**SPI控制器**

|  |  |  |  |
| --- | --- | --- | --- |
| Document Version Control | | | |
| Version | Date | Author | Changes |
| 1.0.0 | 2023.9.26 | 王雨霄 | 实现了SPI控制器 |

注：阅读本文档前，请先阅读SPI模块的相关文档。

1. **功能概述**

本模块用于实现PS端对SPI模块（PL端）工作的控制，将SPI模块工作所需的控制信号、数据信号挂载至AXI总线，并将SPI线上的传输状态同步至寄存器中以供PS端读取。

1. **接口概述**

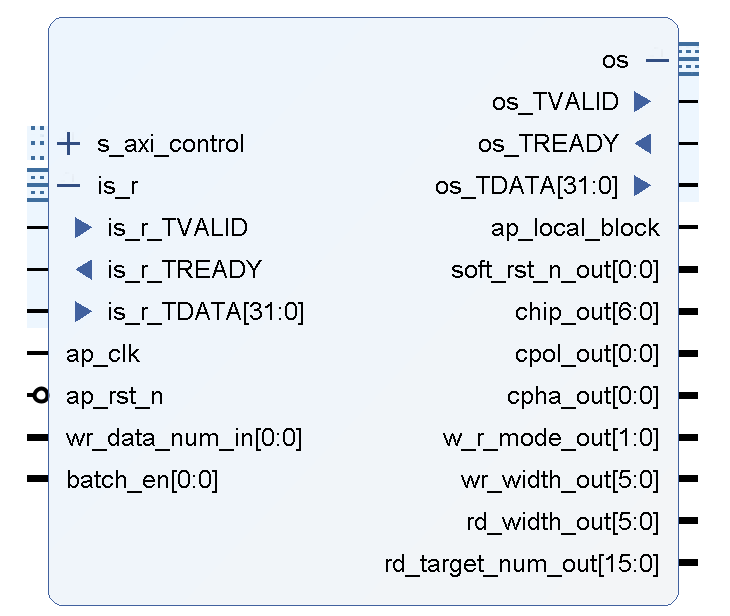
****

图1. 模块接口



图2. 接口功能表

注：

spi模块与spi\_regs模块配合使用时有两种方式：一是将spi模块的AXIS数据接口连接至spi\_regs模块上，此时PS端通过读写spi\_regs模块中的寄存器来实现与spi模块间的数据逐个读写（第三节）；二是将spi模块的AXIS数据接口连接至DMA上，此时PS端通过DMA传输来实现与spi模块间的数据批量读写（第四节），在此应用情景下，spi\_regs模块仅通过xx\_out信号来控制spi模块的工作模式，而不与spi模块间进行AXIS数据传输。

