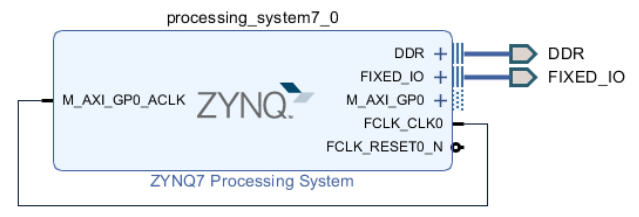
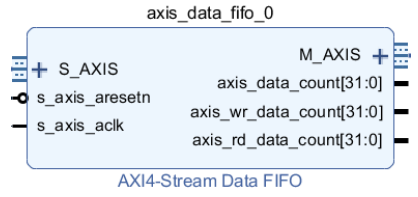
**FPGA BD工程搭建**

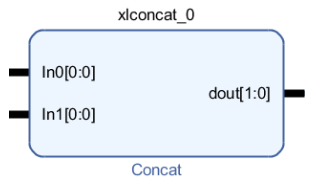
Step1:新建一个名为为Miz\_sys的工程。

Step2:创建一个BD文件，并命名为system，添加并且配置好ZYNQ IP。读者需要根据自己的硬件类型配置好输入时钟频率、内存型号、串口，连接时钟等。

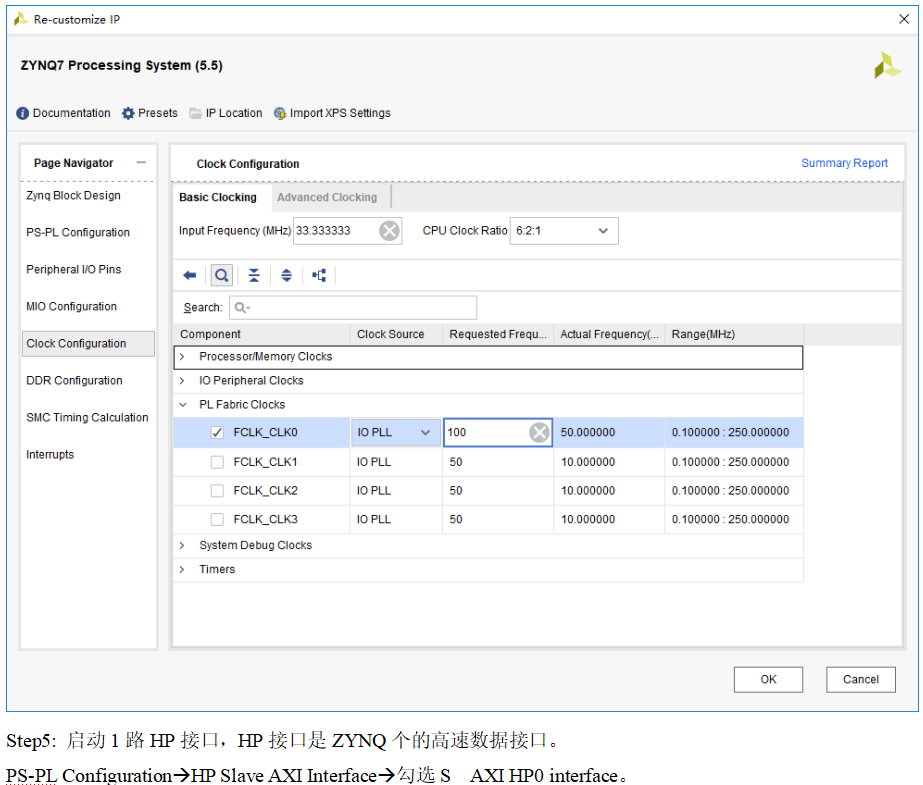


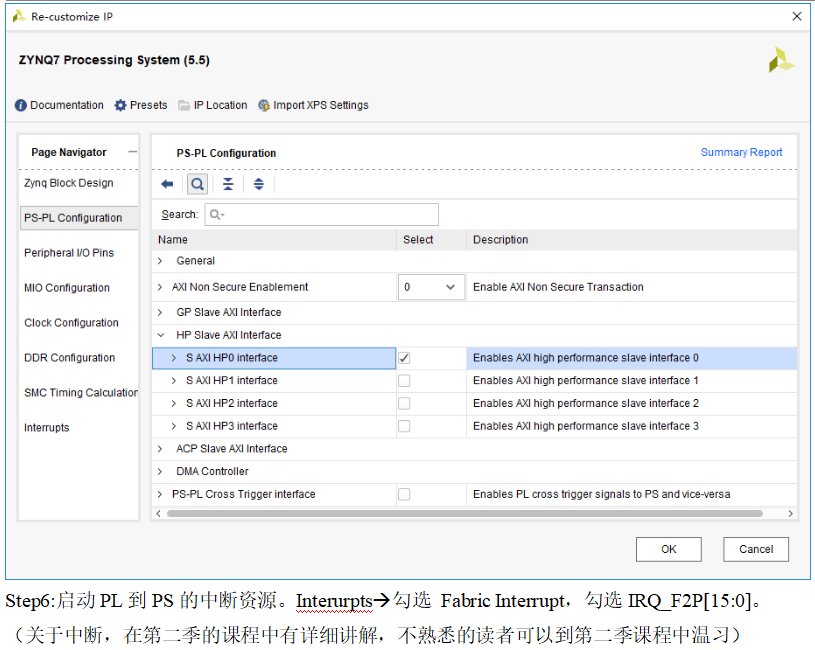


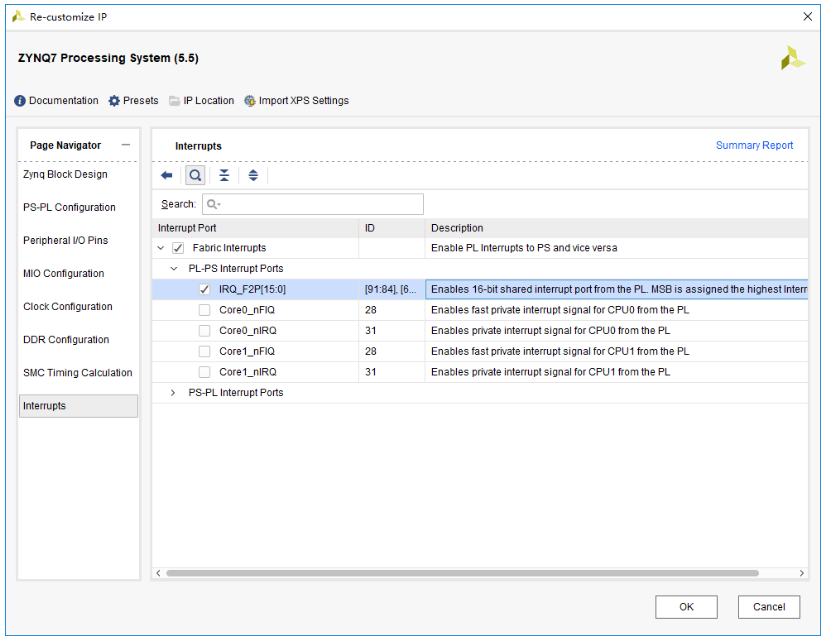




Step4：PL Fabric Clock:勾选FCLK\_CLK0，设置为100，即PS的PLL提供本系统的时钟100MHZ。





