

## 第三篇

### 实践篇

本篇将介绍智能嵌入式系统软硬件以及通信设计方法。掌握基于 Vivado HLS 的硬件 IP 软核生成方法，掌握基于 C++或 Python 的系统软件设计方法，掌握软硬件间批量传输通信和非批量传输通信方式设计方法。本篇将以基于卷积神经网络的交通标志识别系统为例，介绍在异构系统 PYNQ 平台上进行软硬件实现方法，提高交通标志牌识别速度。

通过本篇学习，掌握使用本教材内容建立智能嵌入式系统的解决过程。

本篇共有两章：

第 9 章 系统设计

第 10 章 系统集成

## 第9章 系统设计

如刀如磋，如琢如磨 《诗经·卫风·淇澳》

系统设计是从系统的软件、硬件以及软硬件间通信三个方面进行设计。系统软件设计是使用软件设计语言实现系统软件模块，系统硬件设计是使用硬件设计语言实现系统的硬件模块，而软硬件间通信设计使用符合通信协议的编程语言实现软硬件间的通信模块。

本章介绍使用 Vivado 工具平台进行硬件设计—生成 IP 核，使用编程语言 C++或 Python 进行软件设计，使用 C 语言进行微处理系统 PS 和处理逻辑系统 PL 之间 AXI4 协议的设计。

### 第9.1节 硬件 IP 核设计

硬件 IP 核(Intellectual Property core)<sup>[46]</sup>是知识产权核或知识产权模块的意思，在 EDA 技术开发中具有十分重要的地位。美国著名的 Dataquest 咨询公司将半导体产业的 IP 定义为“用于 ASIC 或 FPGA 中预先设计好的电路功能模块”。

#### 9.1.1 硬件 IP 核形式

硬件 IP 核有三种不同的存在形式：硬件语言形式、网表形式、版图形式。分别对应常说的三类 IP 内核：软核、固核和硬核。这种分类主要依据产品交付的方式，而这三种 IP 内核实现方法也各具特色。

软核是用 Verilog 等硬件描述语言描述的功能块，并不涉及用什么具体电路元件实现这些功能。软核是以源代码的形式提供，这样实际的 RTL 对用户是不可见的，但布局和布线灵活。尽管软核可以采用加密方法，但其知识产权保护问题仍受到巨大挑战。不过大多数应用于 FPGA 的 IP 内核均为软核，软核有助于用户调节参数，增强可复用性。

硬核提供设计阶段最终阶段产品，以版图形式提供给用户。硬核会以经过完全的布局布线网表形式提供，这种硬核既具有可预见性，同时还可以针对特定工艺或购买商进行功耗和尺寸上的优化。由于硬核无须提供寄存器转移级(RTL)文件，因而更易于实现 IP 保护。


固核则是完成了综合的功能块。它有较大的设计深度，以网表文件的形式提交客户使用。如果客户与固 IP 使用同一个 IC 生产线的单元库，IP 应用的成功率

会高得多。在这些加密的软核中，如果对内核进行了参数化，那么用户就可通过头文件或图形用户接口方便地对参数进行操作。对于那些对时序要求严格的内核(如 PCI-Peripheral Component Interconnect-接口内核)，可预布线特定信号或分配特定的布线资源，以满足时序要求。这些内核归为固核。

### 9.1.2 硬件 IP 软核生成

硬件 IP 软核是使用硬件描述语言编写的功能块代码。在第 3 章仿真方法中学过硬件描述语言 Verilog 编写功能模块代码。在第 4 章性能属性获取中学过了 Vivado 工具，这里介绍使用 Vivado 工具平台生成硬件 IP 核。

使用 Vivado HLS 工具平台生硬件 IP 软核，首先是使用 C/C++语言写成代码，然后改成符合 Vivada HLS 标准代码，这样就可以在 Vivado HLS 工具平台上运行并生成硬件 IP 核。以第 4 章例 4.3 欧几里得整数最大公约算法为例，介绍具体步骤：

Solution > Export RTL，或者点击工具栏快捷按钮 ，打开 Export RTL 对话框，会出现 IP Catalog 对话框，按照对话框选 Verilog，单击 OK 按钮。

命令行会打印提示整个 IP 封装过程，生成 IP 的过程结束会产生如下信息：

```
Implementation tool: Xilinx Vivado v.2018.2
Project:          gcd
Solution:         solution1
Device target:    xc7z020clg484-1
Report date:      Sat Jun 06 17:30:55 +0800 2020

#=== Post-Synthesis Resource usage ===
SLICE:           0
LUT:             39
FF:             96
DSP:             0
BRAM:           0
SRL:            1
#=== Final timing ===
CP required:     10.000
CP achieved post-synthesis: 3.442
Timing met
INFO: [Common 17-206] Exiting Vivado at Sat Jun 6 17:30:55 2020...

Finished export RTL.
```

