第 5 章

CHAPTER 5

基于 Python 的 I/O 交互

前面章节介绍了如何使用 Vivado 设计自己的硬件模块，以及如何通过 AxiGPIO 将自己设计的模块与 PS 连接起来。本章主要介绍基于 Jupyter Notebook，通过 Python 代码的方式操作 AxiGPIO 与用户设计的硬件模块进行交互。

5.1 Jupyter Notebook 介绍

Jupyter Notebook 是 Fernando Perez 发起的 IPython 项目。IPython 是一种交互式 shell，与普通的 Python shell 相似却具有一些很好的功能，如语法高亮显示和代码补全等。Jupyter Notebook 的工作方式是通过 Web 浏览器打开并编辑 Python 源代码，将其消息发送给 IPython 内核，由内核执行代码，然后将结果发送回浏览器的 Notebook。IPython 内核是在后台运行的 IPython 应用程序，本书介绍的 IPython 内核是运行在 Zynq-7000 的 PS 上的。

Jupyter 的核心是 Notebook 服务器，通过浏览器连接到该服务器。而 Notebook 呈现为 Web 应用。保存 Notebook 时，它将作为 JSON 文件（文件扩展名为.ipynb）写入到该服务器中。

采用 Jupyter Notebook 架构的一个优点是：内核无须运行 Python。由于 Notebook 和内核相互独立，因此可以在两者之间发送任何语言的代码。例如，早期的两个非 Python 内核分别是 R 语言和 Julia 语言。使用 R 内核时，用 R 编写的代码将发送给执行该代码的 R 内核，这与在 Python 内核上运行 Python 代码完全一样。IPython Notebook 如今已被更名为 Jupyter Notebook，因为 Notebook 变得与编程语言无关。新的名称 Jupyter 由 Julia、Python 和 R 组合而成。

第二个优点是可以在任何地方运行 Notebook 服务器，并且可通过互联网访问服务器。通常会在存储所有数据和 Notebook 文件的自有计算机上运行服务器。但是，也可以在远程计算机或云实例（如本书介绍的 PYNQ 云节点）上设置服务器。之后，就可以在世界上任何地方通过浏览器访问 Notebook。

第三个优点是由于它们比 IDE 平台更具交互性，因此它们被广泛地应用于教学场景。

5.1.1 Jupyter 组件

Jupyter 组件的内容：

(1) Web 浏览器：交互式 Web 应用程序，用于交互式编写和运行代码以及编写笔记本文档。

(2) 内核：由 Notebook Web 应用程序启动的独立进程，它以给定语言运行用户代码并将输出返回给 Notebook Web 应用程序。对于 PYNQ 来说，它是用 Python 编写的，是 Jupyter Notebook 的默认内核，也是 PYNQ 发行版中为 Jupyter Notebook 安装的唯一内核。

(3) Notebook 文档：包含 Notebook Web 应用程序中所有内容表示的自包含文档。

5.1.2　Notebook 基础

Notebook 服务器在 PYNQ 上 PS 端 ARM 处理器上运行。当本地电脑以任何方式与 PYNQ 开发板连接时，使用任意浏览器访问 pynq:9090 即可连接到 Jupyter Notebook。图 5-1 为 Jupyter Notebook 界面。