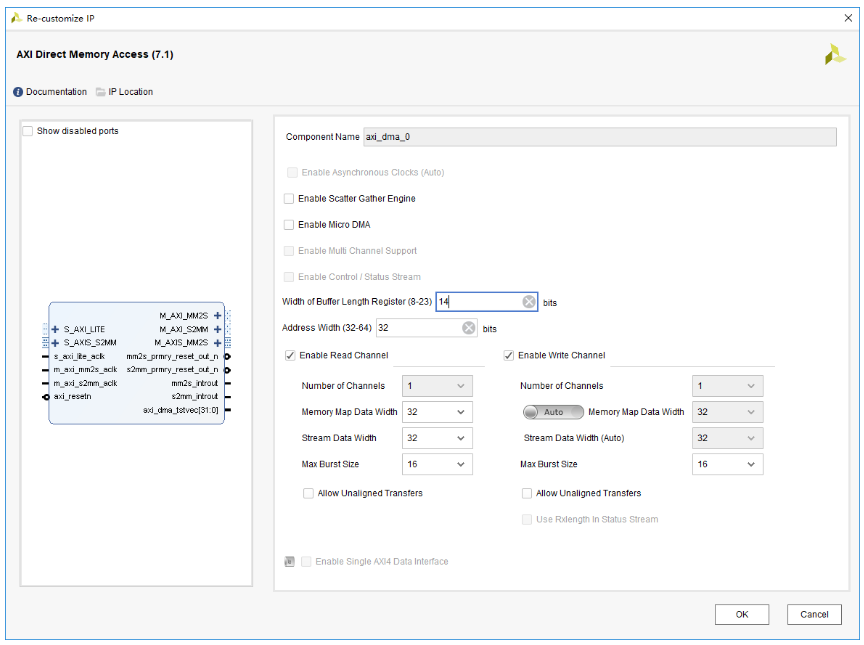


Step7:单击OK。

Step8:双击DMA  IP ，设置如下。

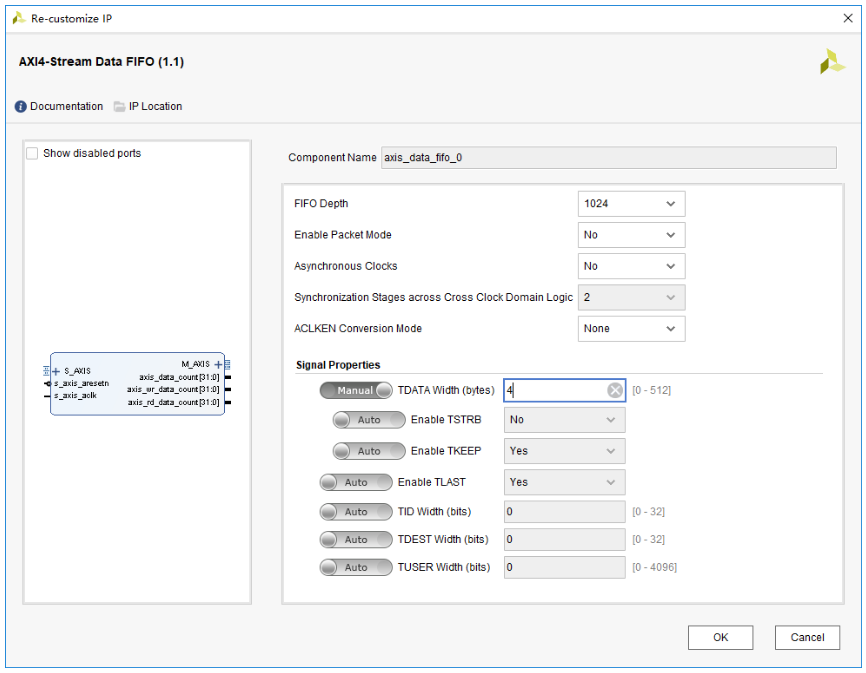
勾选读通道；勾选写通道；

设置Wideh of buffer length register ：14。（寄存器设置最大为23，即2的23次方8,388,607bytes，8M大小，这里设置14bit 就够用了，长度越大，需要的资源也就越多）



Step9:Data FIFO 设置

设置TDATA Width为4。



Step12:Concat IP设置

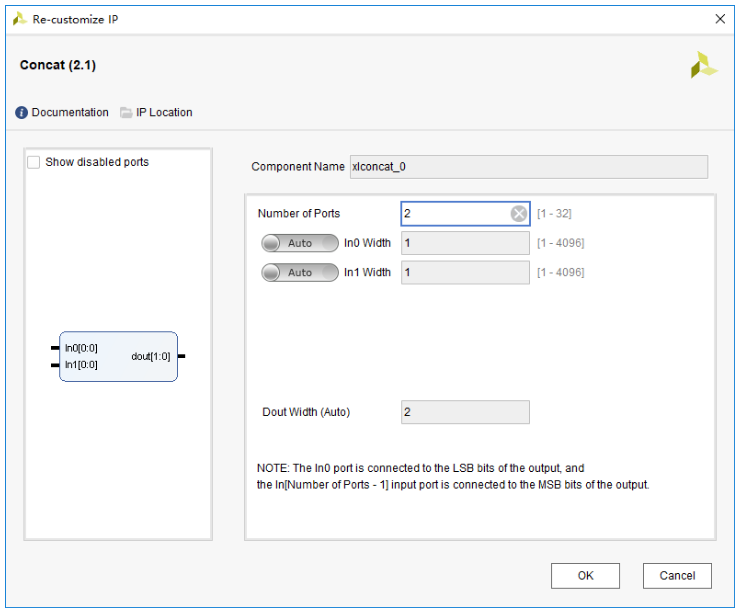
Concate IP实现了单个分散的信号，整合成总线信号。这里，将2个独立的中断信号，合并在一起连接到ZYNQ IP的中断信号接口上。