1. **逐次读写模式使用方法**

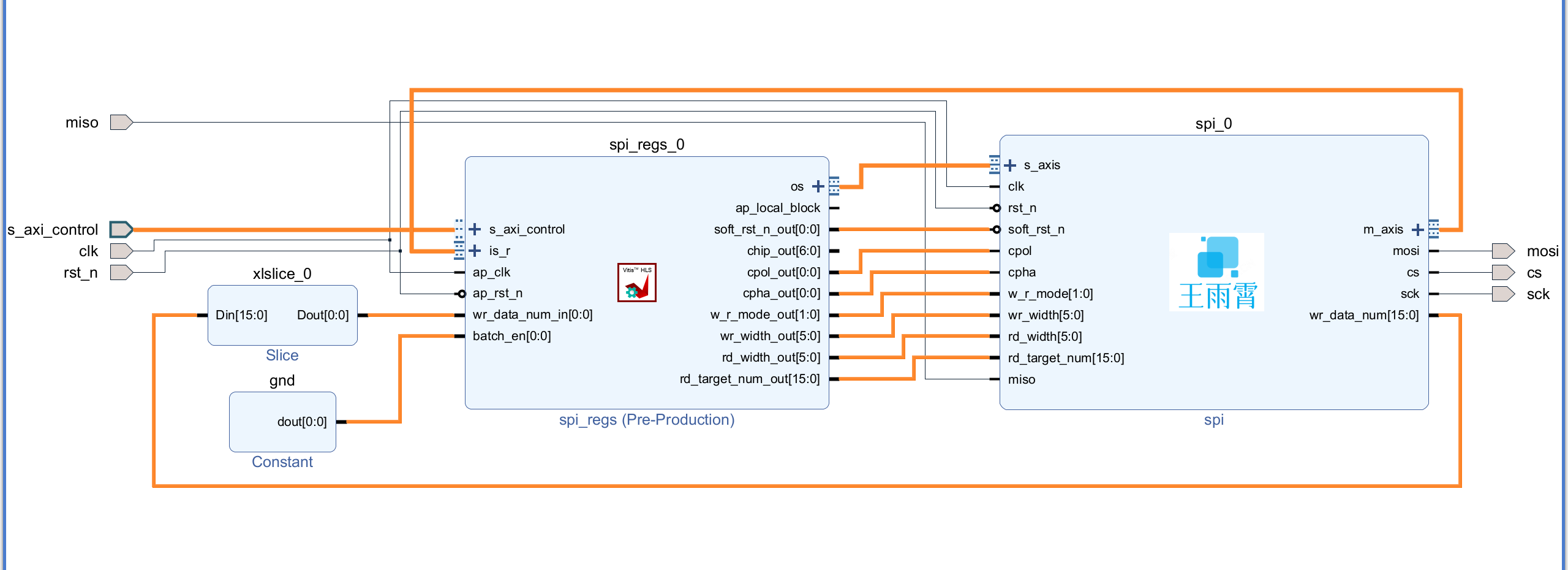


图3. 逐次读写模式的模块连接图

如图3所示，在逐次读写模式下，spi模块的数据信号、控制信号与状态信号全部连接至spi\_regs模块上，且spi\_regs模块的batch\_en信号被置为0。

spi\_regs模块的s\_axi\_control控制接口中包含如下寄存器：

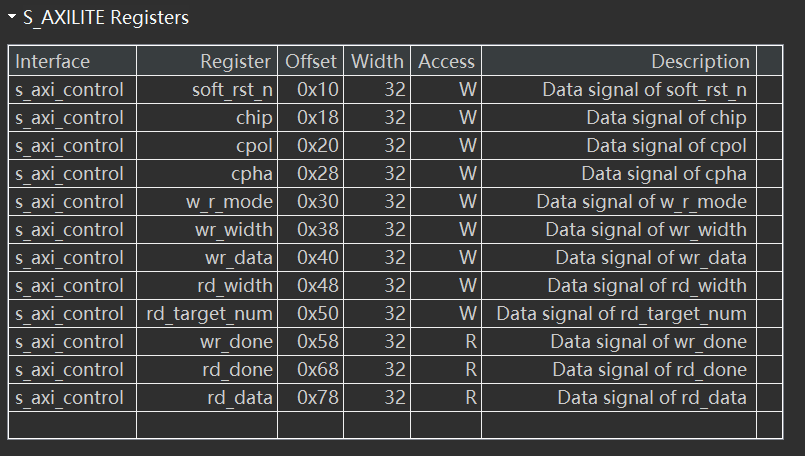


图4. spi\_regs的寄存器表

其中：

1、soft\_rst\_n为软复位控制寄存器，向该寄存器写入0值将触发spi模块的复位；

2、chip为片选控制寄存器，用于在一主多从的情况下切换spi控制对象（详情请见本段的7）；

3、cpol、cpha、w\_r\_mode、wr\_width、rd\_width、rd\_target\_num为spi工作模式控制寄存器，各寄存器及寄存器值的意义与spi模块的控制信号一一对应，详情可见spi模块的相关文档；

4、wr\_data为spi待发送数据寄存器，用于写入spi需要发送的数据；

5、rd\_data为spi已读取数据寄存器，用于存储spi读回的数据以供PS端读取；

6、wr\_done、rd\_done为状态寄存器，用于标志本次spi读/写的完成，从而实现PS端与PL端的软硬件状态同步。

7、一主多从的spi总线系统通常有两种实现方式：

一是将不同从机的miso、mosi、sck及cs信号分别连接至FPGA不同的IO口上（如图5所示），FPGA内部根据当前的chip值将mosi、sck及cs信号分配至相应IO口进行输出，并从多个miso输入中选择一路送入spi主机模块；