IP封装完成后，gcd文件夹下会出现impl文件夹，该文件夹下包含ip、verilog、

vdhl 三个子文件夹。在 IP 文件夹中有 `xilinx_com_hls_gcd_1_0.zip` 文件，复制就可以得到 IP 核文件。在 IP 文件夹的子文件夹 `hdl` 里，有两个文件 `gcd.v` 和 `gcd_urem_8ns_8ns_bkb.v`。这两个文件都是 Verilog 文件，他们是硬件 IP 软核，同时也是生成图形化 IP 核的源文件，这两个文件分别是本书的附录一和附录二。

### 9.1.3 硬件 IP 软核图形化

使用 Vivado 工具（不是 Vivado HLS 工具）生成图形化硬件 IP 核。在 Vivado 平台下 **Create New Project**，建立 Project name 和 Location(如：C:/Vivado-files)，添加 Type 和 Sources (添加 IP 核的路径 `Directories-xilinx_com_hls_gcd_1_0`)，选定硬件编号如 `xc7z020clg484-1`，就完成了 Project 的建立。进入到 Project Manager 管理界面，在 **Settings>IP** 下选择 **Repository**，添加 IP Repository -- `xilinx_com_hls_gcd_1_0`，再打开 IP Catalog 下的 **User Repository>Vivado HLS IP** 就可以看到 **Gcd**，双击 **Gcd** 就可以生成 GCD 的图形 IP 核。具体操作工程可从 Vivado 工具说明书中获得。图 9-1 是例 4.3 欧几里得整数最大公约算法图形化硬件 IP 软核。从这个图可以看到，有 4 个输入端口：`clk`，`rst`，`m` 和 `n`，其中 `m` 和 `n` 是 8 位表示的输入数据，有 1 个输出端口 `return`，是 8 位表示的 `m` 与 `n` 的最大公约数。



图 9-1 例 4.3 欧几里得整数最大公约算法图形化硬件 IP 软核

## 第 9.2 节 软件设计

软件在微处理器上运行。在含有处理系统 PS 和处理逻辑 PL 的异构系统平台中，软件作用主要分为两个方面：一是软件要实现的系统功能，二是协调 PS 与 PL 之间的协同工作。因此在软件设计时，这两个方面都要考虑。

### 9.2.1 软件任务实现

软件设计需选取合适的语言对软件任务进行实现。一般会选择 C/C++ 实现软件的编程。而有的异构平台会带有像 Python 的处理语言，也可以选用平台提供的编程语言。

在第 4.1 节介绍了欧几里得最大公约数算法的 C/C++ 实现代码，这里使用 Python 语言编程实现欧几里得最大公约数算法。

```
# 定义一个函数
def gcd(x, y):
    """该函数返回两个数的最大公约数"""
    # 获取最小值
    if x > y:
        smaller = y
    else:
        smaller = x
    for i in range(1, smaller + 1):
        if ((x % i == 0) and (y % i == 0)):
            gcd = i
    return gcd
# 用户输入两个数字
num1 = int(input("输入第一个数字: "))
num2 = int(input("输入第二个数字: "))
print(num1, "和", num2, "的最大公约数为", gcd(num1, num2))
```

欧几里得整数最大公约算法 Python 代码

稍微复杂例子介绍卷积神经网络计算模型，是第 10 章交通标志识别系统 TSR(Traffic Sign Recognition system)的基础[40]。

卷积神经网络(Convolutional Neural Networks, CNN)的雏形早在 1980 年就被提出[47],在 1998 年 [LéCUN](#) 等人使用 CNN 识别手写数字后得到迅速发展,现在已被广泛应用于图像识别领域[48]。有关 CNN 的介绍也可参考文献[49]。一个卷积神经网络通常包括输入层(Input Layer)、卷积层(Convolutional Layer)、池化层(Pooling Layer)、全连接层(Fully Connected Layer)和输出层(Output Layer)。其中输入层和输出层分别是网络结构的第一层和最后一层,用于数据的输入和输出;卷积层是 CNN 的关键部分,使用卷积核对数据进行局部感知;池

化层可以将数据降维，减少参数个数；全连接层一般位于 CNN 的尾部，与传统神经网络中的连接方式相同，是将前一层所有神经元都与本层神经元相连。

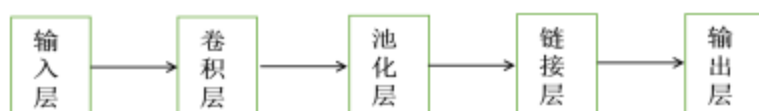


图 9-1 卷积神经网络结构图

文献[40]使用 C++语言对卷积神经网络的卷积层、池化层以及全链接层进行实现，这里摘录部分代码，全部代码可以从教材网站上下载。其中 BNN 是二值神经网络(Binary Neural Network)，对 CNN 的二值化。