设置：Number of  Ports:2

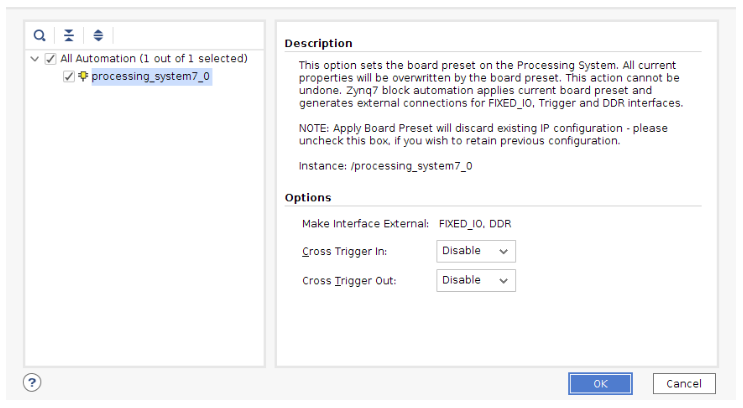
          In0 Width :1

          In1 Width :1

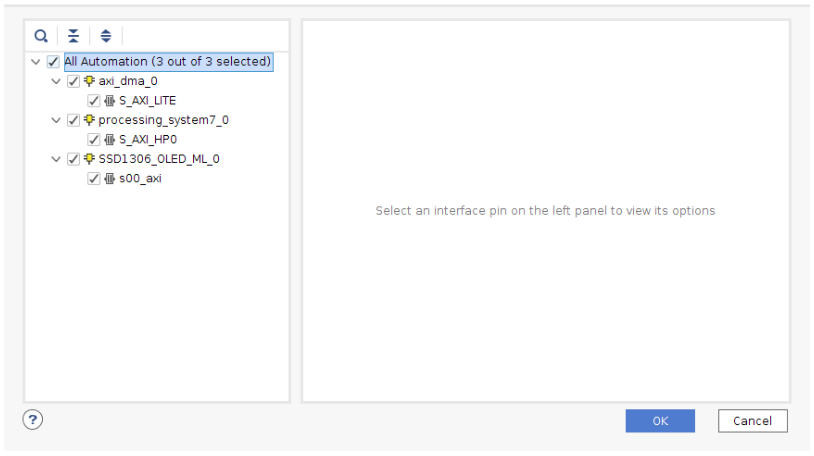
          Dout Width:2



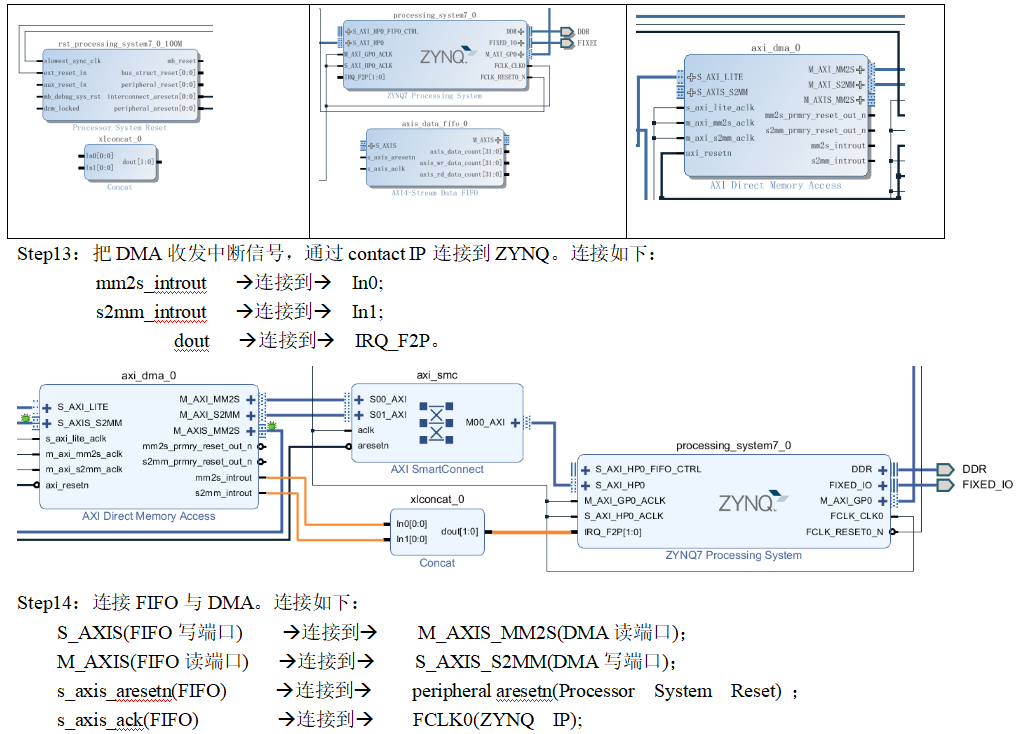
Step10:点击Run  Block  Automation ，自动配置ZYNQ IP 。



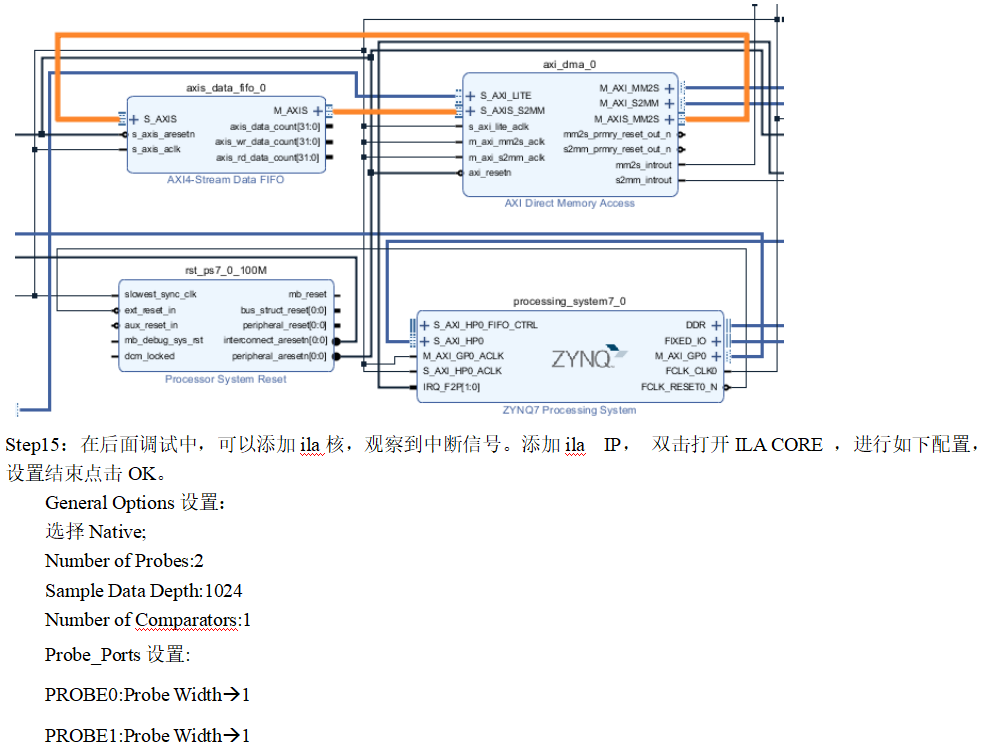
Step11：点击Run Connection Automation 自动连线。只要软件提示你需要自动连线，一般都需要进行自动连线，除非自己知道如何连线，有特殊需求。

