第一章 基于 FPGA 的手势识别控制系统设计

1.1 章节简介

手势识别是一种基于人机交互的技术，可以通过分析人体姿态和动作来实现与计算机的交互操作。手势识别技术已被广泛应用于游戏、娱乐、智能家居、医疗保健、工业自动化等领域。

手势识别技术主要包括手势数据采集和手势结果指示两个基本步骤。其中，手势数据采集是指通过手势传感器 PAJ7620U2 采集操作者的动作信息；手势结果指示是指对采集到的手势数据进行处理和分析，并利用各类外设对结果作出指示。

随着智能物联网和智能家居等技术的普及，手势识别技术在日常生活中的应用前景也越来越广泛。

本章笔者将带领各位读者通过 I2C 协议去驱动手势传感器 PAJ7620U2，实现对操作者上、下、左、右挥手动作的采集，并利用四个 LED 灯外设对挥手动作作出指示。

1.2 PAJ7620U2

1.2.1 简介

PAJ7620U2 将手势识别功能与通用 I2C 接口集成在一颗芯片中。它能够识别包括向上、向下、向左、向右、向前、向后、顺时针、逆时针旋转和挥手等 9 种手势，这些手势信息可以通过 I2C 总线被轻松读取到。PAJ7620U2 还提供内置的接近检测功能，用于感知物体的靠近或离开。该芯片设计为在-40°C 至+85°C 的温度范围内，以 2.8V 至 3.3V 的电压工作，并且 I2C 总线和中断线的上拉电压范围为 1.8V 至 3.3V。

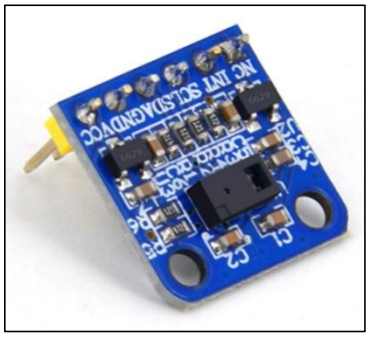

图 1.1　PAJ7620U2 实物图

1.2.2 引脚配置

表 1.1 PAJ7620U2 引脚配置

Pin NO.	符号	类型	功能
1	Vbus	POWER	为总线供电
2	SDA	IN/OUT（开漏）	I2C 数据引脚
3	INT	OUT（开漏）	中断引脚（低电平有效）
4	TESTMD	IN	仅作为模块测试
5	SCL	IN（开漏）	I2C 时钟引脚
6	GND	GND	接地
7	GPIO3	IN/OUT	仅作为模块测试
8	GPIO2	IN/OUT	仅作为模块测试
9	GPIO1	IN/OUT	仅作为模块测试
10	GND	GND	接地
11	Vled	POWER	LED 供电输入
12	Vdd	POWER	主电源供电
13	GPIO0	IN/OUT	仅作为模块测试

如表 1.1 所示，手势识别传感器总共有 13 个引脚，不过多数引脚是同类型或者是没有实际意义的，因此在硬件设计时可将同类型的引脚（如 GND）合并，将无意义的引脚断连（如 GPIO0~3）。

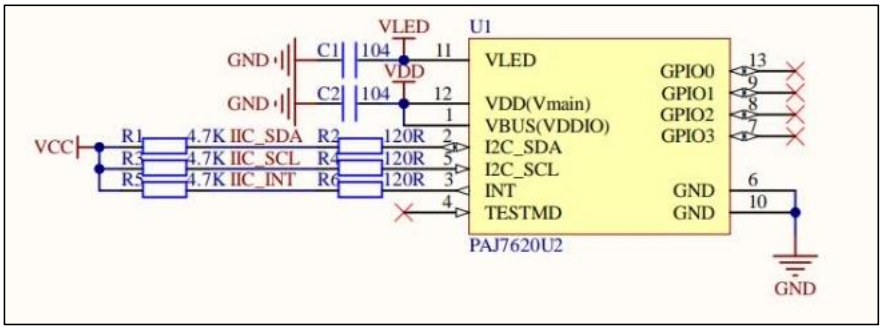

图 1.2　PAJ7620U2 硬件原理图

需要注意的是，SCL、SDA 和 INT 引脚都是使用开漏输出方式，即在发送低电平时，总线上的器件会将相应的信号线拉低；但是在低电平状态下，总线上的器件不会主动驱动信号线为高电平，而是通过上拉电阻使信号线回到高电平。因此，如图 1.2 所示，在硬件设计时，需要外接上拉电阻，保证信号在不被使用的空闲状态时处于高电平状态，并提高信号线的高电平恢复能力。

同时，为了使传感器同时兼容 3.3V 和 5V 的工作电压，在硬件设计时可以加入稳压芯片，将输入电压稳定地调节为恒定的输出电压 3.3V，并具有过流保

护、过热保护和短路保护等功能，保护连接的电路免受电压波动或故障引起的损害。

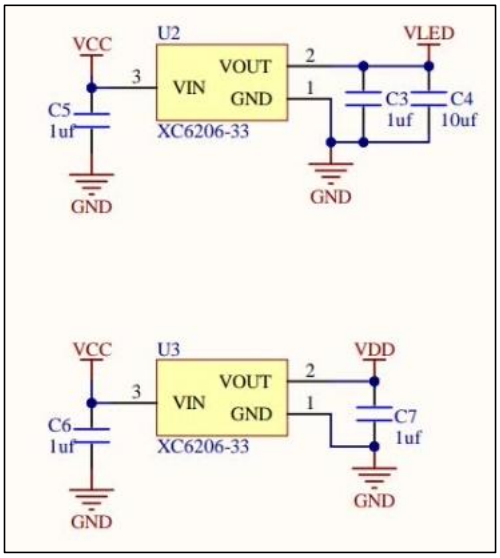

图 1.3 稳压电路原理图

综上所述，PAJ7620U2 传感器总共保留了五个引脚：VCC、GND、SCL、SDA、INT，在获取手势数据时，对 SDA 总线上的数据进行采集并拼接，即可得到完整的手势数据，因此可以不使用 INT 引脚。在后期上板验证时，只需将上述除 INT 引脚外的其余四个引脚，与 FPGA 开发板连接即可。

1.2.3 寄存器映射和功能

Register Bank 0 (Switch to Register Bank 0 by setting Addr 0xEF to 00)

Address	Register Function	Access	Default
0x03	I^2C suspend command (Write 0x01 to enter suspend state). I^2C wake-up command is slave ID wake-up. Refer to topic "I^2C Bus Timing Characteristics and Protocol"	W	0x01
0x41	Gesture detection interrupt flag mask	R/W	0xFF
0x42	Gesture/PS detection interrupt flag mask	R/W	0xFF
0x43	Gesture detection interrupt flag	R	-
0x44	Gesture/PS detection interrupt flag	R	-
0x45	State indicator for gesture detection (Only functional at gesture detection mode)	R	-
0x69	PS hysteresis high threshold (Only functional at proximity detection mode)	R/W	0xC8
0x6A	PS hysteresis low threshold (Only functional at proximity detection mode)	R/W	0x40
0x6B	PS approach state, Approach = 1 , (8 bits PS data >= PS high threshold) Not Approach = 0 , (8 bits PS data <= PS low threshold) (Only functional at proximity detection mode)	R	-
0x6C	PS 8 bit data (Only functional at proximity detection mode)	R	-
0xB0	Object Brightness (Max. 255)	R	
0xB1 0xB2	Object Size (Max. 900)	R	

图 1.4 Bank0 寄存器映射和功能

Register Bank 1 (Switch to Register Bank 1 by setting Addr 0xEF to 01)

Address	Register Function	Access	Default
0x44	PS gain setting (Only functional at proximity detection mode)	R/W	0xA0
0x67	IDLE S1 Step, for setting the $S_{1,\ Response\ Factor}$	R/W	0x68
0x68			0x01
0x69	IDLE S2 Step, for setting the $S_{2,\ Response\ Factor}$	R/W	0xD0
0x6A			0x02
0x6B	OPtoS1 Step, for setting the OPtoS1 time of operation state to standby 1 state	R/W	0xB0
0x6C			0x04
0x6D	S1toS2 Step, for setting the S1toS2 time of standby 1 state to standby 2 state	R/W	0x60
0x6E			0x09
0x72	Enable/Disable PAJ7620U2	R/W	0x00

图 1.5　Bank1 寄存器映射和功能

如图 1.4 和图 1.5 展示 Bank0 和 Bank1 内列举的部分寄存器及其功能说明，对各种功能和参数，包括地址、函数名、访问权限（只可写、只可读或可读可写）和默认值进行了说明。需要注意的是，如果操作者要访问某个寄存器，必须首先激活对应的 Bank 区域。例如，要访问地址为 0x43 的寄存器，必须先将 Bank0 激活。