图5. 一主多从spi总线-方案一

二是将不同从机的miso、mosi及sck信号分别并联后连接至FPGA的三个IO口上（如图6、7所示），并通过FPGA内部逻辑或外部数据分配器将spi主机的cs信号根据chip值分配至相应的从机。



图6. 一主多从spi总线-方案二



图7. 一主多从spi总线-方案三

进行spi写操作时，应按照“执行软复位(向0x10寄存器写入0值)➡写入控制信号及数据信号(分别向0x18、0x20、0x28、0x30、0x38、0x40寄存器写入相应的控制信号及待发送的数据值)➡撤销软复位(向0x10寄存器写入1值)➡读取状态位(循环读取0x58寄存器的值，直至返回值为1)”的次序操作寄存器，在PYNQ中的示例驱动如下：

|  |
| --- |
| # base addr  SPI\_CONTROL = 0x40000000  # regs  soft\_rst\_n\_reg    = 0x10  chip\_reg          = 0x18  cpol\_reg          = 0x20  cpha\_reg          = 0x28  w\_r\_mode\_reg      = 0x30  wr\_width\_reg      = 0x38  wr\_data\_reg       = 0x40  rd\_width\_reg      = 0x48  rd\_target\_num\_reg = 0x50  wr\_done\_reg       = 0x58  rd\_done\_reg       = 0x68  rd\_data\_reg       = 0x78  # SPI从机列表(rd\_mode=0-只读模式;rd\_mode=1-指令控读模式)  chip\_settings = {      "MAX6675": {"chip\_num":0, "cpol":0, "cpha":0, "wr\_width":0, "rd\_mode":0, "rd\_comd\_width":0, "rd\_data\_width":16},      "BMI160":  {"chip\_num":1, "cpol":1, "cpha":1, "wr\_width":16, "rd\_mode":1, "rd\_comd\_width":8, "rd\_data\_width":8},  }  # 写寄存器  def write\_reg(base\_addr, offset, val):      mmio = MMIO(base\_addr, 64\*1024)   # 创建一个名为mmio的MMIO对象,可用地址空间为64KB(与VIVADO设计中的地址分配相吻合)  mmio.write(offset, val)           # 向地址为base\_addr+offset的寄存器写入val  # 读寄存器  def read\_reg(base\_addr, offset):      mmio = MMIO(base\_addr, 64\*1024)      read\_val = mmio.read(offset)  return read\_val  # 写SPI  def spi\_write(chip, wr\_data):      if chip in chip\_settings:          chip\_num = chip\_settings[chip]["chip\_num"]          cpol = chip\_settings[chip]["cpol"]          cpha = chip\_settings[chip]["cpha"]          wr\_width = chip\_settings[chip]["wr\_width"]      write\_reg(SPI\_CONTROL, soft\_rst\_n\_reg, 0x0)      write\_reg(SPI\_CONTROL, chip\_reg, chip\_num)      write\_reg(SPI\_CONTROL, cpol\_reg, cpol)      write\_reg(SPI\_CONTROL, cpha\_reg, cpha)      write\_reg(SPI\_CONTROL, w\_r\_mode\_reg, 0x1)      write\_reg(SPI\_CONTROL, wr\_width\_reg, wr\_width)      write\_reg(SPI\_CONTROL, wr\_data\_reg, wr\_data)      write\_reg(SPI\_CONTROL, soft\_rst\_n\_reg, 0x1)      wr\_done = read\_reg(SPI\_CONTROL, wr\_done\_reg)      while wr\_done!=1 :          wr\_done = read\_reg(SPI\_CONTROL, wr\_done\_reg)  print("Successfully write.")  spi\_write("BMI160", wr\_data) |