### 卷积层:

```

@file convlayer.h
*
* Library of templated HLS functions for BNN deployment.
* This file lists a set of convenience funtions used to implement
* convolutional layers
*
*
*****
*****/

template<
// convolution parameters
    unsigned int ConvKernelDim, // e.g 3 for a 3x3 conv kernel (assumed
square)
    unsigned int IFMChannels, // number of input feature maps
    unsigned int IFMDim, // width of input feature map (assumed square)
    unsigned int OFMChannels, // number of output feature maps
    unsigned int OFMDim, // IFMDim-ConvKernelDim+1 or less

    // matrix-vector unit parameters
    unsigned int SIMDWidth, // number of SIMD lanes
    unsigned int PEGCount, // number of PEs
    unsigned int PopCountWidth, // number of bits for popcount
    unsigned int WMemCount, // entries in each PEs weight memory
    unsigned int TMemCount // entries in each PEs threshold memory
>
void StreamingConvLayer_Batch(stream<ap_uint<IFMChannels>> & in,
    stream<ap_uint<OFMChannels>> & out,
  
```

```

    const ap_uint<SIMDWidth> weightMem[PECount][WMemCount],
    const ap_uint<PopCountWidth> thresMem[PECount][TMemCount],
    const unsigned int numReps)
{
    // compute weight matrix dimension from conv params
    const unsigned int MatrixW = ConvKernelDim * ConvKernelDim * IFMChannels;
    const unsigned int MatrixH = OFMChannels;

#pragma HLS INLINE
    stream<ap_uint<IFMChannels>> >
convInp("StreamingConvLayer_Batch.convInp");
    WidthAdjustedOutputStream <PECount, OFMChannels, OFMDim * OFMDim *
OFMChannels / PECount> mvOut (out, numReps);
    StreamingConvolutionInputGenerator_Batch<ConvKernelDim, IFMChannels,
IFMDim, OFMDim, 1>(in, convInp, numReps);
    WidthAdjustedInputStream <IFMChannels, SIMDWidth, OFMDim * OFMDim *
ConvKernelDim * ConvKernelDim> mvIn (convInp, numReps);
    StreamingMatrixVector_Batch<SIMDWidth, PECount, PopCountWidth,
MatrixW, MatrixH, WMemCount, TMemCount>(mvIn, mvOut, weightMem, thresMem,
numReps * OFMDim * OFMDim);
}

template<
// convolution parameters
    unsigned int ConvKernelDim, // e.g 3 for a 3x3 conv kernel (assumed
square)
    unsigned int IFMChannels,    // number of input feature maps
    unsigned int IFMDim,        // width of input feature map (assumed
square)
    unsigned int OFMChannels,    // number of output feature maps
    unsigned int OFMDim,        // IFMDim-ConvKernelDim+1 or less

    // matrix-vector unit parameters
    unsigned int InpWidth,       // size of the fixed point input
    unsigned int InpIntWidth,   // number of integer bits for the fixed
point input
    unsigned int SIMDWidth,     // number of SIMD lanes
    unsigned int PECount,       // number of PEs
    unsigned int AccWidth,      // number of bits for accumulation
    unsigned int AccIntWidth,   // number of integer bits for
accumulation
    unsigned int WMemCount,     // entries in each PEs weight memory
    unsigned int TMemCount     // entries in each PEs threshold memory
>

```

```

void StreamingFxdConvLayer_Batch(stream<ap_uint<IFMChannels * InpWidth> >
& in,
    stream<ap_uint<OFMChannels> > & out,
    const ap_uint<SIMDWidth> weightMem[PECount][WMemCount],
    const ap_fixed<AccWidth, AccIntWidth>
thresMem[PECount][TMemCount],
    const unsigned int numReps)
{
    // compute weight matrix dimension from conv params
    const unsigned int MatrixW = ConvKernelDim * ConvKernelDim * IFMChannels;
    const unsigned int MatrixH = OFMChannels;
#pragma HLS INLINE
    stream<ap_uint<IFMChannels * InpWidth> >
convInp("StreamingFxdConvLayer_Batch.convInp");

    WidthAdjustedOutputStream <PECount, OFMChannels, OFMDim * OFMDim *
OFMChannels / PECount> mvOut (out, numReps);

    StreamingConvolutionInputGenerator_Batch<ConvKernelDim,
        IFMChannels, IFMDim, OFMDim, InpWidth>(in, convInp, numReps);

    WidthAdjustedInputStream <IFMChannels * InpWidth, SIMDWidth * InpWidth,
OFMDim * OFMDim * ConvKernelDim * ConvKernelDim> mvIn (convInp, numReps);

    StreamingFxdMatrixVector_Batch<InpWidth, InpIntWidth, SIMDWidth,
PECount, AccWidth, AccIntWidth, MatrixW, MatrixH, WMemCount,
TMemCount>(mvIn, mvOut, weightMem, thresMem, numReps * OFMDim * OFMDim);
}

```

注：卷积层代码中出现了四个对象在相应的文件中。其中

WidthAdjustedOutputStream、WidthAdjustedInputStream 在文件 Streamtools.h 中进行了定义，StreamingConvolutionInputGenerator\_Batch 在文件 Slidingwindow.h 文件中，StreamingFxdMatrixVector\_Batch 在 matrixvector.h 文件中。

### 池化层：

```

* @file maxpool.h
*
* Library of templated HLS functions for BNN deployment.
* This file implement the BNN maxpool layer
*