在官方手册里面，也只是简单列举了少部分寄存器，并进行了简单的阐述，完成手势识别具体需要使用到哪些寄存器，还需要继续阅读手册并完成配置。

Register Bank0, ADDR 0x43

	Register Bank 0, ADDR 0x43, Gesture Detection Interrupt Flag							
NAME	Counter Clockwise	Clockwise	Backward	Forward	Right	Left	Down	Up
BIT #	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
ACCESS	R	R	R	R	R	R	R	R
DEFAULT	-	-	-	-	-	-	-	-
	-							

NAME	FUNCTION/OPERATION
Counter Clockwise	1: Counter clockwise gesture be detected 0: No Counter clockwise gesture be detected
Clockwise	1: Clockwise gesture be detected 0: No Clockwise gesture be detected
Backward	1: Backward gesture be detected 0: No Backward gesture be detected
Forward	1: Forward gesture be detected 0: No Forward gesture be detected
Right	1: Right gesture be detected 0: No Right gesture be detected
Left	1: Left gesture be detected 0: No Left gesture be detected
Down	1: Down gesture be detected 0: No Down gesture be detected
Up	1: Up gesture be detected 0: No Up gesture be detected

图 1.6　0x43 寄存器功能图

如图 1.6 所示为 BANK0 内地址为 0x43 的寄存器，功能为手势检测，初始如果没有检测到手势动作，那么 8 位都为低电平，且当前的手势识别结果不受前面

手势识别结果的影响。当检测到某一位为高电平时，即代表检测到对应位的手势动作，如 bit[3]位由“0”变为“1”，表示此时传感器检测到向右的挥手动作。

1.2.4 操作说明

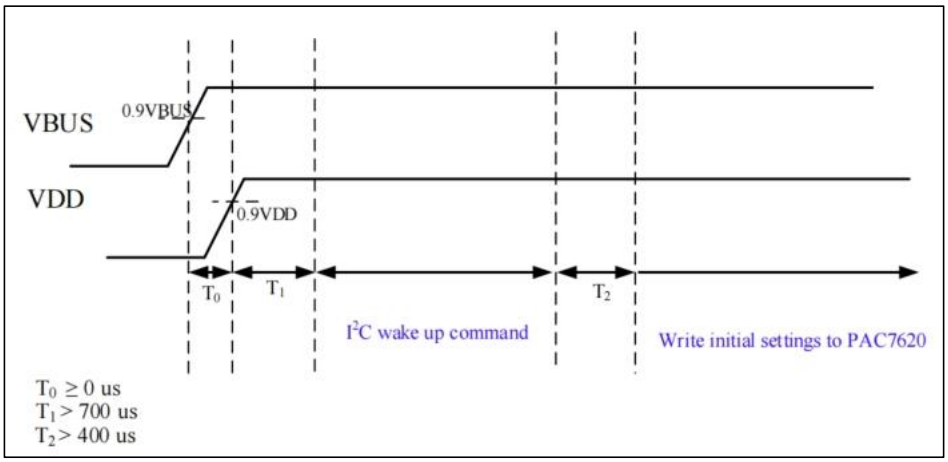

图 1.7　PAJ7620U2 上电时序图

图 1.7 描述了设备的启动顺序和设置过程。根据时序图可知，首先需要在 VBUS 上通电，然后在 VDD 上通电。VBUS 表示总线工作的电压，VDD 表示器件内部的工作电压，在硬件设计时，需要考虑通电的先后顺序。但因用户使用的是封装好的器件，所以在本设计中，不需要考虑通电顺序。

通电后，等待 T1 微秒使 PAJ7620U2 稳定，然后通过 I2C 总线将唤醒指令写入处理 I2C 唤醒。经过 T2 微秒后，将初始设置和不同模式的设置写入 PAJ7620U2。

```
Step 1：Power On, The VBus must be power on before VDD.
Step 2：Wait 700us for PAJ7620U2 to stabilize.
Step 3：Write slave ID or I2C read command to process I²C wake-up.
        It's recommend to read Reg_0x00. It will return "0x20" when wake-up finish.
        By the way, There is no-ack from PAJ7620U2 before wake-up finish.
Step 4：Write initial setting to gesture.
          unsigned char initial_register_array[][2] = {
```

图 1.8　PAJ7620U2 操作步骤

如图 1.8 所示，PAJ7620U2 操作步骤为：

Step1：上电，Vbus 必须在 Vdd 之前上电；

Step2：器件在上电后会进行一系列自检和初始化工作，此时如果向器件发送配置指令，指令可能不会被器件接收到，从而导致配置失败，因此上电后等待

700us 让 PAJ7620U2 稳定。在本设计中，笔者延长等待时间至 1000us，因为 1000us 与 700us 相比，两者相差不大，因此可以适当延长时间确保器件初始化完成。

Step3：写入从设备 ID 或者 I2C 读指令去唤醒 PAJ7620U2，在本设计中使用如图 1.9 所示唤醒指令去激活传感器。

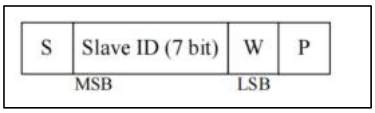

图 1.9 唤醒指令

唤醒指令发送完成后，需要读取 0x00 寄存器内的数据，如果得到的值为 0x20，则代表唤醒操作成功，反之则唤醒失败，需要重新发送唤醒指令唤醒传感器。

需要注意的是，根据 1.2.3 节的说明，PAJ7620U2 内部的寄存器被划分为两个 Bank 区域。在对 0x00 寄存器进行操作之前，需要激活相应的 Bank。然而，官方手册没有具体说明 0x00 寄存器属于哪个 Bank 区域。为了解决这个问题，可以先激活 Bank0 区域，然后读取 0x00 寄存器内的数据。如果读取到的值不是 0x20，则说明 0x00 寄存器不属于 Bank0 区域。在这种情况下，需要更改发送的指令，激活 Bank1 并重新读取数据。

经过笔者的测试，0x00 寄存器是隶属于 Bank0 区域，因此在设计时，需要先激活 Bank0，然后读取其内部 0x00 寄存器内的数据，读取到的值如果为 0x20，则代表唤醒操作成功，才可以进行下一步配置。

step4：向寄存器组内写入初始化值，为获取手势数据作准备。第一次接触 I2C 器件的读者可能会有疑问：为什么不直接读取手势数据寄存器内的数据来获取操作者挥手动作呢？

因为读取手势数据是最后一步，必须在配置寄存器组完成之后进行。可以举个例子来说明，假设读者想要做饭，那么首先必须购买一系列食材。只有准备好了这些食材，才能开始做饭。不可能在没有购买食材的情况下就开始做饭，或者是先将饭做好了再去购买食材，这是不合理的。

同理，初始化完成寄存器组是为最后的手势识别这一功能服务的，如果寄存器组没有初始化成功，那么传感器就无法实现手势识别，因此这一步是必须要执行的。

step5：获取手势数据。

ii. Get Gesture result

Step 1：Set Interrupt or I²C polling timer.

Step 2：Read Bank_0_Reg_0x43/0x44 for gesture result if interrupt or timer happen.

Gesture result will be clean when I²C read finish.

图 1.10 获取手势数据操作步骤

如图 1.10 所示，获取手势结果流程为：设置中断或者轮询计时器来访问手势数据寄存器内的数据，寄存器地址为 0x43 或者为 0x44，当读操作完成以后，需要清除手势结果以便下次获取。

使用 FPGA 获取手势数据时，设计中不使用中断引脚，可以使用轮询的方式来重复读取手势数据。由于每次只能一位一位地读取数据，所以采用拼接操作来获取完整的手势数据是最合适的。当次采集到的手势数据在读取完成后会自动清除，下一次读数据时会自动读出新的手势数据。

1.2.5 时序分析

i. **I²C Timing Parameter**

Parameter	Symbol	STANDARD MODE		FAST MODE		Unit
		Min.	Max.	Min.	Max.	
SCL clock frequency.	f_{scl}	10	100	10	400	kHz
Hold time for Start/Repeat Start. After this period, the first clock pulse is generated.	$t_{HD.STA}$	4		0.6		μs
Set-up time for a repeated Start.	$t_{SU.STA}$	4.7		0.6		μs
Low period of SCL clock.	t_{LOW}	4.7		1.3		μs
High period of SCL clock.	t_{HIGH}	4		0.6		μs
Data hold time.	$t_{HD.DAT}$	0		0		μs
Data set-up time.	$t_{SU.DAT}$	250		100		ns
Rise time of both SDA and SCL signals.	t_r		1000	-	300	ns
Fall time of both SDA and SCL signals.	t_f		300	-	300	ns
Set-up time for STOP condition.	$t_{SU.STO}$	4		0.6		μs
Bus free time between a STOP and START.	t_{BUF}	4.7		1.3		μs

** maximum current is 5mA and capacitance load spec. =100pF*

图 1.11 时序参数图

由图 1.11 可知，SCL 时钟频率如果位于 10KHZ 到 100KHZ 之间，可以使用标准模式（STANDARD MODE）和快速模式（FAST MODE），一般是使用的标准模式；SCL 时钟频率如果位于 100KHZ 到 400KHZ 之间，使用的就是快速模式。在本设计中，为了提升通信速率，采用的是快速模式进行通信。