(1) 要创建新的 Notebook 文档，请单击列表顶部的 New 按钮，然后从下拉列表中选择，如图 5-2 所示。

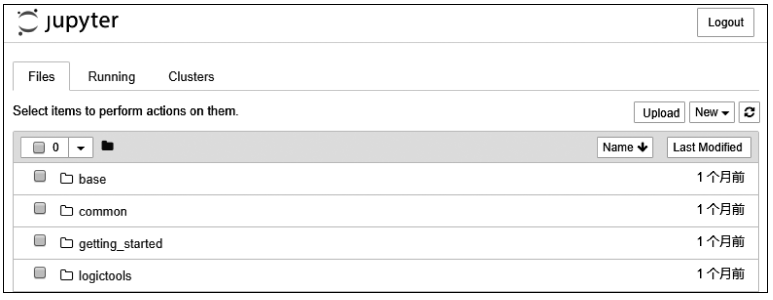

图 5-1　Jupyter Notebook 界面

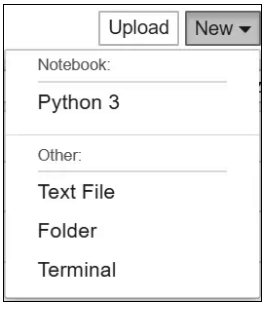

图 5-2　新建文档

(2) 列表中图标为绿色，且显示绿色 Running 文本的文件是正在运行的文件，如图 5-3 所示。直到明确关闭为止，这些文件会一直运行。

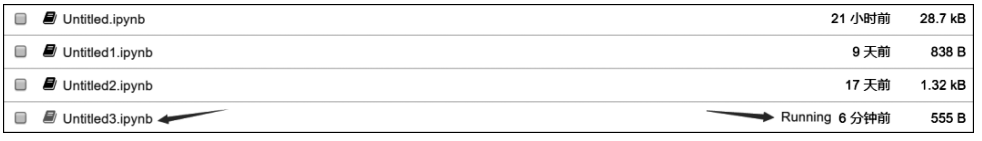

图 5-3　正在运行的文件

(3) 要查看所有正在运行的文件及其目录，可以选择 Running 选项，如图 5-4 所示。

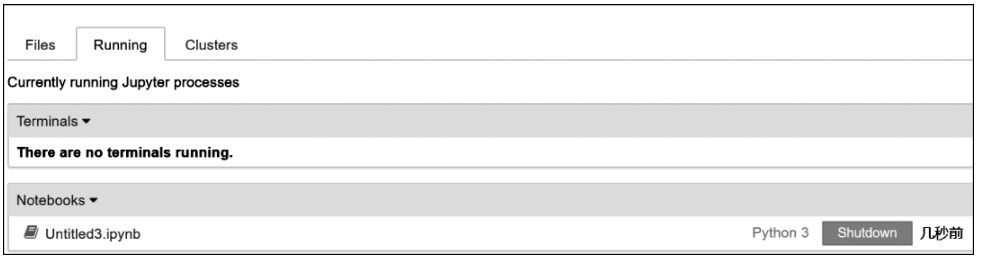

图 5-4　查看运行文件

5.1.3 Notebook 用户界面

创建文件或打开现有文件后，将进入 Notebook 用户界面。Notebook 用户界面具有以下主要区域：

(1) Notebook 的名称。

(2) 主工具栏提供了保存、导出、重载 Notebook，以及重启内核等选项。

(3) 快捷键。

(4) Notebook 主要区域，包含了 Notebook 的内容编辑区，界面如图 5-5 所示。

图 5-5 用户界面

Jupyter Notebook 有两种模式：编辑模式和命令模式。

1. 编辑模式

编辑模式由绿色单元格边框指示。当单元格处于编辑模式时，可以像普通文本编辑器一样输入单元格。按 Enter 键或单击单元格的编辑器区域可进入编辑模式，如图 5-6 所示。

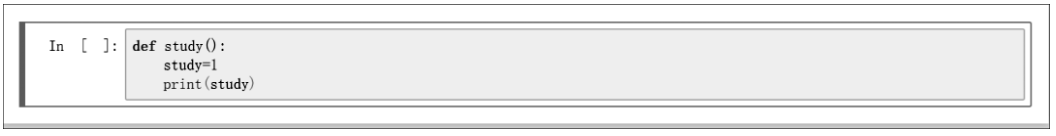

图 5-6 编辑模式

2. 命令模式

命令模式由带有蓝色左边距的灰色单元格边框表示，如图 5-7 所示。

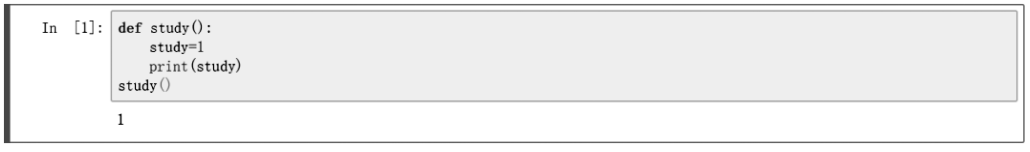

图 5-7 命令模式

Notebook 可以通过修改之前的单元格，对其重新运算，同时更新整个文档。例如将图 5-7 中的 print(study)改为 print("love")，然后按下 Shift+Enter 组合键重新计算该单元格，输出结果马上就更新成了"love"。通过这种方法可以对某一模块进行多次修改，可大大减少程序开发与调试的时间。