```



```

*
*****
*****/

template<unsigned int ImgDim, unsigned int PoolDim, unsigned int
NumChannels>
void StreamingMaxPool(stream<ap_uint<NumChannels> > & in,
    stream<ap_uint<NumChannels> > & out)
{
    CASSERT_DATAFLOW(ImgDim % PoolDim == 0);
    // need buffer space for a single maxpooled row of the image
    ap_uint<NumChannels> buf[ImgDim / PoolDim];
    for(unsigned int i = 0; i < ImgDim / PoolDim; i++)
    {
        #pragma HLS UNROLL
        buf[i] = 0;
    }

    for (unsigned int yp = 0; yp < ImgDim / PoolDim; yp++)
    {
        for (unsigned int ky = 0; ky < PoolDim; ky++)
        {
            for (unsigned int xp = 0; xp < ImgDim / PoolDim; xp++)
            {
                #pragma HLS PIPELINE II=1
                ap_uint<NumChannels> acc = 0;
                for (unsigned int kx = 0; kx < PoolDim; kx++) {
                    acc = acc | in.read();
                }
                // pool with old value in row buffer
                buf[xp] |= acc;
            }
        }

        for (unsigned int outpix = 0; outpix < ImgDim / PoolDim; outpix++)
        {
            #pragma HLS PIPELINE II=1
            out.write(buf[outpix]);
            // get buffer ready for next use
            buf[outpix] = 0;
        }
    }
}

```

```
// calling 1-image maxpool in a loop works well enough for now
template<unsigned int ImgDim, unsigned int PoolDim, unsigned int
NumChannels>
void StreamingMaxPool_Batch(stream<ap_uint<NumChannels> > & in,
    stream<ap_uint<NumChannels> > & out, unsigned int numReps)
{
    for (unsigned int rep = 0; rep < numReps; rep++)
    {
        StreamingMaxPool<ImgDim, PoolDim, NumChannels>(in, out);
    }
}
```

### 全连接层:

```
/*
*****
*
*
* @file fclayer.h
*
* Library of templated HLS functions for BNN deployment.
* This file lists a set of convenience funtions used to implement fully
* connected layers
*****
*****/

// helper function for fully connected layers
// instantiates matrix vector unit plus data width converters
template<unsigned int InStreamW, unsigned int OutStreamW,
    unsigned int SIMDWidth, unsigned int PEOCount,
    unsigned int PopCountWidth,
    unsigned int MatrixW, unsigned int MatrixH,
    unsigned int WMemCount, unsigned int TMemCount>
void StreamingFCLayer_Batch(stream<ap_uint<InStreamW> > & in,
    stream<ap_uint<OutStreamW> > & out,
    const ap_uint<SIMDWidth> weightMem[PECount][WMemCount],
    const ap_uint<PopCountWidth> thresMem[PECount][TMemCount],
    const unsigned int numReps)
{
    #pragma HLS INLINE
    unsigned const InpPerImage = MatrixW / InStreamW;
    unsigned const OutPerImage = MatrixH / PEOCount;
```

```

    WidthAdjustedInputStream <InStreamW, SIMDWidth, InpPerImage> wa_in
(in, numReps);
    WidthAdjustedOutputStream<PECount, OutStreamW, OutPerImage> wa_out(out,
numReps);

    StreamingMatrixVector_Batch<SIMDWidth, PECount, PopCountWidth, MatrixW,
MatrixH, WMemCount, TMemCount> (wa_in, wa_out, weightMem, thresMem,
numReps);

}

// helper function for fully connected layers with no activation
// instantiates matrix vector unit plus data width converters
template<unsigned int InStreamW, unsigned int OutStreamW,
        unsigned int SIMDWidth, unsigned int PECount,
        unsigned int PopCountWidth, unsigned int MatrixW,
        unsigned int MatrixH, unsigned int WMemCount>

void StreamingFCLayer_NoActivation_Batch(stream<ap_uint<InStreamW> > & in,
        stream<ap_uint<OutStreamW> > & out,
        const ap_uint<SIMDWidth> weightMem[PECount][WMemCount],
        const unsigned int numReps)
{
#pragma HLS INLINE
    stream<ap_uint<SIMDWidth> > in2mvu("StreamingFCLayer_NoAct_Batch.in2m
vu");
    stream<ap_uint<PECount * PopCountWidth> > mvu2out(
        "StreamingFCLayer_NoAct_Batch.mvu2out");
    const unsigned int InpPerImage = MatrixW / InStreamW;
    StreamingDataWidthConverter_Batch<InStreamW, SIMDWidth,
InpPerImage>(in, in2mvu, numReps);
    StreamingMatrixVector_NoActivation_Batch<SIMDWidth, PECount,
PopCountWidth, MatrixW, MatrixH, WMemCount>(in2mvu, mvu2out, weightMem,
numReps);
    const unsigned int OutPerImage = MatrixH / PECount;
    StreamingDataWidthConverter_Batch<PECount * PopCountWidth, OutStreamW,
OutPerImage>(mvu2out, out, numReps);
}