采用快速模式通信，那么 SCL 低电平必须要大于 1.3us，高电平必须要大于 0.6us，如果采用上述 1.3us 和 0.6us 这两个极限值，设计出来的 SCL 时钟信号如图 1.12 所示：

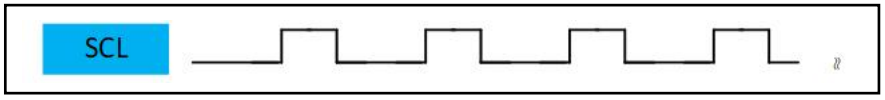

图 1.12 SCL 信号图示（1）

如图可知，SCL 信号占空比不为 50%，而且高低电平的持续时间存在较大差异。考虑到开发板的时钟频率为 50MHZ，使用计数器设计来实现 SCL 高电平和低电平变化会比较复杂。因此，一种方便的设计方法是适当延长 SCL 的低电平和高电平持续时间，以确保满足快速模式的要求，并生成占空比为 50%的 SCL 时钟，这种设计方法也更加方便实施。

将 SCL 低电平持续时间设计为 2us，高电平持续时间也为 2us，这样一个 SCL 周期就是 4us，时钟频率为 250KHZ，满足快速模式的要求。

综上设计出来的 SCL 时钟信号如图 1.13 所示：

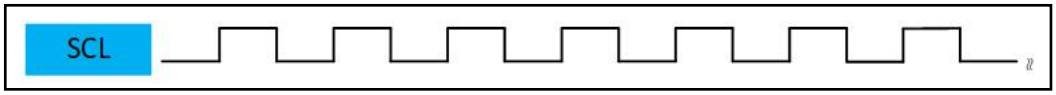

图 1.13　SCL 信号图示（2）

SCL 时钟信号设计完毕，此时需要考虑在 SCL 时钟信号上升沿到来之前，数据线上的信号需要保持稳定不变的时间，即建立时间（Set-up time），这是为了采集数据时能够在时钟上升沿捕获到正确的数据；还需要考虑在 SCL 时钟上升沿之后，数据线上的信号需要保持稳定不变的时间，即保持时间（Hold time），这是为了确保接收端有足够的时间采集数据。

因为 I2C 通信只有两根通信总线：SCL 和 SDA，SCL 在发送和接收数据时一般是周期性的变化，只有 SDA 在每个 SCL 周期可能会更新数据，那么为了设计方便，可以引入一个 i2c 驱动时钟 i2c_clk，将 SCL 周期四等分，设定 i2c_clk 时钟频率为 1MHZ，即一个时钟周期为 1us。SDA 线上的数据更新与采集如图 1.14 所示：

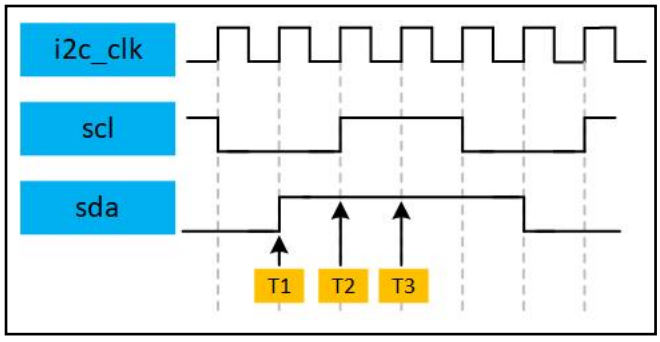

图 1.14　SDA 数据更新与采集

T2 时刻为 SCL 的上升沿，SDA 信号在 SCL 低电平中点，即 T1 时刻更新数据，那么 T1 与 T2 时刻之间的间隔为一个 i2c_clk 时钟周期，即 1us，满足图 1.11 中“Data set-up time(数据建立时间)”要求；在 T3 时刻采集更新后的数据，T2

与 T3 时刻之间的间隔也为一个 i2c_clk 时钟周期，亦满足图 1.11 中“Data hold time(数据保持时间)”要求。

综上可得：

表 1.2 时序参数表

Parameter	说明	使用
SCL clock frequency.	SCL 时钟频率	250KHZ
Hold time for Start/Repeat Start. ...	开始/重复开始信号保持时间	设计中设定≥1us
Set-up time for a repeated Start.	重复开始信号的建立时间	设计中设定≥1us
Low period of SCL clock.	SCL 低电平时间	2us
High period of SCL clock.	SCL 高电平时间	2us
Data hold time.	数据保持时间	1us
Data set-up time.	数据建立时间	1us
Rise time of both SDA and SCL signals.	SCL 和 SDA 信号上升时间	/
Fall time of both SDA and SCL signals.	SCL 和 SDA 信号下降时间	/
Set-up time for STOP condition.	结束状态建立时间	设计中设定≥1us
Bus free time between a STOP and START.	开始和结束状态的总线空闲时间	设计中设定≥2us

1.3 I2C 协议

1.3.1 简介

I2C（Inter-Integrated Circuit）是一种串行、半双工的通信协议，用于在集成电路（IC）之间进行数据传输。它由飞利浦半导体（现在的恩智浦半导体）于 1982 年开发，并已成为广泛应用于各种电子设备和系统中的标准通信协议。

I2C 协议提供了一种简单、可靠的方式来连接和通信不同的集成电路，在传感器、存储器、显示屏等器件和其他外设之间的通信中被广泛使用。

1.3.2 特点

▼ 总线结构：I2C 协议使用两根线进行通信，分别为数据线（SDA）和时钟线（SCL）。

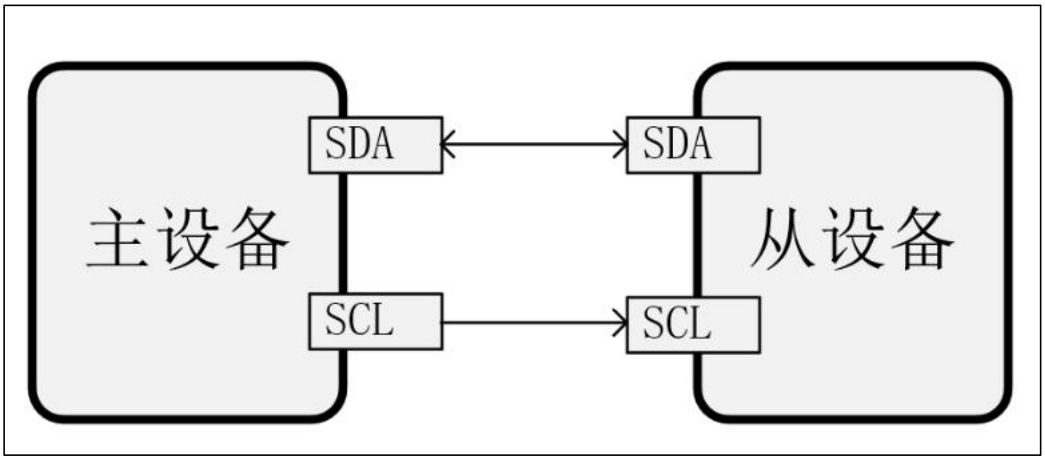

图 1.15　I2C 通信图示

从设备的工作时钟是由外部的主设备产生提供的，从设备和主设备之间会通过 SDA 数据总线进行数据交互，因此，对于 FPGA 而言，SCL 是输出信号，SDA 既是输入信号，又是输出信号。