进行无需指令控制的spi读操作（如下图8所示）时，应按照“执行软复位(向0x10寄存器写入0值)➡写入控制信号 (分别向0x18、0x20、0x28、0x30、0x48、0x50寄存器写入相应的控制信号)➡撤销软复位(向0x10寄存器写入1值)➡读取状态位(循环读取0x68寄存器的值，直至返回值为1)”的次序操作寄存器；进行需要写入控制指令的spi读操作（如下图9所示）时，应按照“执行软复位(向0x10寄存器写入0值)➡写入控制信号及数据信号(分别向0x18、0x20、0x28、0x30、0x38、0x40、0x48、0x50寄存器写入相应的控制信号及待发送的指令值)➡撤销软复位(向0x10寄存器写入1值)➡读取状态位(循环读取0x68寄存器的值，直至返回值为1)”的次序操作寄存器；上述两种情况在PYNQ中的示例驱动如下：

|  |
| --- |
| def spi\_read(chip, rd\_command):      if chip in chip\_settings:          chip\_num = chip\_settings[chip]["chip\_num"]          cpol = chip\_settings[chip]["cpol"]          cpha = chip\_settings[chip]["cpha"]          rd\_mode = chip\_settings[chip]["rd\_mode"]          rd\_comd\_width = chip\_settings[chip]["rd\_comd\_width"]          rd\_data\_width = chip\_settings[chip]["rd\_data\_width"]      if rd\_mode==0 :          write\_reg(SPI\_CONTROL, soft\_rst\_n\_reg, 0x0)          write\_reg(SPI\_CONTROL, chip\_reg, chip\_num)          write\_reg(SPI\_CONTROL, cpol\_reg, cpol)          write\_reg(SPI\_CONTROL, cpha\_reg, cpha)          write\_reg(SPI\_CONTROL, w\_r\_mode\_reg, 0x0)          write\_reg(SPI\_CONTROL, rd\_width\_reg, rd\_data\_width)          write\_reg(SPI\_CONTROL, rd\_target\_num\_reg, 0x1)          write\_reg(SPI\_CONTROL, soft\_rst\_n\_reg, 0x1)      else :          write\_reg(SPI\_CONTROL, soft\_rst\_n\_reg, 0x0)          write\_reg(SPI\_CONTROL, chip\_reg, chip\_num)          write\_reg(SPI\_CONTROL, cpol\_reg, cpol)          write\_reg(SPI\_CONTROL, cpha\_reg, cpha)          write\_reg(SPI\_CONTROL, w\_r\_mode\_reg, 0x2)          write\_reg(SPI\_CONTROL, wr\_width\_reg, rd\_comd\_width)          write\_reg(SPI\_CONTROL, wr\_data\_reg, rd\_command)          write\_reg(SPI\_CONTROL, rd\_width\_reg, rd\_data\_width)          write\_reg(SPI\_CONTROL, rd\_target\_num\_reg, 0x1)          write\_reg(SPI\_CONTROL, soft\_rst\_n\_reg, 0x1)      rd\_done = read\_reg(SPI\_CONTROL, rd\_done\_reg)      while rd\_done!=1 :          rd\_done = read\_reg(SPI\_CONTROL, rd\_done\_reg)      rx\_data = read\_reg(SPI\_CONTROL, rd\_data\_reg)      rx\_data\_bin = format(rx\_data, '032b')[-rd\_data\_width:]      rx\_data = int(rx\_data\_bin, 2)      print(f"Successfully read, the result is {rd\_data\_width}'b{rx\_data\_bin}.")      return rx\_data |