```

注：全链接层中出现了六个对象在相应的文件中。WidthAdjustedInputStream、WidthAdjustedOutputStream 在文件 Streamtools.h 中进行了定义，StreamingMatrixVector\_Batch、StreamingMatrixVector\_NoActivation\_Batch 在 matrixvector.h 文件中，StreamingDataWidthConverter\_Batch 在 Streamtools.h 文

件中。

### 9.2.2 协同任务软件实现

处理系统PS和处理逻辑PL之间任务协同和调用需要依据异构平台所提供的语言实现。本段使用第10章介绍的交通标志识别系统TSR，展示使用Python语言实现硬件和软件之间的协同。系统TSR实现所选取的异构系统平台PYNQ-Z1开发板顶层支持Python语言，因此使用平台提供的Python语言，实现调用硬件IP核以及C++代码，以便系统集成。整个代码有109行，这里仅提供前6行，全代码见本书附录三：TRS.py。

```
1 from pynq import Overlay, PL #提供的包，用于调用硬件核 PYNQIP
2 from PIL import Image #用于图片处理
3 import numpy as np
4 import cffi #用于调用 C++代码
5 import os
6 import tempfile
7
8 BNN_ROOT_DIR = os.path.dirname(os.path.realpath(__file__))
9 BNN_LIB_DIR = os.path.join(BNN_ROOT_DIR, 'libraries')
10 BNN_BIT_DIR = os.path.join(BNN_ROOT_DIR, 'bitstreams')
11 BNN_PARAM_DIR = os.path.join(BNN_ROOT_DIR, 'params')
12
13 RUNTIME_HW = "python_hw"
14 RUNTIME_SW = "python_sw"
15 _ffi = cffi.FFI()
16
```

TRS.py 前 16 行代码

## 第 9.3 节 软硬件间通信设计

异构系统平台通常有两个系统：处理系统(PS)和处理逻辑(PL)，通过数据流完成数据传输。完成数据间的通信需要数据传输总线、传输协议、传输方式和传输接口。本段简单介绍这些基本内容，更详细的内容可以参阅文献[7]的第11章。

### 9.3.1 数据传输总线

处理系统PS与处理逻辑PL间通信实质是通信代码指令对通信电路和通信数据存储区的操作。通信电路和通信数据存储区相对于微处理器MPU来说是输入



次译码以选择相应的 APB 设备。

### 9.3.2 数据传输协议

Xilinx 为 ZYNQ 系列的 PL 和 PS 通信引入了 AXI 接口设计,帮助用户实现 ARM 和 FPGA 之间高速数据通信。

AXI 是 ARM 公司提出的 AMBA3.0 协议中一部分。它是一种总线协议,主要用于描述主设备和从设备之间通过握手信号建立连接的数据传输方式。该协议是支持高带宽低延迟的片内总线传输方式。

目前最为流行的为 AMBA 4 中新增加的三个新接口协议:AXI4;AXI4-Lite 和 AXI4-Stream。

AXI4 协议在用于多个主接口时,可提高互连的性能和利用率。它支持高达 256 位的突发传输,能够发送服务质量信号,支持多区域接口,有助于最大化性能和能效。

AXI4-Lite 是 AXI4 协议的子协议,适用于与组件中更简单且更小的控件寄存器式的接口通信。AXI4-Lite 接口的所有事务突发长度均为 1,所有数据存取大小均与数据总线的宽度相同,不支持独占访问。

AXI4-Stream 协议可用于从主接口到辅助接口单向数据传输,可显著降低信号路由速率。它使用同一组共享线支持单数据流和多数据流,在同一互连内支持多个数据宽度。

这三种总线的特性如表 9-1 所示。

表 9-1 三种 AXI 总线特性

AXI 协议	特性	控制方式	适合场景
AXI4	地址/突发数据传输 可对一片地址连续进行 一次性读写	内存映射 ARM 将用户自定义 IP 编 入某一地址进行访问	高性能内存映射 需求
AXI4-Lite	地址/单数据传输 一次读写一个字(32bit)	内存映射 ARM 将用户自定义 IP 编 入某一地址进行访问	简单的低吞吐量 内存映射通信(例 如,往来于控制和 状态寄存器)
AXI4-Stream	突发数据传输 连续流接口,不需要地址 线	需要转换装置实现内存 映射到流式接口的转换	高速流数据(如, 视频流处理)

### 9.3.3 数据传输方式

在实际系统中，总线访问时长因传输方式不同而有较大差异。在目前的处理器系统中，MPU 与内部存储器之间采用高速总线，而与外部设备连接的总线通常速率低一些。对外部访问有两种方式：一种是直接内存访问 DMA (Direct Memory Access) 方式，进行批量传输；另一种是非 DMA 方式，进行单独传输。

DMA 方式由 DMA 控制器进行操作，传输操作周期短、速率快。DMA 传输的源和目的地都是连续地址，访问分散地址的存储空间。DMA 方式要求 MPU 必须先将所传输的数据放在本地存储器中，增加了软件资源。