▼ 主从结构：在 I2C 总线中，有一个主设备和一个或多个从设备。

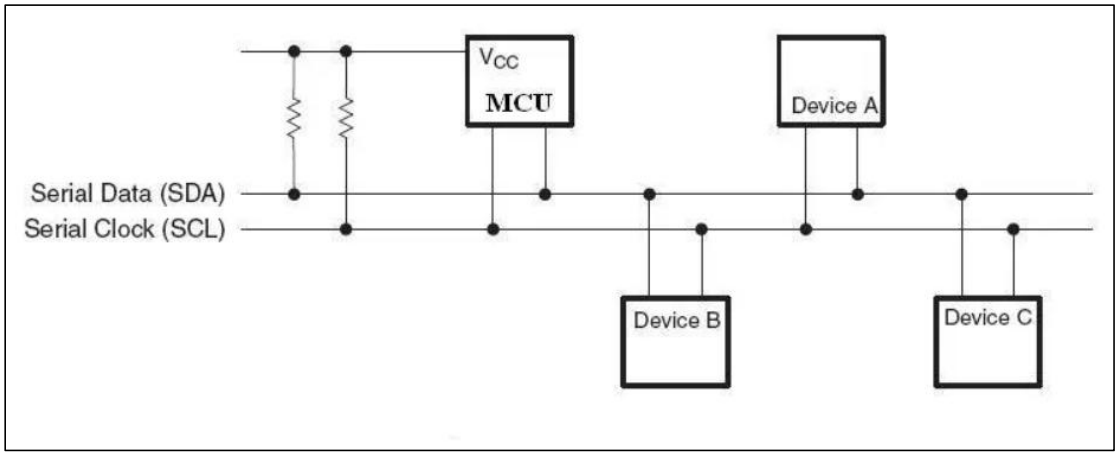

图 1.16　I2C 总线连接图

▼ 传输方式：I2C 协议采用同步传输方式，即写入和读出数据时钟是同步的，都是 SCL。

▼ 帧结构：I2C 通信以帧的形式进行。每个帧包含一个起始位、若干数据位、若干应答位和一个停止位。每个 SCL 周期传输一个数据位，高位在前，低位在后。

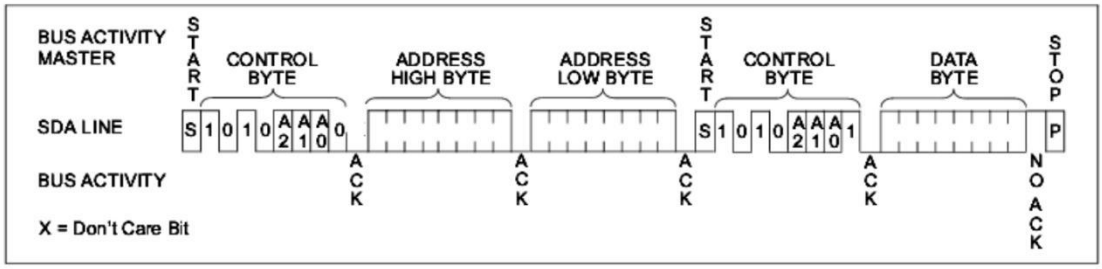

图 1.17　I2C 帧结构

▼ 地址分配：每个 I2C 设备都有一个唯一的 7 位地址。主设备在发起通信时会发送目标设备的地址，从而确定要与哪个从设备进行通信。

▼ 应答机制：在每个字节传输完成后，接收方会发送一个应答位来指示是否成功接收了数据。

▼ 多主设备：I2C 协议支持多个主设备同时存在于同一总线上。多主模式下，主设备必须通过仲裁过程来竞争总线的控制权。

1.3.3 写操作

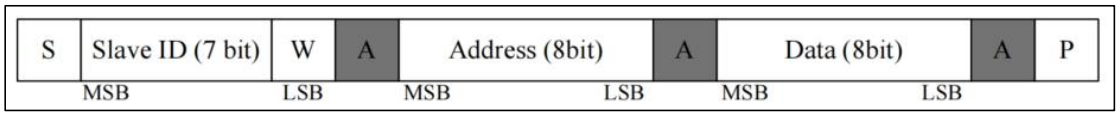

图 1.18 写操作指令格式

▼ 开始信号：开始信号由主设备发送，用于指示通信的开始。当 SCL 为高电平时，主设备将 SDA 线拉低，形成一个下降沿来表示开始信号的发出，这个下降沿是开始信号的标志。

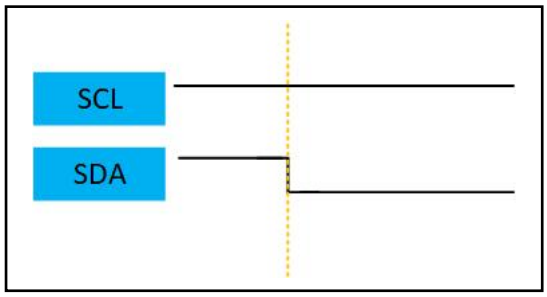

图 1.19　I2C 开始信号图示

▼ 设备地址+写位：主设备发送 7 位目标设备地址，同时在结尾添加“0”指示写操作。“X”为不关心。

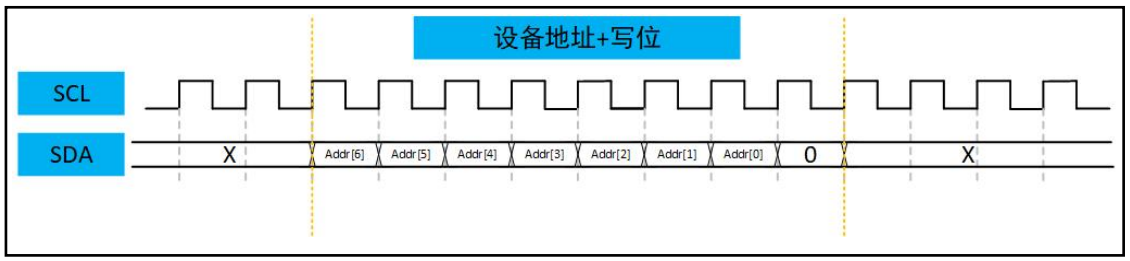

图 1.20 设备地址+写位图示

▼ 数据传输与应答：主设备将要写入的数据字节依次发送给从设备并接收和判断从设备应答信号。“X”为不关心，1byte 数据是由主设备产生，并通过 SDA 总线一位一位地发送给从设备；响应位是从设备每接收到一次主设备发送的 1byte 数据，都会产生 1bit 响应并通过 SDA 总线传输给主设备。主设备接收到数据为“0”代表响应信号有效，反之则无效。

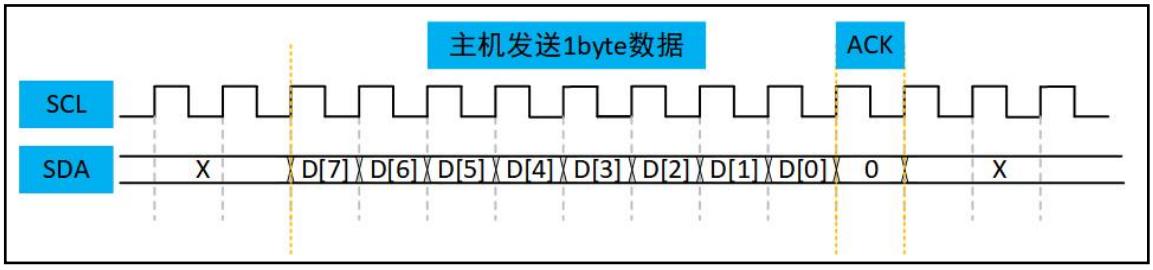

图 1.21 数据传输与应答图示

▼ 停止信号：停止信号信号也是由主设备发送，用于指示通信的结束。当 SCL 为高电平时，主设备将 SDA 线拉高，形成一个上升沿来表示停止信号的发出，这个上升沿是停止信号的标志。

图 1.22 I2C 结束信号图示

1.3.4 读操作

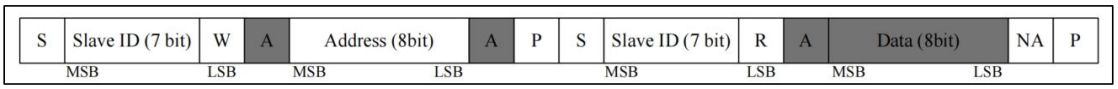

图 1.23 读操作指令格式

▼ 开始信号：由主设备发送，用于指示通信的开始。当 SCL 为高电平时，主设备将 SDA 线拉低，形成一个下降沿来表示开始信号的发出，这个下降沿是开始信号的标志。如图 1.19 所示。

▼ 设备地址+写位：主设备发送 7 位目标设备地址，同时在结尾添加“0”指示写操作。如图 1.20 所示。

▼ 寄存器地址传输与应答：主设备将要进行读数据的寄存器地址，依次发送给从设备并接收和判断从设备返回的应答信号。如图 1.21 所示。

▼ 设备地址+读位：主设备发送 7 位目标设备地址，同时在结尾添加“1”指示读操作。“X”为不关心。

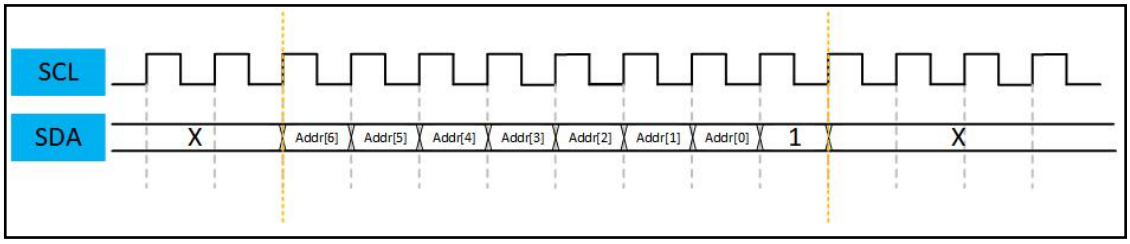

图 1.24 设备地址+读位图示

▼ 数据接收与发送应答：主设备读取从设备的数据，读取完成后向从设备发送应答信号。

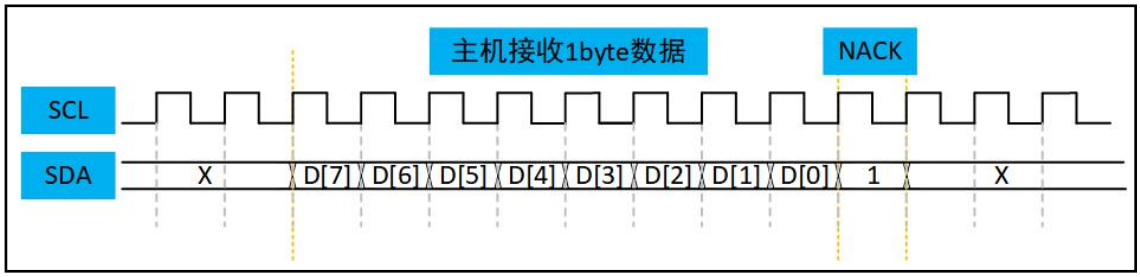

图 1.25 数据接收与应答图示

▼ 停止信号：停止信号信号也是由主设备发送，用于指示通信的结束。当 SCL 为高电平时，主设备将 SDA 线拉高，形成一个上升沿来表示停止信号的发出，这个上升沿是停止信号的标志。如图 1.22 所示。