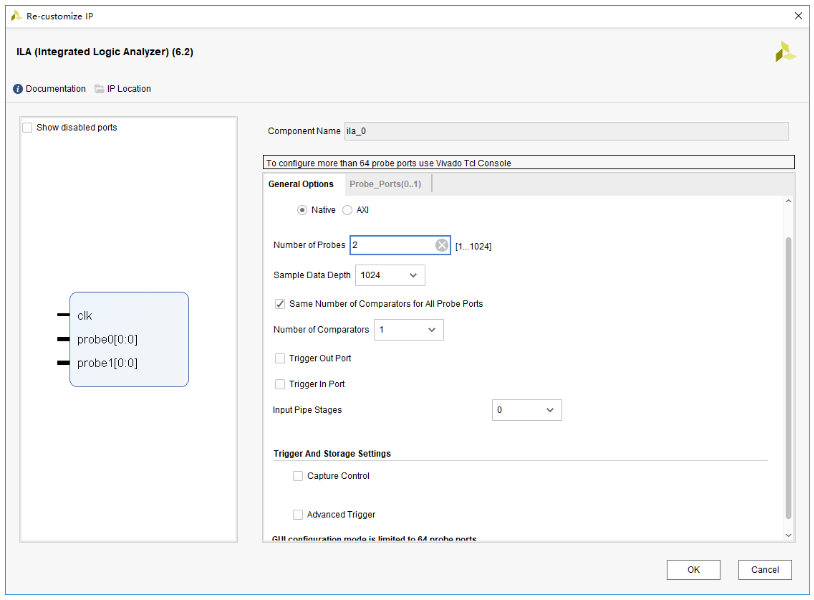


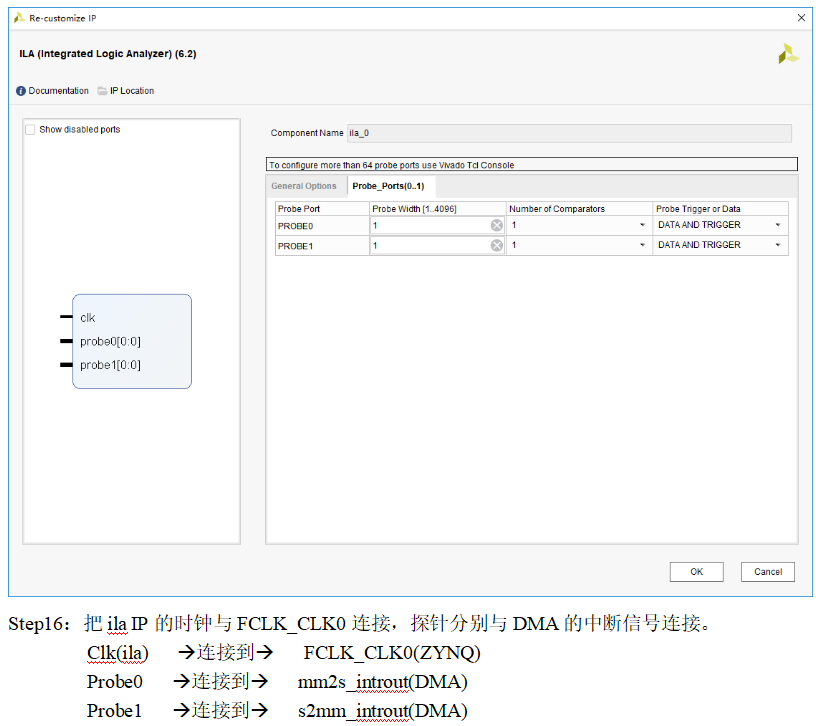
Step12：如果还有提示需要自动连线的继续让软件自动连线，直到出下如下。可以看到，还有未连线的模块。