非 DMA 方式由 MPU 进行操作，传输操作周期相对较长、速率慢，但数据地址可以是分散的，计算结果可以直接进行传输，本地可以不需要存储空间，不增加软件资源。

### 9.3.4 数据传输接口

将硬件电路作为一个外部设备，通过接口的寄存器进行数据交换和命令启动。外部设备的接口大都采用寄存器方式进行数据交互及控制操作，占用较少存储空间。每个接口电路至少含有两个寄存器，即控制寄存器 CR (Control Register) 和数据寄存器 DR (Data Register)。地址总线 AB (Address Bus)，数据总线 DB (Data Bus)，低读控制线 nRD (ni1 ReaD)，低写控制线 nWR (ni1 WRite)。

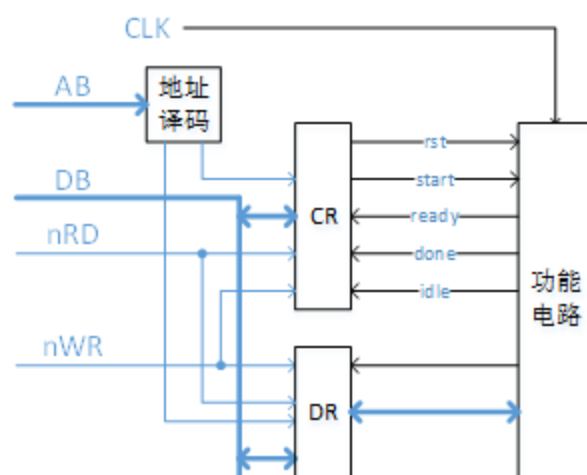


图 9-4 基本接口示意图

将硬件模块所需要的控制信号与 CR 相连，CR 中采用两个只写位来连接重启 (rst) 和启动 (start) 两个控制信号，采用三个只读位来连接准备好 (ready)、

完成 (done) 和空闲 (idle) 三个状态信号。

MPU 通过总线读写控制寄存器 CR 实现对硬件电路的控制。

数据寄存器 DR 主要是软件与硬件之间数据交换通道, MPU 写操作实现软件向硬件传送数据, 读操作实现硬件向软件传送数据。

为了加快数据传输, 可以采用双口 RAM 作为数据通道的存储器, 如图 9-5 所示, 可以实现硬件功能电路的高速处理及控制的要求。需要提高实时性时, 可以由功能电路提供中断信号供软件中断进行数据交互。

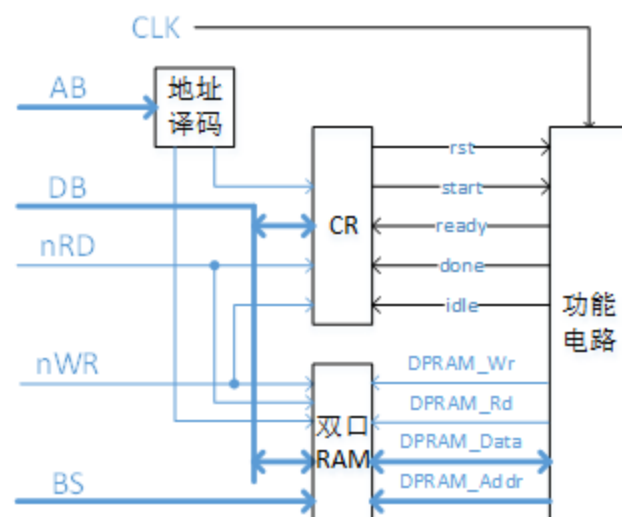


图 9-5 基于双口 RAM 数据交互的接口示意图

AXI 接口是 AXI 协议的物理实现, 即针对协议在硬件上实现的传输接口设计。在 ZYNQ 中用硬件实现了 9 个物理接口, 包括 4 个 AXI-GP 接口 (AXI-GP0~AXI-GP3), 4 个 AXI-HP 接口 (AXI-HP0~AXI-HP3) 和 1 个 AXI-ACP 接口。AXI 接口分布如图 9-5 所示。