1.4 实践操作

1.4.1 设计任务

识别操作者向上、下、左、右的挥手动作，并利用 LED 灯做出指示。

1.4.2 硬件设计

PAJ7620U2 引脚图如图 1.26 所示：

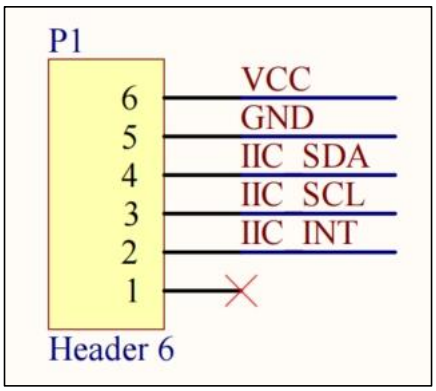

图 1.26　PAJ7620U2 引脚图示

开发板实物图如图 1.27 所示：

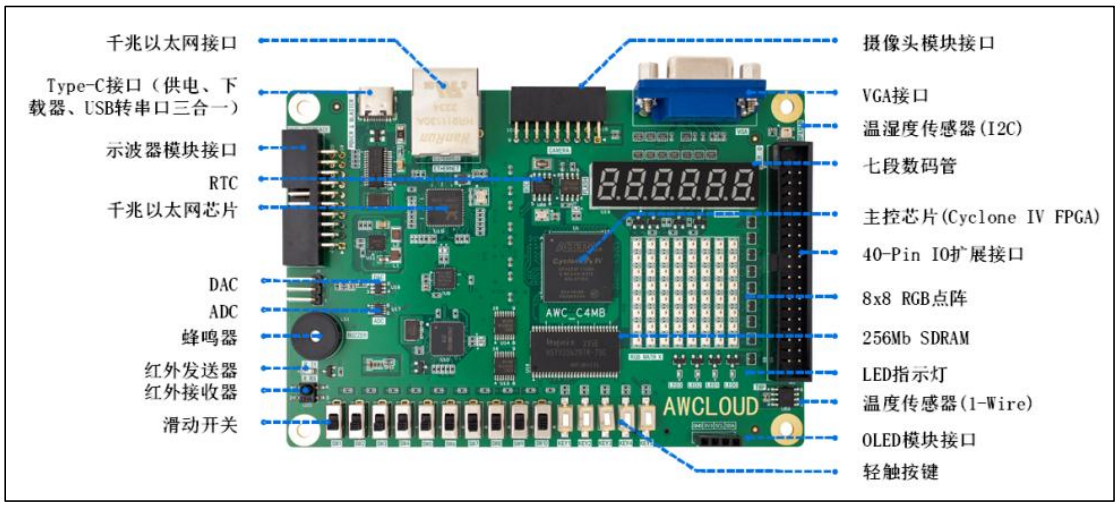

图 1.27 开发板实物图

将 PAJ7620U2 的 VCC、GND、SCL、SDA 引脚与 FPGA 开发板 40-Pin 扩展 IO 接口对应连接，INT 引脚不连接。其中 VCC 连接扩展 IO 口的 5V/3.3V 引脚，GND 连接扩展 IO 口的 GND 引脚，SCL 和 SDA 引脚可使用扩展 IO 口供电和接地外的任意两个引脚，笔者使用 N14、M12 两个引脚分别与 SCL 和 SDA 相连。硬件连接示意图如图 1.28 所示：

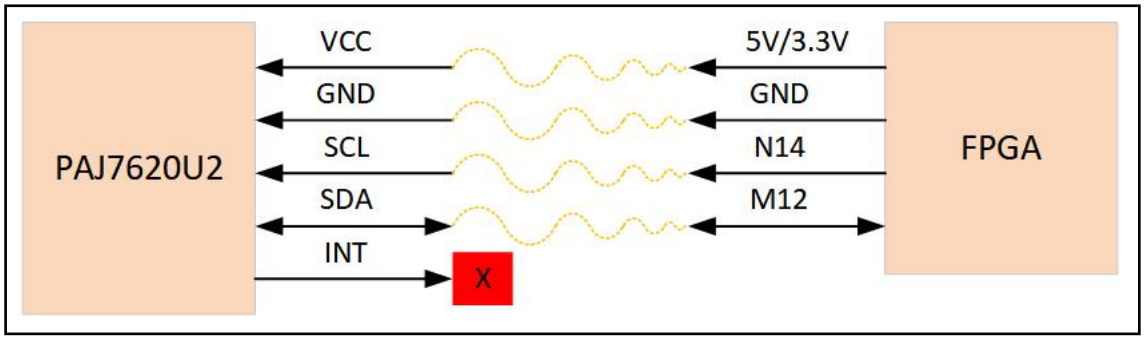

图 1.28 硬件连接示意图

1.4.3 程序设计

1. 手势识别顶层模块

1.1 模块框图

手势识别顶层模块框图如图 1.29 所示：

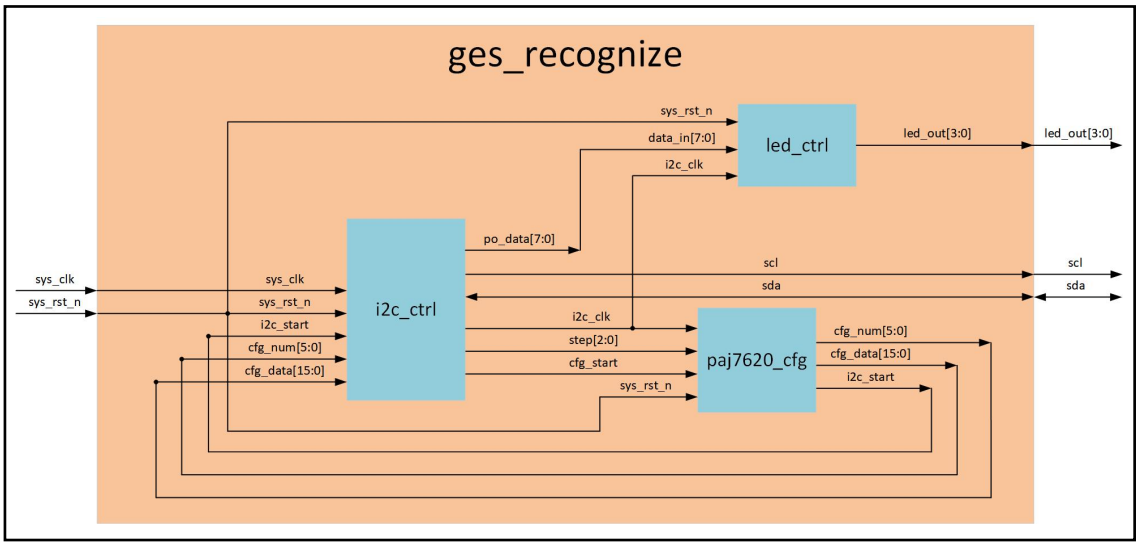

图 1.29 整体模块框图

本设计包含四个模块，各模块简介见表 1.3。

表 1.3 手势识别各模块简介

模块名称	功能说明
i2c_ctrl	I2C 控制模块
paj7620_cfg	PAJ7620U2 寄存器组配置模块
led_ctrl	LED 灯控制模块
ges_recognize	手势识别顶层模块

在本设计中，i2c_ctrl 模块主要完成对唤醒、写和读指令的执行；paj7620_cfg 模块功能为对待配置的寄存器地址和数据打包，并发送给 i2c_ctrl 模块进行配置；led_ctrl 模块功能是对采集到的手势数据，点亮不同的 LED 灯作为指示；ges_recognize 是本设计的顶层模块，主要级联上述涉及的模块。