Notebook 中还有 Header 单元格和 Markdown 单元格两种不同的单元格，可以利用它们对代码进行美化，如图 5-8 所示。

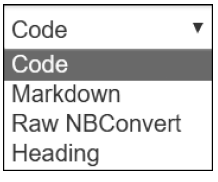

图 5-8 其他单元格类型

Heading 可以创建不同层级的标题，并将之与代码相区分，

使得代码的层次更加清晰。Markdown 可以对解释文本进行漂亮的渲染，产生美观的注释，效果如图 5-9 所示。

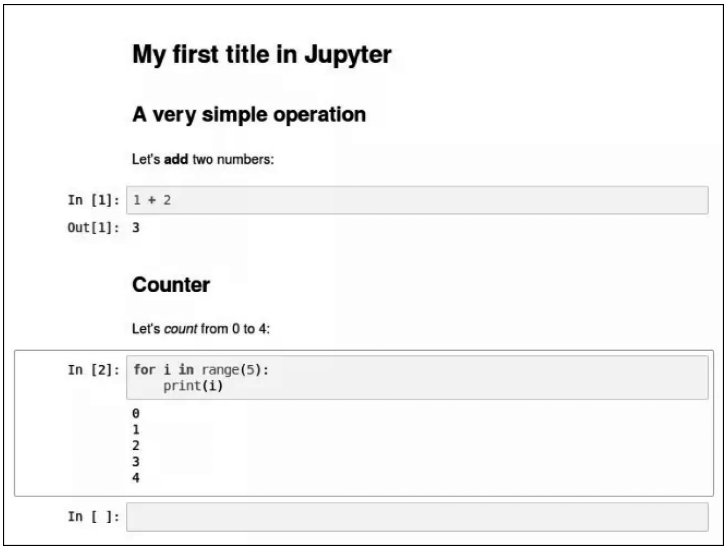

图 5-9　Heading 和 Markdown 使用效果

5.2　使用 PYNQ Overlay 加载流文件

用户设计的比特流文件可以通过 Vivado 工具下载到 PL 中，也可以在 Jupyter Notebook 中直接使用 Python 代码进行下载。PYNQ 中已经内置了 Jupyter Notebook。

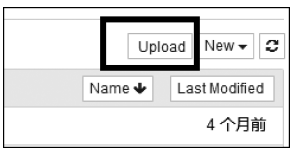

图 5-10　上传文件

首先将 Vivado 中生成的 *.bit 和 *.tcl 这两个文件通过 Jupyter Notebook 导入到 PYNQ 板卡上。如图 5-10 所示，在 Jupyter Notebook 中单击 Upload，选择文件路径将上述两个文件加载进 Jupyter Notebook 可访问的文件夹中。

在使用 Python 代码下载比特流文件时，首先要引入 PYNQ 包中的 Overlay 文件，然后通过 Overlay 来加载比特流到 PYNQ 中。如示例中将 base.bit 流文件加载到 PYNQ 板卡上的 FPGA 芯片中去。想要配置自己的比特流文件，只需要生成命名相同的 *.bit 与 *.tcl 文件。例如：

```
from pynq import Overlay
overlay = Overlay("base.bit")     # 加载流文件"base.bit"。Tcl 文件也是需要的，会进行解析
```

5.3　Python 引脚绑定

AxiGPIO 接口模块提供了从外部通用外设（如 led、按钮、通过 AxiGPIO 控制器 IP 连接到 PL 的开关）读取、写入和接收中断的方法。AxiGPIO 模块控制 PL 中 AxiGPIO 控制