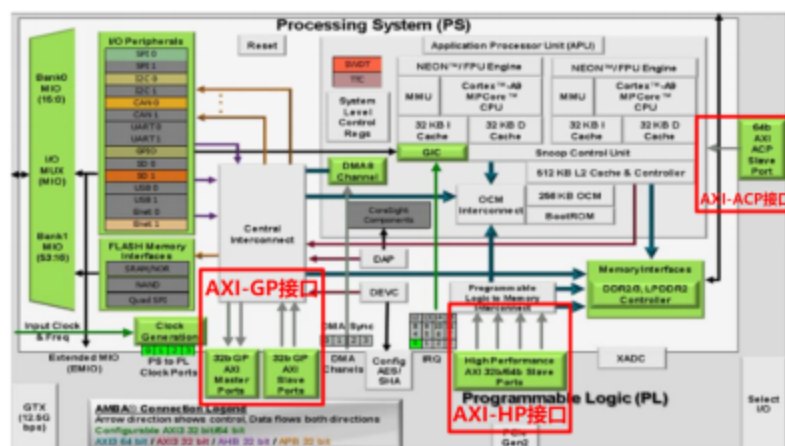


图 9-6 ZYNQ 芯片 AXI 接口分布图



图 9-6 可以总结出 ZYNQ 芯片 AXI 接口特性如表 9-2 所示。

表 9-2 ZYNQ 芯片 AXI 接口特性

名称	AXI 接口	个数	属性	理论带宽
通用端口	AXI-GP	4	2 个主接口 2 个从接口	32bit, 600MB/s
高性能端口	AXI-HP	4	从接口	32bit/64bit, 1200MB/s
加速一致端口	AXI-ACP	1	从接口	64bit, 1200MB/s

由表 9-2 可以看出有两个 AXI-GP 接口是主接口 (Master Port)，其余 7 个是从接口 (Slave Port)。PS 可以主动通过主接口访问 PL 逻辑。在其余从接口中，PS 只能被动接受来自 PL 的读写。

9.3.5 数据传输例子

下面以 ZYNQ702 开发板为例，通过一个实例来完成通信的设计。

在 ZYNQ7020 中，双口 RAM 是由在 PL 端的 BRAM 构建而成，PS 端通过工作时钟为 100MHz 的 AXI 总线与 BRAM 控制器相联，由 BRAM 控制器对该 BRAM 进行读写操作。其中

(1)非 DMA 方式

在 Vivado 中构建如图 9-7（同图 4-4）所示的测试电路图，其中 ZYNQ7 Processing System是 ARM CORTEX A9 硬核，其它单元均为 PL 资源。AXI BRAM Controller 主要负责 AXI 总线与 Block Memory 总线的转换。

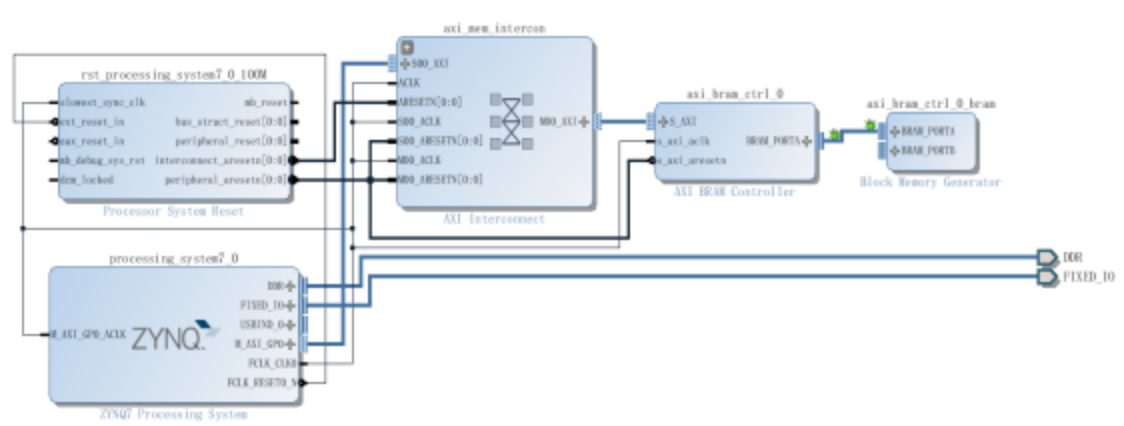


图 9-7 非 DMA 方式软硬通信测试电路连接图

BRAM 端口 A 为 PS 使用端口，端口 B 被 PL 逻辑处理电路直接使用。软硬件间通信主要体现在 MPU 对 BRAM 端口 A 的访问。

非 DMA 程序 C 代码如下:

```
//XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR 是 axi_bram_ctrl_0 的地址 0x40000000
#define XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR ((volatile int *)0x40000000)

int main(){

int x;

init_platform();

XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR[0]=0; //存储地址为 0x40000000
XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR[1]=1; //存储地址为 0x40000004
XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR[2]=2; //存储地址为 0x40000008
XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR[3]=3; //存储地址为 0x4000000C
for(x=0; x<5; x++); //延时

x=XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR[0]; //存储地址为 0x40000000
x=XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR[1]; //存储地址为 0x40000004
x=XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR[2]; //存储地址为 0x40000008
x=XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR[3]; //存储地址为 0x4000000C
x=x+1; //用来保证最后一个读操作执行

cleanup_platform();

return 0;

}
```