1.2 信号波形图

手势识别顶层模块主要功能为关联定义的所有子模块，只需要涉及到信号定义和模块关联即可，不需要使用信号波形图作辅助。

代码清单

```
1.  module  ges_recognize
2.  (
3.      input   wire            sys_clk     ,
4.      input   wire            sys_rst_n   ,
5.
6.      output  wire            scl         ,
7.      output  wire    [3:0]   led_out     ,
8.
9.      inout   wire            sda
10. );
11.
12. wire            i2c_start   ;
13. wire    [5:0]   cfg_num     ;
14. wire    [15:0]  cfg_data    ;
15. wire            i2c_clk     ;
16. wire    [2:0]   step        ;
17. wire            cfg_start   ;
18. wire    [7:0]   ges_data    ;
19.
20. i2c_ctrl  i2c_ctrl_inst
21. (
22.     .sys_clk    (sys_clk    ),
23.     .sys_rst_n  (sys_rst_n  ),
24.     .i2c_start  (i2c_start  ),
25.     .cfg_num    (cfg_num    ),
26.     .cfg_data   (cfg_data   ),
27.     .scl        (scl        ),
28.     .i2c_clk    (i2c_clk    ),
29.     .step       (step       ),
30.     .cfg_start  (cfg_start  ),
```

```
    .po_data    (ges_data   ),
    .sda        (sda        )
);

paj7620_cfg  paj7620_cfg_inst
(
    .i2c_clk    (i2c_clk    ),
    .sys_rst_n  (sys_rst_n  ),
    .step       (step       ),
    .cfg_start  (cfg_start  ),
    .cfg_num    (cfg_num    ),
    .cfg_data   (cfg_data   ),
    .i2c_start  (i2c_start  )
);

led_ctrl  led_ctrl_inst
(
    .i2c_clk    (i2c_clk    ),
    .sys_rst_n  (sys_rst_n  ),
    .data_in    (ges_data   ),
    .led_out    (led_out    )
);

endmodule
```

2. I2C 控制模块

模块框图如图 1.30 所示：

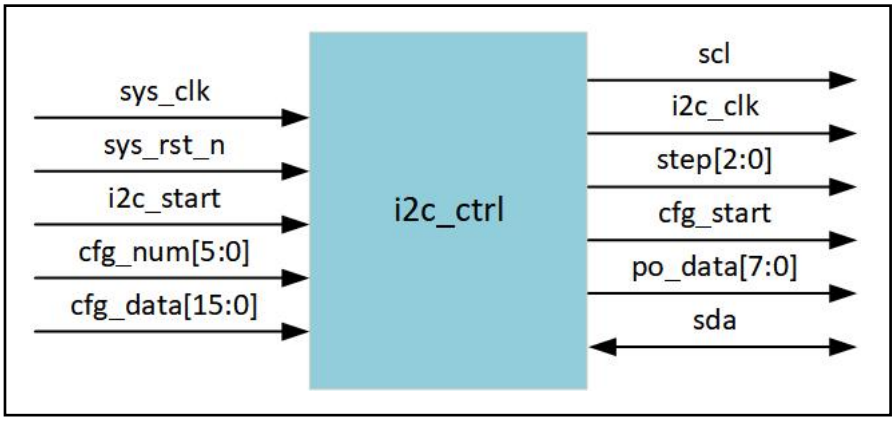

图 1.30　I2C 控制模块框图

根据模块框图，对模块端口信号说明如表 1.4 所示：

表 1.4 i2c_ctrl 模块端口信号说明

信号名称	信号端口类型	信号位宽	信号说明
sys_clk	input	1	系统时钟，频率为 50MHZ
sys_rst_n	input	1	系统复位，低电平有效
i2c_start	input	1	I2C 模块开始工作信号
cfg_num	input	6	寄存器配置模块打包完成的数据个数
cfg_data	input	16	寄存器配置模块打包的待配置的寄存器地址和向该地址内写入的数据
scl	output	1	串行时钟信号
i2c_clk	output	1	I2C 控制模块工作时钟，频率为 1MHZ
step	output	3	操作步骤信号，不同操作步骤对应不同配置状态跳转
cfg_start	output	1	寄存器配置模块开始工作信号
po_data	output	8	采集到的手势数据
sda	inout	1	串行数据信号

在前面的 1.2.4 节笔者有向各位读者介绍整体的操作步骤，接下来我们可以把之前的操作步骤整理细分一下，完成各操作步骤的状态转移图和信号波形图设计。

2.1 step0：唤醒操作

唤醒操作指令格式如图 1.31 所示：

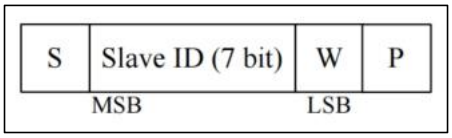

图 1.31 唤醒操作指令格式

根据唤醒操作指令格式，绘制的状态转移图如图 1.32 所示：

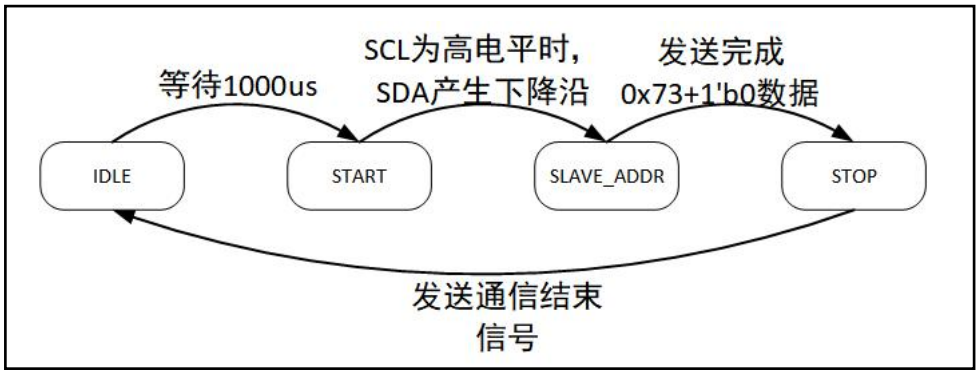

图 1.32 唤醒操作状态转移图

根据状态转移图可知，器件初始状态为 IDLE 状态，上电后等待 1000us 让器件内部初始化完成，趋于稳定状态，然后跳转到 START 状态；

在 START 状态下，SCL 为高电平时，SDA 产生下降沿，I2C 通信开始，跳转至 SLAVE_ADDR 状态；

在 SLVAE_ADDR 状态下，发送 7bit 从设备 ID+1bit 写位，即 0x73+0，发送完成后跳转到 STOP 状态；

在 STOP 状态下，SCL 为高电平时，SDA 产生上升沿，表示 I2C 一次通信结束，重新回到 IDLE 状态。

接下来需要产生一个 I2C 驱动时钟 i2c_clk，频率为 1MHZ，替代系统时钟 sys_clk 作为通信的主时钟，用来驱动后续 I2C 控制模块相关信号的产生，50MHZ 系统时钟分频产生 1MHZ I2C 驱动时钟信号波形图如图 1.33 所示：

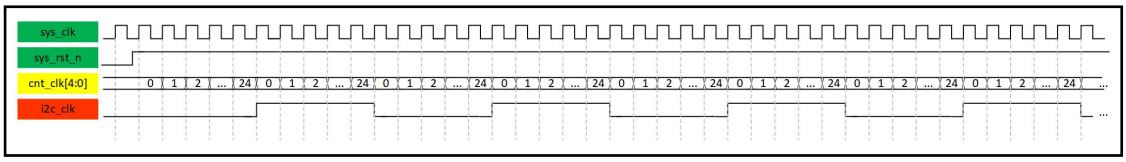

图 1.33 生成 I2C 驱动时钟信号波形图

根据图 1.32 唤醒操作状态转移图，绘制的唤醒操作信号波形图如图 1.34 所示：

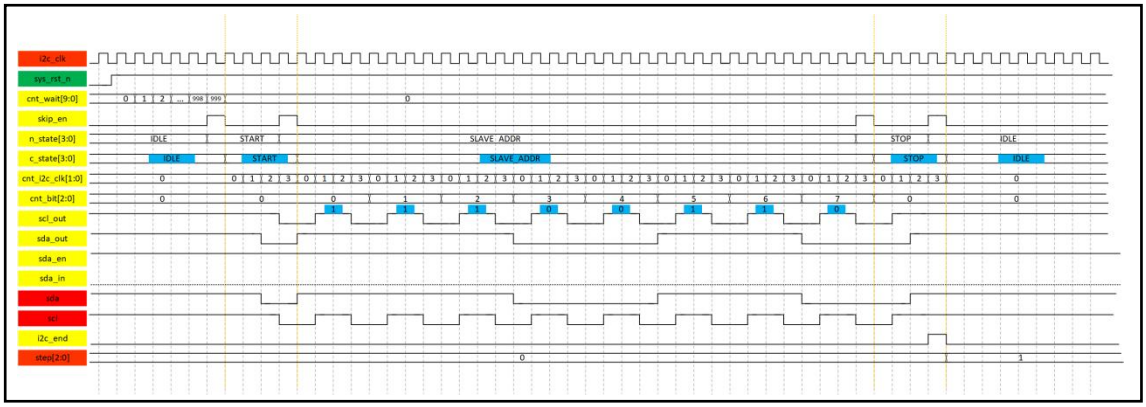

图 1.34 唤醒操作信号波形图

根据信号波形图，I2C 控制模块唤醒操作引入的内部信号说明如表 1.5 所示：

表 1.5 唤醒操作引入的内部信号说明表

信号名称	信号类型	信号位宽	信号说明
cnt_wait	reg	10	上电等待计数器
skip_en	reg	1	状态跳转信号
n_state	reg	4	三段式状态机次态
c_state	reg	4	三段式状态机现态
cnt_i2c_clk	reg	2	i2c_clk 时钟个数计数器
cnt_bit	reg	3	发送或者接收到的比特数据个数计数器
scl_out	reg	1	FPGA 产生输出至外部设备的串行时钟信号
sda_out	reg	1	FPGA 产生输出至外部设备的串行数据信号
sda_en	wire	1	SDA 总线通信使能信号
sda_in	wire	1	传感器输入至 FPGA 的数据信号
i2c_end	reg	1	I2C 一次通信结束标志