器的实例。AxiGPIO 框图如图 5-11 所示，每个 AxiGPIO 最多有两个通道，每个通道位宽最大为 32 位。

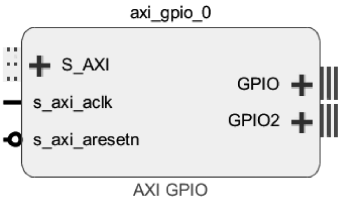

图 5-11　AxiGPIO 框图

通过 Python 调用 read()和 write()用于在通道上读写数据。

setdirection()和 setlength()可用于配置 IP。

AxiGPIO 的方向可以是 in、out 和 inout；AxiGPIO 默认的方向是 inout。设定了 in 或 out 后只允许对 IP 分别进行读和写操作，如果读取'out '或写入'in'将会报错。

官方文档中包含的例子介绍了如何使用 AxiGPIO 模块。在实践中，可以使用 LED、按钮、开关和 RGBLED 等来扩展 AxiGPIO，这些外围设备需要使用 Overlay 来加载 bit 文件。在加载 Overlay 之后，通过将 AxiGPIO 控制器的名称传递给实例化的 AxiGPIO 实例。

加载 bit 文件和端口分配的示例如下：

```
from pynq import Overlay
from pynq.lib import AxiGPIO
ol = Overlay("base.bit")

led_ip = ol.ip_dict['gpio_leds']
switches_ip = ol.ip_dict['gpio_switches']
leds = AxiGPIO(led_ip).channel1
switches = AxiGPIO(switches_ip).channel1
```

首先将 bit 文件通过 Overlay 函数赋值给变量 o1，再通过 led_ip = o1.ip_dict['gpio_leds']将采用 Vivado 设计时命名为'gpio_leds'的 AxiGPIO 模块赋值到 Jupyter Notebook 中的 led_ip 端口，完成 Jupyter 与 AxiGPIO 的连接。使用 leds = AxiGPIO(led_ip).channel1 语句对 AxiGPIO 的通道 1 进行定义。

简单的读写操作的示例如下：

```
mask = 0xffffffff
leds.write(0xf, mask)
switches.read()
```

AxiGPIO 需要包含掩码进行写操作。首先定义一组掩码值，再对定义完成的端口通过 write()和 read()进行数据的读写操作。

5.4　基于 Python 调试组合逻辑

通过 Python 调试组合逻辑方法相对比较简单，本书涉及的组合逻辑实验基本都使用相同的调试文件。文件主要使用 read()和 write()两个模块。下面的例子来自第 8 章加法器实验的 Python 程序。

编辑 Python 代码时需要调用很多的库函数，在 PYNQ 平台上内置了很多的库函数，使用时可以直接调用。

```
from pynq import Overlay                          #调用 Overlay 库
from pynq.lib import AxiGPIO                      #调用 AxiGPIO 库
overlay = Overlay("/home/xilinx/jupyter_notebooks/….bit")    #bit 文件路径
overlay?                                          #显示出 bit 文件中的各种 IP 信息
```

对各个端口进行命名：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']    #将使用的 AxiGPIO 模块 Jupyter Notebook 中进行命名
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']

a = AxiGPIO(gpio_0).channel1              #对 AxiGPIO 上的各个通道进行命名
b = AxiGPIO(gpio_0).channel2
cin = AxiGPIO(gpio_1).channel1
s = AxiGPIO(gpio_1).channel2
cout = AxiGPIO(gpio_2).channel1
```

对 AxiGPIO 的输出端口使用 write()进行赋值，将数据送到 PL 端进行处理。待 PL 端处理结束后将结果送回 AxiGPIO 的输入端，并通过 read()读取出处理后的结果。通过 AxiGPIO 输出数据时需要掩码才能输出，一般默认掩码每个位都为 1。Python 中 4 位十六进制的掩码为 0xf，用户可以根据输出数据的位数改变掩码的位数。如第 4 章介绍，Channel 的位宽是用户在 Vivado 中调用 AxiGPIO 时设置好的。