## (2) DMA 方式

在 Vivado 中构建如图 9-8 (同图 4-7) 所示的测试电路图, 增加了 AXI Central DMA 和 AXI SmartConnect 部件。

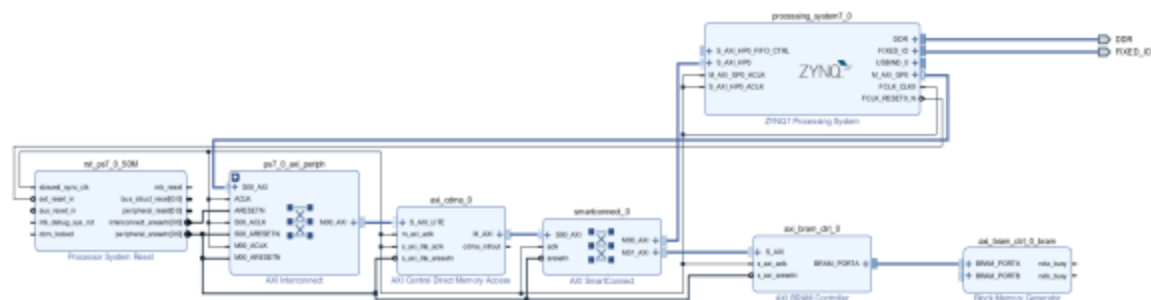


图 9-8 DMA 方式软硬通信测试电路连接图

与非 DMA 方式相似，BRAM 端口 A 为 PS 使用端口，端口 B 被逻辑处理电路直接使用。软硬件间通信主要体现在 MPU 对 BRAM 端口 A 的访问。DMA 程序 C 代码如下：

```
#define PL_BRAM_Addr 0xC0000000 // 需要单独定义

#define BUFLen 4

int main()
{
    XAxiCdma_Config *axi_cdma_cfg;

    XAxiCdma axi_cdma;

    u32 *buf = (u32*)0x11000000;

    int i;

    for (i=0;i<BUFLen;i++)    buf[i]=i;

    init_platform();

    axi_cdma_cfg = XAxiCdma_LookupConfig(XPAR_AXICDMA_0_DEVICE_ID);
    XAxiCdma_CfgInitialize(&axi_cdma, axi_cdma_cfg, axi_cdma_cfg->BaseAddress);
    XAxiCdma_IntrDisable(&axi_cdma, XAXICDMA_XR_IRQ_ALL_MASK);
    while (XAxiCdma_IsBusy(&axi_cdma));

    Xil_DCacheFlush();

    //ps to pl
    //第 1 次 DMA 传输
    XAxiCdma_WriteReg(axi_cdma.BaseAddr, XAXICDMA_SRCADDR_OFFSET, buf);
    XAxiCdma_WriteReg(axi_cdma.BaseAddr, XAXICDMA_DSTADDR_OFFSET, PL_BRAM_Addr);
    XAxiCdma_WriteReg(axi_cdma.BaseAddr, XAXICDMA_BTT_OFFSET, sizeof(u32)*BUFLen);
    //第 2 次 DMA 传输
    XAxiCdma_WriteReg(axi_cdma.BaseAddr, XAXICDMA_SRCADDR_OFFSET, buf);
    XAxiCdma_WriteReg(axi_cdma.BaseAddr, XAXICDMA_DSTADDR_OFFSET, PL_BRAM_Addr);
```

```

XAxiCdma_WriteReg(axi_cdma.BaseAddr,XAXICDMA_BTT_OFFSET,sizeof(u32)*BUFLEN);

Xil_DCacheFlush();

//pl to ps
//第 1 次 DMA 传输
XAxiCdma_WriteReg(axi_cdma.BaseAddr, XAXICDMA_SRCADDR_OFFSET, PL_BRAM_Addr);
XAxiCdma_WriteReg(axi_cdma.BaseAddr, XAXICDMA_DSTADDR_OFFSET, buf);
XAxiCdma_WriteReg(axi_cdma.BaseAddr,XAXICDMA_BTT_OFFSET,sizeof(u32)*BUFLEN);
//第 2 次 DMA 传输
XAxiCdma_WriteReg(axi_cdma.BaseAddr, XAXICDMA_SRCADDR_OFFSET, PL_BRAM_Addr);
XAxiCdma_WriteReg(axi_cdma.BaseAddr, XAXICDMA_DSTADDR_OFFSET, buf);
XAxiCdma_WriteReg(axi_cdma.BaseAddr,XAXICDMA_BTT_OFFSET,sizeof(u32)*BUFLEN);
Xil_DCacheFlush();

cleanup_platform();

return 0;
}

```

## 第 9.4 节 本章小结

本章介绍了系统软件实现、硬件实现和通信实现。软件实现使用 C++或 Python 等编程语言实现其代码。硬件实现是使用 Vivado 工具将 C++代码自动生成硬件 IP 核，包括语言版和图形版。处理系统与处理逻辑间通信使用 C 语言编写的代码。使用 Vivado 工具实现处理系统与处理逻辑间通信电路图是实现整个系统的关键。

### 习题:

**9.1** 从第 3 章第 3.1 节中的例子和作业中选两个题目使用 Vivado 工具生成 IP 核（Verilog 版和图形版两种），同时与使用 Verilog 语言编程实现的 Verilog 代码进行比较，指明其相同性和差异性。

**9.2** 使用 Vivodal 工具生成卷积神经网络的卷积层、池化层、全连接层硬件 IP 核代码和图形；

**9.3** 使用 Vivodal 工具生成 DMA 和非 DMA 通信电路连接图。