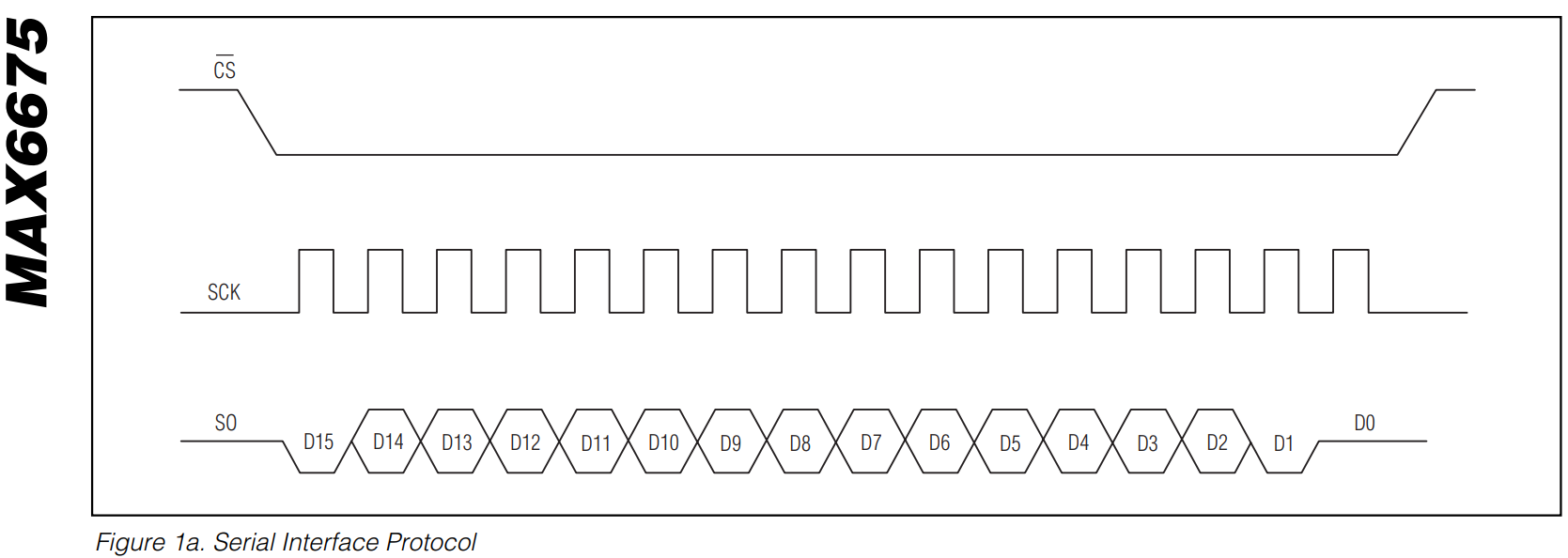


图8. MAX6675读时序

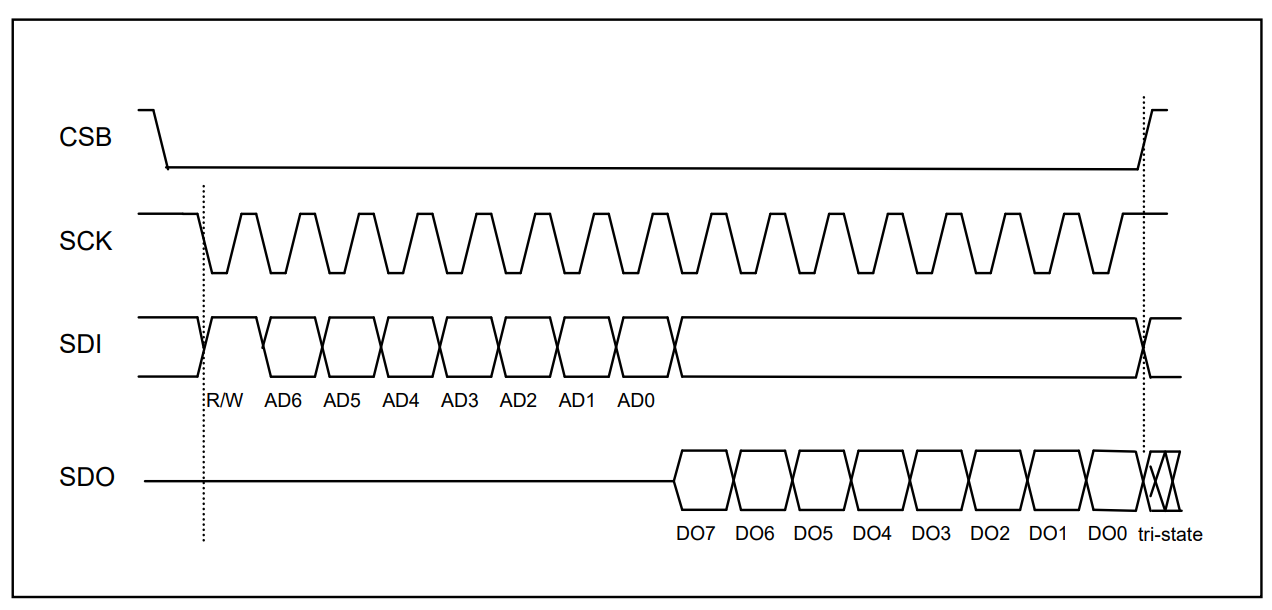


图9. BMI160读时序