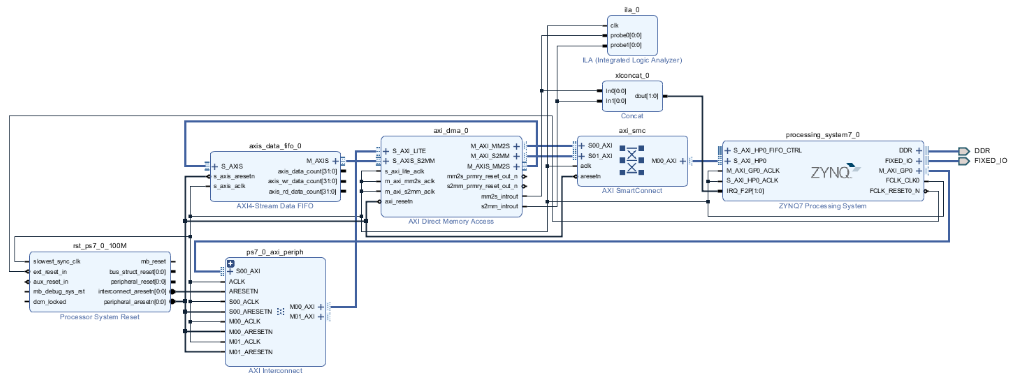
连接完成后如下图







**21.3 搭建好的FPGA BD工程**

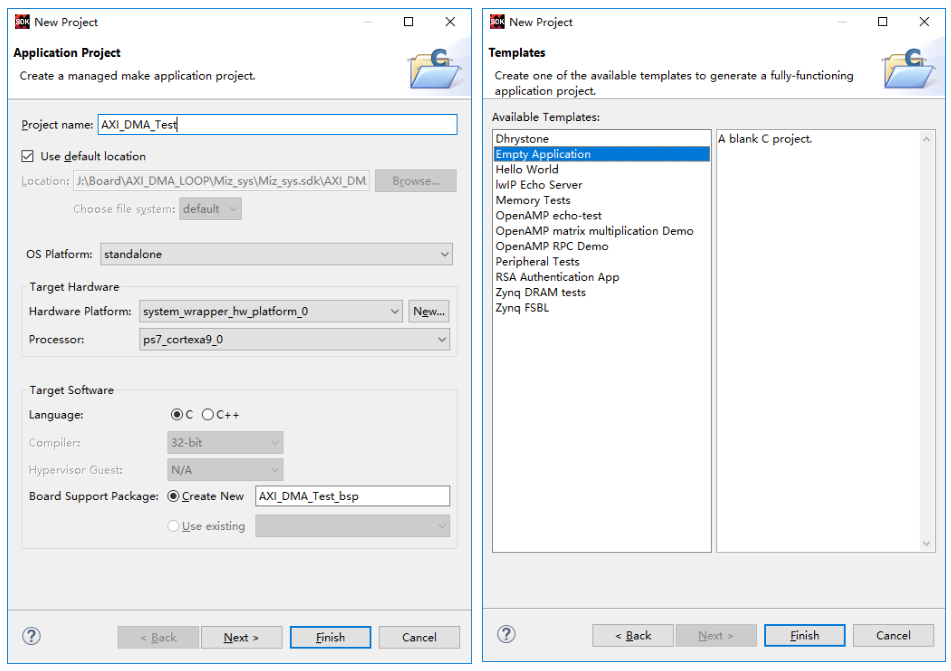


       至此，就完成了工程架构的搭建。后面的操作过程是Validate Design->Gerate Out products->Create wrappers-> Generate Bitstream ，产生完成后导出硬件，加载SDK。

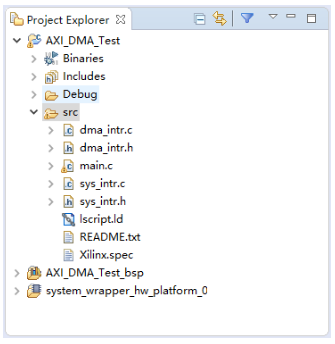
**21.4 PS部分软件分析**

**21.4.1新建SDK工程**

Step1:新建一个名为AXI\_DMA\_Test的空的软件工程



Step2:将提供例程中SDK工程的源文件复制，并粘贴到新建SDK工程，软件会自动编译。



**21.3.2 main.c源码的分析**

**函数分析1**

函数名：int init\_intr\_sys(void)；

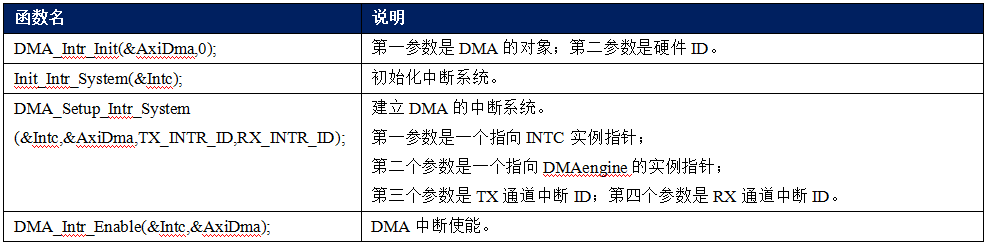
功能：对中断资源的初始化，使能中断资源。

说明：这个函数里面调用的函数是笔者封装好的初始化函数，使用起来比较方便。一般只要给出中断对象，中断号，就可以对中断进行初始化。

**init\_intr\_sys函数**

|  |
| --- |
| int init\_intr\_sys(void)  {  DMA\_Intr\_Init(&AxiDma,0);//initial interrupt system  Init\_Intr\_System(&Intc); // initial DMA interrupt system  Setup\_Intr\_Exception(&Intc);  DMA\_Setup\_Intr\_System(&Intc,&AxiDma,TX\_INTR\_ID,RX\_INTR\_ID);//setup dma interrpt system  DMA\_Intr\_Enable(&Intc,&AxiDma);  } |

**init\_intr\_sys()调用函数**



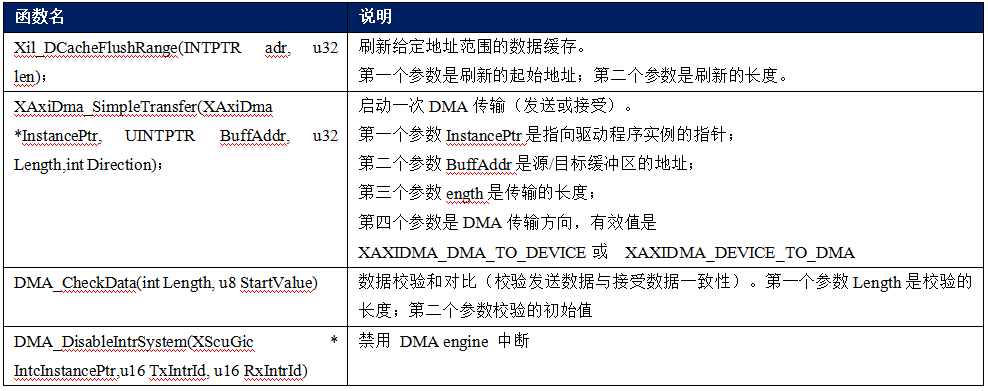
**函数分析2**

函数名称：int axi\_dma\_test()

功能：AXI DMA 传输测试

说明：为了发送的数据是已知是确定数据，先对TxBufferPtr 发送缓冲进行初始化，初始化后用Xil\_DCacheFlushRange 函数把数据全部刷到DDR中。