结合表 1.5，如下是对引入的内部信号说明：

▼ cnt_wait：上电等待计数器，因为驱动时钟为 i2c_clk，一个时钟周期为 1us，计数满 1000，即计数 1000us 时清零，表示上电等待时间结束，后续状态不使用该计数器，则一直保持“0”值不变；

▼ n_state：三段式状态机次态，比现态提前一个时钟周期；

▼ c_state：三段式状态机现态，对信号赋值都是在此状态描述，因此在图 1.34 中，笔者用橙色辅助线将现态与次态分开，并用淡蓝色填充现态，这样便于观测现态中变化的信号；

▼ cnt_i2c_clk：i2c_clk 时钟个数计数器，最多计数 4 个时钟周期，因为 4 个时钟周期刚好是 4us，可以构成低电平 2us、高电平 2us 的 SCL 信号，所以位宽设置为 2 个即可，在后面的设计中，主机接收、发送应答状态，开始和结束状态，都是在一个 SCL 周期内表示；

▼ cnt_bit：发送或者接收到的比特数据个数计数器，在 FPGA 需要向传感器发送配置指令，或者需要接收传感器返回的响应或数据时，因为发送或者接收的数据个数都是 1byte（8bit），所以需要该计数器，记录接收到的比特数据个数；

▼ skip_en：状态跳转信号，引入该信号是为了让状态机第二段，用代码描述状态跳转情况简便一些，该信号在现态每个状态的最后一个时钟周期拉高，当检测到该信号为高电平时，状态就会发生跳转，从而避免状态机第二段代码复杂和冗余；

▼ scl_out：主机产生输出至外部设备的串行时钟信号，该信号与端口信号 scl 无异，可以直接对 scl 赋值，重复引入 scl_out 目的是为了与 sda_out 保持一致；

▼ sda_en：SDA 总线通信使能信号，当 sda_en 为高电平时，表示主机向从机发送数据；当 sda_en 为低电平时，表示从机向主机发送数据。各位读者可以根据数据手册中，指令格式中的颜色来进行判断，如果是白色，数据传输方向为主机到从机，此时 FPGA 占用 SDA 总线；如果是灰色，数据传输方向为从机到主机，此时 FPGA 要放弃对 SDA 总线的控制（将对 SDA 总线的赋值置为高阻态），控制权交由从机。

例如，在图 1.31 唤醒操作指令格式中，颜色全是白色，表示此时 SDA 总线都是由主机占用；在图 1.35 写操作指令格式中，三个灰色的“A”即 ACK 响应状态都表示由从机向主机发送数据，此时从机占用 SDA 总线，其它白色的区域才是由主机占用；在图 1.40 读操作指令格式中，三个灰色的“A”和“DATA”状态都表示主机需要接收从机的数据，此时这些状态也都得是从机占用 SDA 总线，其它白色区域才代表是主机占用。

需要注意的是，当 FPGA 放弃对 SDA 总线控制时（赋高阻态），SDA 上不是没有数据，也不是为高阻态，此时是从机占用此总线，因此传输的是从机发送至主机的数据。

▼ sda_out：主机产生输出至外部设备的串行时钟信号，只有在 sda_en 为高电平，即 FPGA 占用 SDA 通信线时，才能将此信号赋值给 SDA；如果主机不占用 SDA 通信线，sda_out 为何值，都无法赋值给 SDA。

▼ sda_in：传感器输入至 FPGA 的数据信号，只有在从机发送数据给主机时，该信号的值才需要被考虑，其它时刻是主机占用 SDA 通信总线，sda_in 为何值完全可以不用关心；

▼ i2c_end：一次通信结束的标志，在 STOP 状态最后一个时钟周期拉高。

2.2 step1：激活 Bank0

激活 Bank0 采用写操作指令格式，如图 1.35 所示：

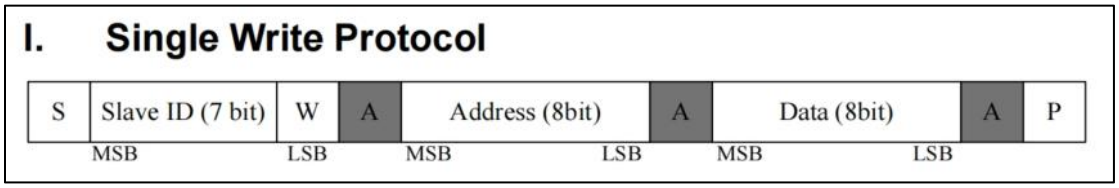

图 1.35 写操作指令格式

根据写操作指令格式，绘制的激活 Bank0 状态转移图如图 1.36 所示：

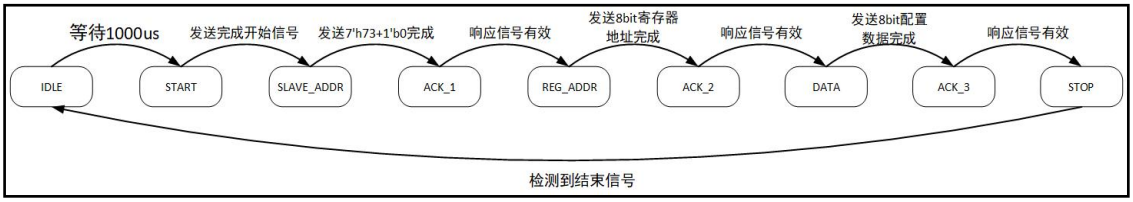

图 1.36 激活 Bank0 状态转移图

根据状态转移图可知，唤醒操作完成后，状态回到 IDLE 状态，由图 1.7 可知，唤醒指令发送完成后需要等待 T2（400us）时间，才能跳转到 START 状态，此处因为等待时间比较小，仍可以用之前等待 1000us 的时间，等待完成后跳转到 START 状态；

在 START 状态下，SCL 为高电平时，SDA 产生下降沿，即发送完成开始信号，状态跳转到 SLAVE_ADDR 状态；

在 SLAVE_ADDR 状态下，发送完成设备地址+写位，即 7'h73+1'b0，跳转到响应 ACK 状态；

在 REG_ADDR 状态下，发送完成 1byte 待配置的寄存器地址，完成后跳转到响应 ACK 状态；

在 DATA 状态下，发送完成 1byte 向寄存器地址里面配置的数据，完成后也是跳转到 ACK 状态；

在 ACK_1、ACK_2、ACK_3 状态下，此时涉及到从机向主机发送响应信号，要跳转到下一个状态，必须判断是否接收到有效的响应。因此在该状态下，需要在 cnt_i2c_clk == 2 时采集 sda_in，即从机返回的响应数据，如果此时 sda_in 为低电平，则响应有效，拉高跳转信号 skip_en，反之则不拉高。

读者可能会有疑问，为什么在 cnt_i2c_clk == 2 时采集到低电平数据就代表接收到有效响应了呢？因为响应一般是连续的，如果在 cnt_i2c_clk == 2 时采集到低电平，那么在前面的 cnt_i2c_clk == 0/1 时，都应该为相同的低电平状态。在后续使用 Signal Tap 抓取信号波形验证时，如果发现 cnt_i2c_clk == 3 时接收到的是高电平响应，此时不用担心，因为前面三个时钟周期从机给了主机正确的响应，在最后一个时钟周期从机放弃控制总线，这也是代表主机发送的指令有效。

