (cmp_out1  ),
    .out2   (cmp_out2  )
);
```

从这里可以看到，硬核本身包含两组比较器：编号为 1 的一组和编号为 2 的一组。

而从接口定义看，如下：

```
module alta_cmp (
    input enb1, enb2, hyst1, hyst2, mode1, mode2, stop,
    input [2:0] imsel1, imsel2,
    input [1:0] ipsel1, ipsel2,
    output out1, out2
);
```

拆开成两组。
每组有：

    intput enb, hyst, mode,
    input[2:0] imsel,
    input[1:0] ipsel,
    output out,

各项意义：
enb：同 ADC/DAC 的 enb。是否开启 CMP 功能（0：enable；1：disable）
hyst：是否开启迟滞性（1：开启；0：不开启）
mode：设置速度（0：快速；1：慢速）

ipsel：选择哪个正相输入通道；（可选值：x_1 和 x_2）
imsel：选择哪个负相输入通道；（7 种选择：3 个 IO + 4 个参考电压）
out：输出比较值

使用时，设置正相输入和反相输入，然后 enable，就可以获取输出的比较值了。
目前的 MCU 端并没有设置 mode 和 hyst 的接口，如果需要，可以自己在相应位置增加值。
hyst 和 mode 在比较器中的作用，请自行百度。

如果需要增加比较器触发的中断，在目前的 cpld 处理上，增加 out 的检测 wire，当满足要求时，触发 mcu 端的中断即可。

cpld 代码部分太简单，不再分析。

七、底层逻辑的对应

由上边的描述，已经知道：AG32 中是包含了 3 个 ADC 硬核和 2 个 DAC 硬核的。
但从以上的代码中，却看不出来 cpld 的 3 路 ADC 是如何一一对应到 3 个 ADC 硬核的。
比如说，
如果只开一路 ADC1（关闭 ADC0 和 ADC2），怎么确保是从硬核 ADC1 来采集的？
如果只开一路 DAC1（关闭 DAC0），怎么确保输出的信号能到硬核 DAC1 的引脚？

这部分的对应关系，并不是在代码里指定的，而是通过 asf 文件配置出来的。
参考样例工程 example/analog/logic/路径下的.asf 文件：

```
set_instance_assignment -extension -name FIXED_COORD -to macro_inst|gen_per[0].gen_adc.adc_inst|adc_inst {22 7}
set_instance_assignment -extension -name FIXED_COORD -to macro_inst|gen_per[1].gen_adc.adc_inst|adc_inst {22 8}
set_instance_assignment -extension -name FIXED_COORD -to macro_inst|gen_per[2].gen_adc.adc_inst|adc_inst {22 9}
set_instance_assignment -extension -name FIXED_COORD -to macro_inst|gen_per[3].gen_dac.dac_inst|dac_inst {22 11}
set_instance_assignment -extension -name FIXED_COORD -to macro_inst|gen_per[4].gen_dac.dac_inst|dac_inst {22 12}

```

ADC 和 DAC 的逻辑和硬核对应关系，是在上述文件中指定的。
上述文件中，5 行，分别定义了 3 个 ADC 和 2 个 DAC 在硬核逻辑中的对应关系。

上述配置，对照 analog_ip.v 中的代码，就会发现 gen_per 对应到实例化的对象列表如下：

```
genvar i;
generate
  for (i = 0; i < PER_CNT; i = i + 1) begin : gen_per
    assign per_psel[i]    = apb_psel    & select[i];
    assign per_penable[i] = apb_penable & select[i];
    assign per_pwrite[i]  = apb_pwrite;
    assign per_paddr[i]   = apb_paddr[PER_BITS-1:0];
    assign per_pwdata[i]  = apb_pwdata;
    if (i < 3) begin : gen_adc
      apb_adc adc_inst(
        .stop        (stop         ),
        .dma_req     (dma_req[i]   ),
        .dma_clr     (dma_clr[i]   ),
        .apb_clock   (apb_clock    ),
        .apb_resetn  (resetn       ),
        .apb_psel    (per_psel[i]  ),
        .apb_penable (per_penable[i]),
        .apb_pwrite  (per_pwrite[i] ),
        .apb_paddr   (per_paddr[i]  ),
        .apb_pwdata  (per_pwdata[i] ),
        .apb_prdata  (per_prdata[i] )
      );
    end else if (i < 5) begin : gen_dac
```

3路ADC

2路DAC

这里要注意 asf 中每一行的最后一个参数 {22 xx}。
这个{22 xx}，其实是已经预设好的硬核 ID（或者叫“硬核位置/坐标”）。
比如：
{22 7}对应的是 ADC0 的 ID，在 cpld 最终编译后这个位置就是硬核 ADC0 的连接点；
{22 8}对应的是 ADC1 的 ID，在 cpld 最终编译后这个位置就是硬核 ADC1 的连接点；
… …
注：这些位置坐标在内部已经指定好（由 supra 编译时指定），不能更改。

另外，该.asf 内的配置，其实也是 AG32 的默认配置：
即：如果用户没有这样的配置，那么编译时默认 ADC 和 DAC 的配置就是这样的顺序。如果用户有自定义的配置，则编译时使用用户的配置。

知道了以上的对应关系，那么想改变顺序，配合 cpld 的代码来对应调整 asf 即可。

另外，3 路独立的 ADC，在各个通道上是连通的。
ADC 采集时，引脚只和通道相关。跟使用哪个 ADC 其实关系不大。不管 cpld 里边配置了几路 ADC，硬核的 3 路 ADC 都是连通并打开的。
也就是说，就算 mcu/cpld 里只打开了 ADC0，但 asf 里将它配置成 ADC 硬核 1，cpld 里仍然能读到正常的值（cpld 认为是从 adc0 读，但实际从 adc1 读）。
对比 DAC，DAC 则是一路 DAC 对应一个引脚，指定了 DACx 就是指定了引脚。在这里其实跟 ADC 没有类比性。