**axi\_dma\_test()调用函数**



**Main.c文件**

|  |
| --- |
| int axi\_dma\_test()  {  int Status;  TxDone = 0;  RxDone = 0;  Error = 0;    xil\_printf("\r\n----DMA Test----\r\n");  xil\_printf("PKT\_LEN=%d\r\n",MAX\_PKT\_LEN);    //while(1)  for(i = 0; i < Tries; i ++)  {  Value = TEST\_START\_VALUE + (i & 0xFF);  for(Index = 0; Index < MAX\_PKT\_LEN; Index ++) {  TxBufferPtr[Index] = Value;    Value = (Value + 1) & 0xFF;  }    /\* Flush the SrcBuffer before the DMA transfer, in case the Data Cache   \* is enabled   \*/  Xil\_DCacheFlushRange((u32)TxBufferPtr, MAX\_PKT\_LEN);    Status = XAxiDma\_SimpleTransfer(&AxiDma,(u32) RxBufferPtr,  MAX\_PKT\_LEN, XAXIDMA\_DEVICE\_TO\_DMA);    if (Status != XST\_SUCCESS) {  return XST\_FAILURE;  }    Status = XAxiDma\_SimpleTransfer(&AxiDma,(u32) TxBufferPtr,  MAX\_PKT\_LEN, XAXIDMA\_DMA\_TO\_DEVICE);    if (Status != XST\_SUCCESS) {  return XST\_FAILURE;  }    /\*   \* Wait TX done and RX done   \*/  while (!TxDone || !RxDone) {  /\* NOP \*/  }    success++;  TxDone = 0;  RxDone = 0;    if (Error) {  xil\_printf("Failed test transmit%s done, "  "receive%s done\r\n", TxDone? "":" not",  RxDone? "":" not");  goto Done;  }  /\*   \* Test finished, check data   \*/  Status = DMA\_CheckData(MAX\_PKT\_LEN, (TEST\_START\_VALUE + (i & 0xFF)));  if (Status != XST\_SUCCESS) {  xil\_printf("Data check failed\r\n");  goto Done;  }    }  xil\_printf("AXI DMA interrupt example test passed\r\n");  xil\_printf("success=%d\r\n",success);  /\* Disable TX and RX Ring interrupts and return success \*/  DMA\_DisableIntrSystem(&Intc, TX\_INTR\_ID, RX\_INTR\_ID);  Done:  xil\_printf("--- Exiting Test --- \r\n");  return XST\_SUCCESS;    }    int init\_intr\_sys(void)  {  DMA\_Intr\_Init(&AxiDma,0);//initial interrupt system  Init\_Intr\_System(&Intc); // initial DMA interrupt system  Setup\_Intr\_Exception(&Intc);  DMA\_Setup\_Intr\_System(&Intc,&AxiDma,TX\_INTR\_ID,RX\_INTR\_ID);//setup dma interrpt system  DMA\_Intr\_Enable(&Intc,&AxiDma);  }    int main(void)  {    init\_intr\_sys();  axi\_dma\_test();    } |

**21.3.3 dma\_intr.c 源码分析**

      XAxiDma \*AxiDmaInst = (XAxiDma \*)Callback;这句代码是为了获取当前中断的对象。void \*Callback是一个无符号的指针，传递进来的阐述可以强制转换成其他任何的对象，这里就是强制转换成 XAxiDma 对象了。

      IrqStatus =XAxiDma\_IntrGetIrq(AxiDmaInst, XAXIDMA\_DMA\_TO\_DEVICE)这个函数获取当前中断号。

      XAxiDma\_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA\_DMA\_TO\_DEVICE);这个函数是响应当前中断，通知CPU 当前中断已经被接收，并且清除中断标志位。如果中断全部正确，TxDone将被置为1表示发送中断完成。如果有错误，则复位DMA，并且设置超时参数

**DMA\_TxIntrHandler函数**

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*  \*  \* This is the DMA TX Interrupt handler function.  \*  \* It gets the interrupt status from the hardware, acknowledges it, and if any  \* error happens, it resets the hardware. Otherwise, if a completion interrupt  \* is present, then sets the TxDone.flag  \*  \* @param Callback is a pointer to TX channel of the DMA engine.  \*  \* [@return](https://my.oschina.net/u/556800) None.  \*  \* [@note](https://my.oschina.net/u/1188662) None.  \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  static void DMA\_TxIntrHandler(void \*Callback)  {    u32 IrqStatus;  int TimeOut;  XAxiDma \*AxiDmaInst = (XAxiDma \*)Callback;    /\* Read pending interrupts \*/  IrqStatus = XAxiDma\_IntrGetIrq(AxiDmaInst, XAXIDMA\_DMA\_TO\_DEVICE);    /\* Acknowledge pending interrupts \*/      XAxiDma\_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA\_DMA\_TO\_DEVICE);    /\*   \* 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 should never fail for transmit channel   \*/  XAxiDma\_Reset(AxiDmaInst);    TimeOut = RESET\_TIMEOUT\_COUNTER;    while (TimeOut) {  if (XAxiDma\_ResetIsDone(AxiDmaInst)) {  break;  }    TimeOut -= 1;  }    return;  }    /\*   \* If Completion interrupt is asserted, then set the TxDone flag   \*/  if ((IrqStatus & XAXIDMA\_IRQ\_IOC\_MASK)) {    TxDone = 1;  }  } |

接收中断函数的原理和发送一样

      XAxiDma \*AxiDmaInst = (XAxiDma \*)Callback;这句代码是为了获取当前中断的对象。void \*Callback是一个无符号的指针，传递进来的阐述可以强制转换成其他任何的对象，这里就是强制转换成 XAxiDma 对象了。

     IrqStatus = XAxiDma\_IntrGetIrq(AxiDmaInst, XAXIDMA\_DEVICE\_TO\_DMA);这个函数是获取当前中断号。

     XAxiDma\_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA\_DEVICE\_TO\_DMA);这个函数是响应当前中断，通知CPU 当前中断已经被接收，并且清除中断标志位。

     如果中断全部正确，RxDone将被置为1表示接收中断完成。如果有错误，则复位DMA，并且设置超时参数

**DMA\_RxIntrHandler函数**

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*  \*  \* This is the DMA RX interrupt handler function  \*  \* It gets the interrupt status from the hardware, acknowledges it, and if any  \* error happens, it resets the hardware. Otherwise, if a completion interrupt  \* is present, then it sets the RxDone flag.  \*  \* @param Callback is a pointer to RX channel of the DMA engine.  \*  \* [@return](https://my.oschina.net/u/556800) None.  \*  \* [@note](https://my.oschina.net/u/1188662) None.  \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  static void DMA\_RxIntrHandler(void \*Callback)  {  u32 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)) {    RxDone = 1;  }  } |