```
mask = 0xf                                #AxiGPIO 掩码
a.write(8,mask)                           #将 AxiGPIO 的输出端口赋值为 8
b.write(9,mask)
cin.write(0,mask)
```

GPIO 接收端得到数据后不会显示出来，此时需要使用 print()函数来显示得到的结果。

```
print(cout.read())                         #将 AxiGPIO 接收端得到的结果显示出来
print(s.read())
```

5.5　基于 Python 调试时序逻辑

时序逻辑电路通过 Python 进行交互的时候，由于 PL 端并行度高，数据处理速度快，而 PYNQ 平台上的 AxiGPIO 模块有延时。当数据处理量较大时，有时会造成 AxiGPIO 读取时序电路数据时出现数据遗漏，造成数据不完整。为了解决这一问题，将时序电路的时钟输入端口通过 Python 调试文件进行输出，保证 Python 调试程序的时钟与 PL 端相一致。让每个时钟周期的数据都能被记录下来。下面使用包含时序电路的计数器 Python 调试模块进行介绍。

读取比特文件和对各端口进行命名这两部分与逻辑电路时基本相同，在此不再赘述。

```
from pynq import Overlay
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/…bit")
overlay?

gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']

clk = AxiGPIO(gpio_0).channel1              #对时钟所在通道进行命名
rst_n = AxiGPIO(gpio_0).channel2            #对复位所在通道进行命名
out0 = AxiGPIO(gpio_1).channel1
```

在时序逻辑中引入时间函数来进行延时,保证数据有足够的时间写入读出,防止出现错误。复位端口低电平有效,不复位时该端口要置1。

```
import time                                 #调用时间函数
mask = 0b1                                  #AxiGPIO掩码
timedelay = 0.1                             #设置时间延时
rst_n.write(0b0,mask)                       #进行复位
rst_n.write(0b1,mask)
```

时钟的变化为高低电平交替,可通过Python调试文件设置时钟的变换,并通过For循环来模仿时钟连续性。

Python中for循环包含三个变量,第一位为循环的起始位,第二位为循环的次数,第三位为循环步长。当步长为1时每循环一次循环次数加1,如果步长为N,则每次循环后循环次数加N。

由于每个时钟周期后的数值都需要被记录,直接输出结果观察起来会很不方便,通过使用数组可以保证每个时钟的结果都单独保存。每次循环中都建立一个空白的数组,并通过append函数将数值添加到数组中保存起来。

```
import time
N = 30
timedelay = 0.1
sizes = range(0,N,1)                        #设置循环次数
mask = 0b1
for size in sizes:
clk.write(0b0,mask)                         #创造时钟
clk.write(0b1,mask)
tmp = []                                    #创建数组保存结果
a1 = out0.read()
tmp.append(a1)
print(tmp)                                  #打印数据,读出每个时钟中周期的结果
```

5.6　实例演示

在参照4.8节生成.bit和.tcl文件后，请查看第2章完成PYNQ实验环境的准备，之后按照以下步骤完成基于PYNQ的Python调试。

5.6.1　上传.bit和.tcl文件

通过选择New→Folder新建一个文件夹，选中这个文件夹将其重命名为Example。

将4.8节生成的design_1.bit和design_1.tcl上传到Jupyter Notebook新建的Example文件夹下，如图5-12所示。

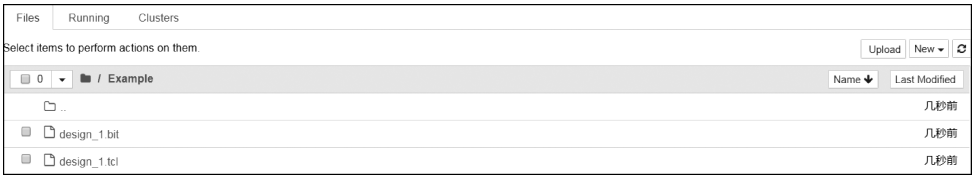

图5-12　上传相关文件

5.6.2　基于Python的I/O交互

基于Python的I/O交互过程如下：

1. 创建Notebook

首先通过选择New→Python 3创建一个Notebook，如图5-13所示。

图5-13　创建Python文件

2. 使用pynq Overlay加载比特流

代码如下：