**DMA\_CheckData函数**

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*  \*  \* This function checks data buffer after the DMA transfer is finished.  \*  \* We use the static tx/rx buffers.  \*  \* @param Length is the length to check  \* @param StartValue is the starting value of the first byte  \*  \* [@return](https://my.oschina.net/u/556800)  \* - XST\_SUCCESS if validation is successful  \* - XST\_FAILURE if validation is failure.  \*  \* @note None.  \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   int DMA\_CheckData(int Length, u8 StartValue)  {  u8 \*RxPacket;  int Index = 0;  u8 Value;    RxPacket = (u8 \*) RX\_BUFFER\_BASE;  Value = StartValue;    /\* Invalidate the DestBuffer before receiving the data, in case the   \* Data Cache is enabled   \*/  #ifndef \_\_aarch64\_\_  Xil\_DCacheInvalidateRange((u32)RxPacket, Length);  #endif    for(Index = 0; Index < Length; Index++) {  if (RxPacket[Index] != Value) {  xil\_printf("Data error %d: %x/%x\r\n",      Index, RxPacket[Index], Value);    return XST\_FAILURE;  }  Value = (Value + 1) & 0xFF;  }    return XST\_SUCCESS;  } |

**21.3.4 dam\_intr.h 文件分析**

一般把DMA相关变量、常量、函数的声明或者定义放到头文件中，dam\_intr.h比较关键的参数如下：

**dam\_intr.h中变量**

|  |  |
| --- | --- |
| **名称** | **说明** |
| TX\_BUFFER\_BASE | DMA发送缓存的基地址 |
| RX\_BUFFER\_BASE | DMA接收缓存的基地址 |
| MAX\_PKT\_LEN | 表示每一包数据传输的长度 |
| NUMBER\_OF\_TRANSFERS | 用在连续测试的时候的测试次数 |
| TEST\_START\_VALUE | 用于测试的起始参数 |

**dam\_intr.h中函数**

|  |  |
| --- | --- |
| **名称** | **说明** |
| DMA\_CheckData | 对数据进行对比 |
| DMA\_Setup\_Intr\_System | DMA 中断注册 |
| DMA\_Intr\_Enable | DMA中断使能 |
| DMA\_Intr\_Init | DMA中断初始化 |

**dam\_intr.h**

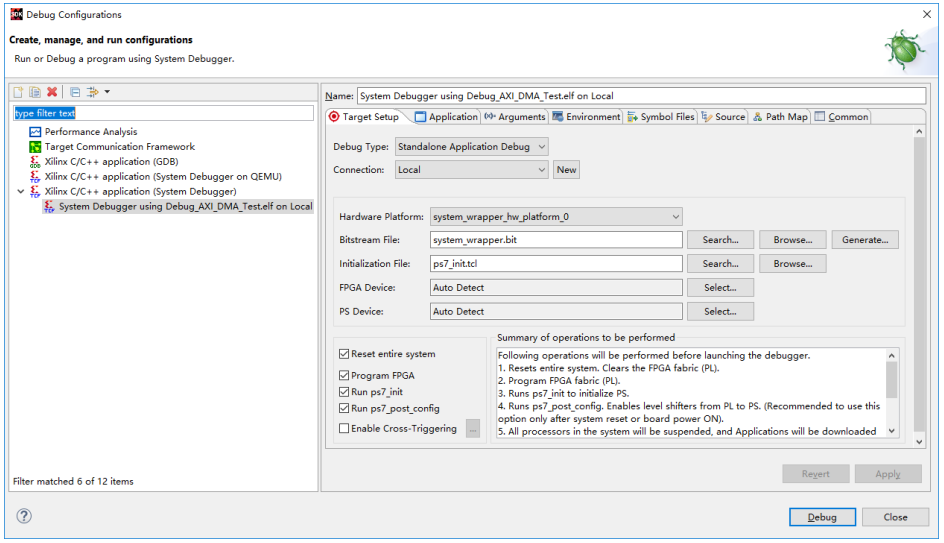
|  |
| --- |
| #ifndef DMA\_INTR\_H  #define DMA\_INTR\_H  #include "xaxidma.h"  #include "xparameters.h"  #include "xil\_exception.h"  #include "xdebug.h"  #include "xscugic.h"    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constant Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*   \* Device hardware build related constants.   \*/  #define DMA\_DEV\_ID XPAR\_AXIDMA\_0\_DEVICE\_ID    #define MEM\_BASE\_ADDR 0x01000000      #define RX\_INTR\_ID XPAR\_FABRIC\_AXI\_DMA\_0\_S2MM\_INTROUT\_INTR  #define TX\_INTR\_ID XPAR\_FABRIC\_AXI\_DMA\_0\_MM2S\_INTROUT\_INTR      #define TX\_BUFFER\_BASE (MEM\_BASE\_ADDR + 0x00100000)  #define RX\_BUFFER\_BASE (MEM\_BASE\_ADDR + 0x00300000)  #define RX\_BUFFER\_HIGH (MEM\_BASE\_ADDR + 0x004FFFFF)      /\* Timeout loop counter for reset   \*/  #define RESET\_TIMEOUT\_COUNTER 10000  /\* test start value   \*/  #define TEST\_START\_VALUE 0xC  /\*   \* Buffer and Buffer Descriptor related constant definition   \*/  #define MAX\_PKT\_LEN 256//4MB  /\*   \* transfer times   \*/  #define NUMBER\_OF\_TRANSFERS 100000    extern volatile int TxDone;  extern volatile int RxDone;  extern volatile int Error;    int  DMA\_CheckData(int Length, u8 StartValue);  int  DMA\_Setup\_Intr\_System(XScuGic \* IntcInstancePtr,XAxiDma \* AxiDmaPtr, u16 TxIntrId, u16 RxIntrId);  int  DMA\_Intr\_Enable(XScuGic \* IntcInstancePtr,XAxiDma \*DMAPtr);  int  DMA\_Intr\_Init(XAxiDma \*DMAPtr,u32 DeviceId);  #endif |

**21.4验证测试**

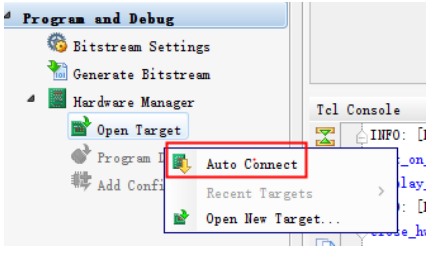
测试是使用软件： VIVADIO 和 SDK

测试步骤如下：

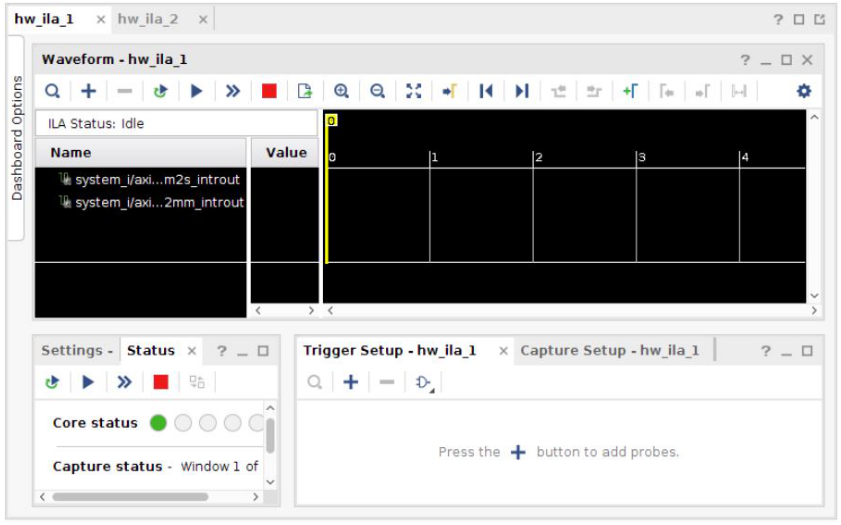
Step1:启动SDK



Step2:在VIVADO工程中点击Open Target 然后点击Auto Connect

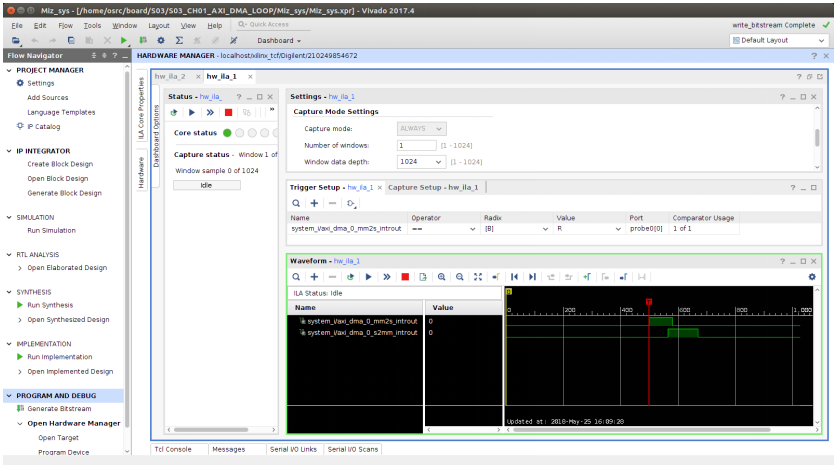


Step3:连接成功后入下图



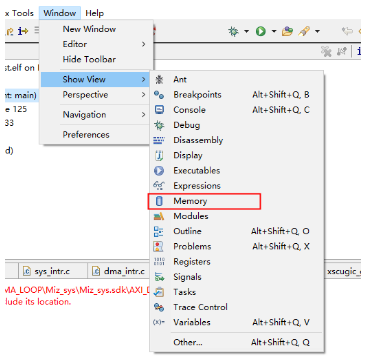


当中断触发的时，VIVAIDO中Hardware Manager出现捕捉波形，如下图所示

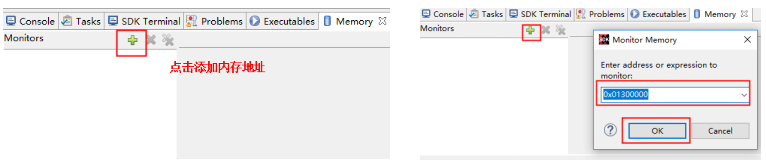


Step6:观察数据

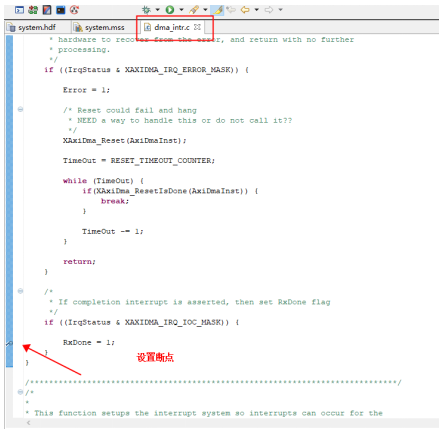
打开Memory：Window->Show View->Memory

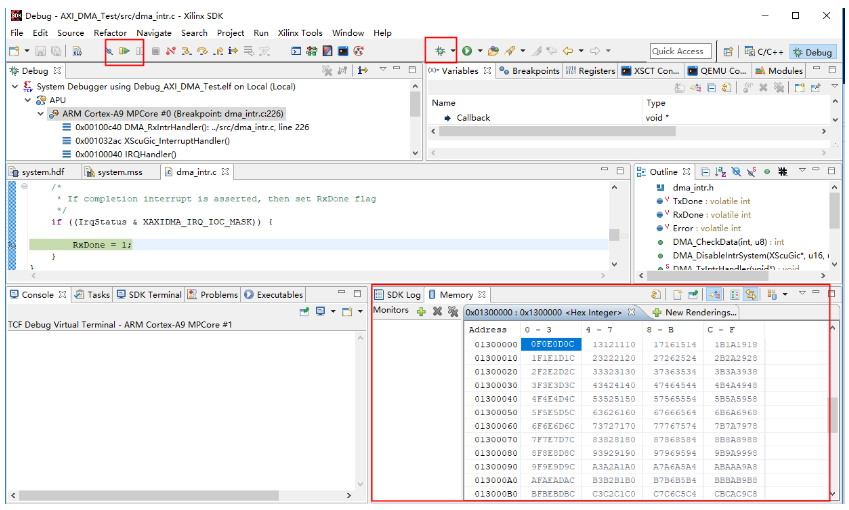


点击添加接收内存部分地址用于观察内存中的数据 地址为 0x01300000

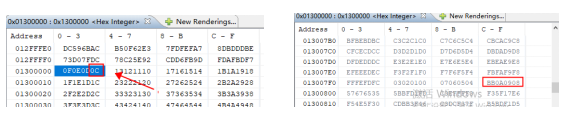


为了观察一次收发数据：设置断点，重新让收发程序跑一次。





  收发一次，从内存中读取的数据如图：



  可以看到第一个数据是0X0C ，后面是依次加1，一次接收的数据量是2047。发送数据也是OX0C后面依次加1,发送量是2047。接收数据一致，测试结束。