```
from pynq import Overlay
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/Example/design_1.bit")
overlay?
```

通过“Overlay?”可以查看overlay的信息，在这里可以看到设计中使用到的两个AxiGPIO，如图5-14所示。

3. 对端口的访问

对于端口的访问要和原理图连接一致，代码如下：

```
Type:            Overlay
String form:     <pynq.overlay.Overlay object at 0xb431cad0>
File:            /usr/local/lib/python3.6/dist-packages/pynq/overlay.py
Docstring:
Default documentation for overlay /home/xilinx/jupyter_notebooks/Example/design_1.bit. The following
attributes are available on this overlay:

IP Blocks
----------
axi_gpio_0           : pynq.lib.axigpio.AxiGPIO
axi_gpio_1           : pynq.lib.axigpio.AxiGPIO
```

图 5-14　Overlay 信息

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_A = AxiGPIO(gpio_0).channel1
gpio_B = AxiGPIO(gpio_0).channel2
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_S = AxiGPIO(gpio_1).channel1
```

4. GPIO 口的读写

对 gpio_A 和 gpio_B 两个端口写入 0 和 1，通过读取 gpio_S 值来判断是否与预期的结果($S=\overline{A}B$)一致，进而判断设计是否正确。代码如下：

```
mask = 0b1
gpio_A.write(0b0,mask)
gpio_B.write(0b1,mask)
S = gpio_S.read()
print(S)
```

第6章 硬件描述语言简介

CHAPTER 6

硬件描述语言(Hardware Description Language,HDL)是以编写代码的方式来描述硬件模块及其功能的语言。相比原理图,硬件描述语言在描述大型复杂硬件系统时效率更高。常用的硬件描述语言有 Verilog HDL 和 VHDL。Verilog HDL 的优点是语法类 C,由于计算机类的学生通常具有 C 语言的编程基础,因此可以快速掌握 Verilog HDL 的编写。但缺点也是因为语法太像 C 语言,学生容易与软件编程混淆,从而用软件设计的思维来进行 Verilog HDL 编写,最终导致不可综合的设计。而 VHDL 相反,其语法更加严谨并需要一定的重新学习过程,但优点是没有软件编程的历史包袱,一开始学生就以硬件的思维来使用它。因为硬件描述语言本质上是对某个硬件模块的描述,因此 Verilog HDL 和 VHDL 逻辑上是等价的。总体来说,使用任何一种语言都是可以的,在同一个工程中,这两种设计语言也可以混合使用,只要同一个模块选定一种就可以。企业在招聘时也通常将这两种语言的技能同等看待。

本章仅介绍数字逻辑及计算机组成原理实验所需要的最基本的 HDL 知识,方便学生快速入门。鉴于网络上有大量公开的 HDL 教程,更为详细的介绍请自行参考相关资料。考虑到不同学校的使用习惯,这里给出了 Verilog 和 VHDL 对照的版本。也希望更有利于读者理解 HDL 是硬件描述的本质。

6.1 “模块”的描述

“模块”是 Verilog HDL 和 VHDL 描述的最基本单元,复杂的系统则可以通过模块的层次化嵌套使用来完成。如图 6-1 是两种语言对模块的描述:在 VHDL 中,模块描述的关键词叫 Entity,也就是实体,Entity 部分只关注该模块的外在表现如模块名称、输入/输出端口。而对于模块内部功能的详细描述是由 Entity 对应的 Architecture 部分来完成的。Architecture 的描述也分为两部分:在 Architecture 和 Begin 之间是对该模块所用的变量和信号进行的声明。Begin 之后才是真正的模块功能的详细描述。

Verilog HDL 的语法则要简洁一些。Verilog 中模块的关键字就是 Module,而对于该模块的输入/输出端口、内部所用的变量以及模块功能的详细描述都在 Module 和 Endmodule 之间的区域内完成。