在检测到最后一个 ACK 状态有效响应后，跳转到 STOP 状态，在该状态下，SCL 为高电平时，SDA 产生上升沿，重新回到 IDLE 状态。

根据图 1.36 激活 Bank0 状态转移图，绘制的信号波形图如下所示：

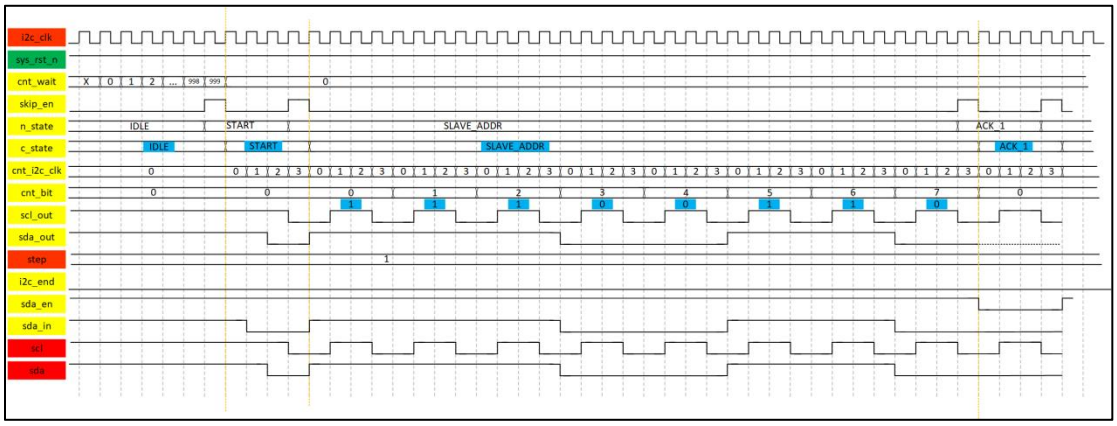

图 1.37 激活 Bank0 信号波形图（1）

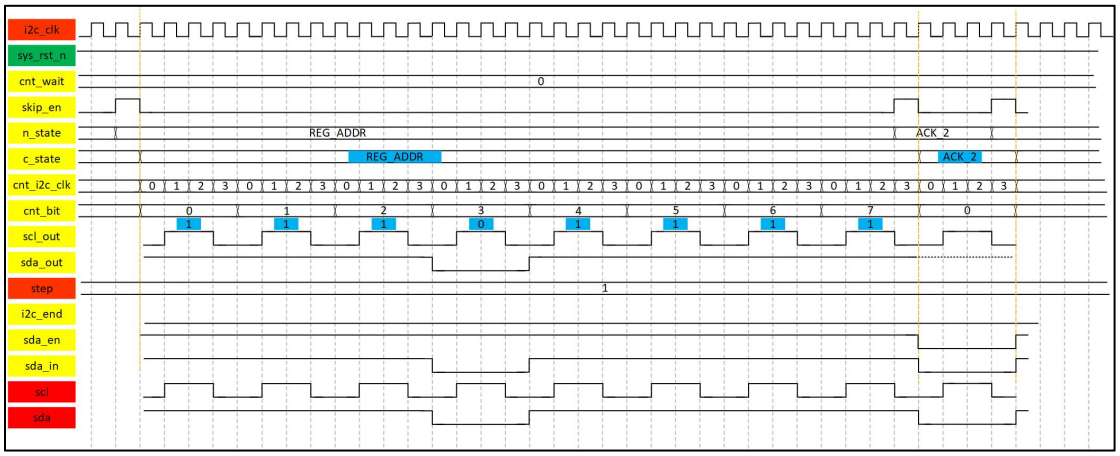

图 1.38 激活 Bank0 信号波形图（2）

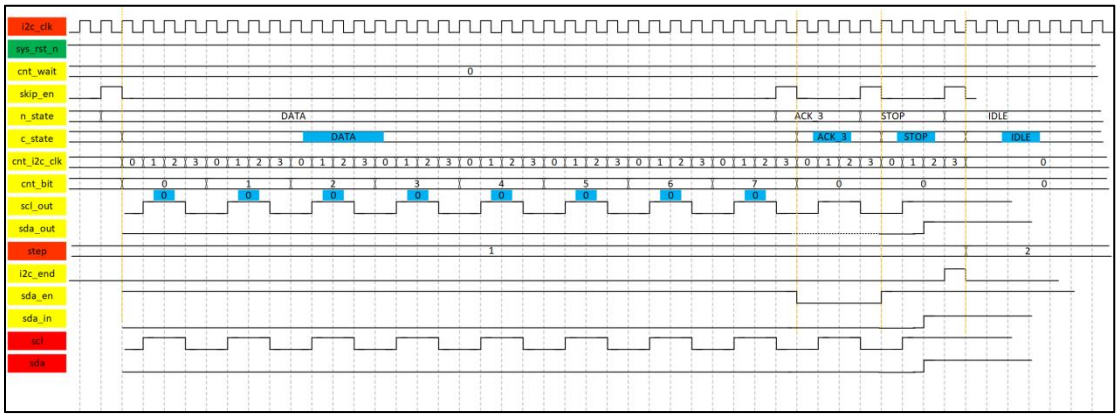

图 1.39 激活 Bank0 信号波形图（3）

因为激活 Bank0 操作与唤醒操作相比，未引入新的内部信号，此处不再对上述信号进行说明。

2.3 step2：读取 0x00 寄存器数据（上）

读数据采用的指令格式如图 1.40 所示：

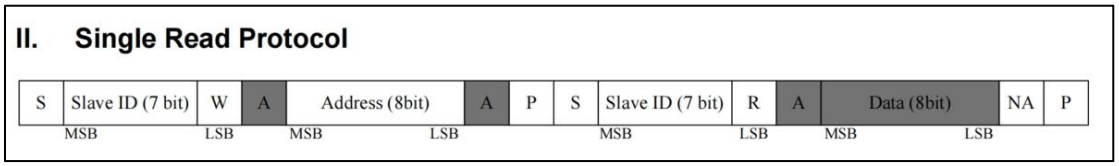

图 1.40 读数据指令格式

在设计时，可将上述读数据指令格式分成两个部分，第一部分为第一个“S”到第一个“P”，第二部分为第二个“S”到第二个“P”。step2 是实现前半部分，指定读数据的寄存器地址。根据读数据指令格式前半部分，绘制的状态转移图如图 1.41 所示：

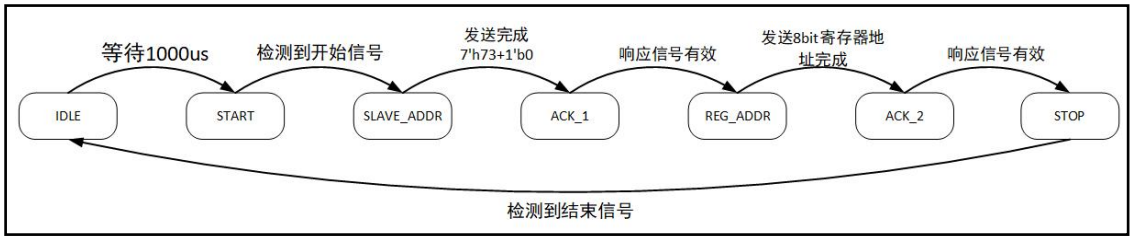

图 1.41 读取 0x00 寄存器数据（上）状态转移图

在前面设定的 IDLE 状态跳转到 START 状态跳转条件为等待 1000us 结束，因为这个时间很短暂，对功能实现没有任何影响，因此这里可以继续保留这个跳转条件。

其它状态之间的相互跳转情况及跳转条件，与 step1 是很类似的，此处就不再赘述。根据图 1.41 状态转移图绘制的信号波形图如下图所示：

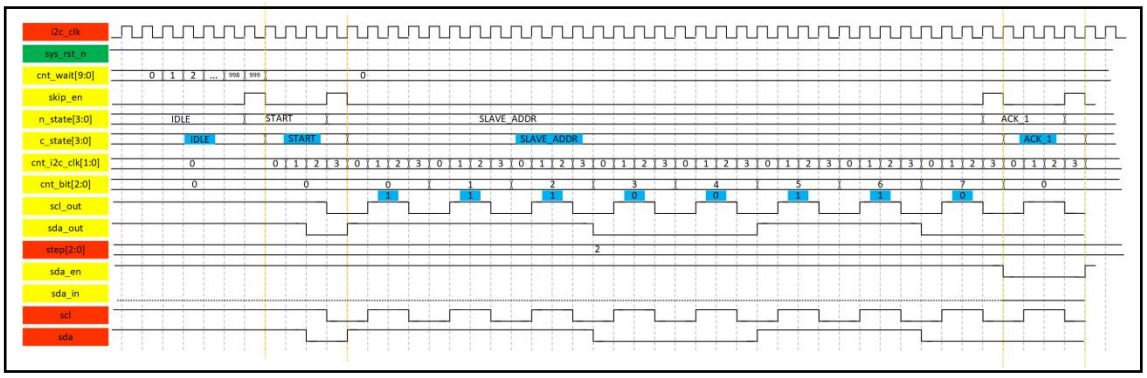

图 1.42 读取 0x00 寄存器数据（上）信号波形图（1）

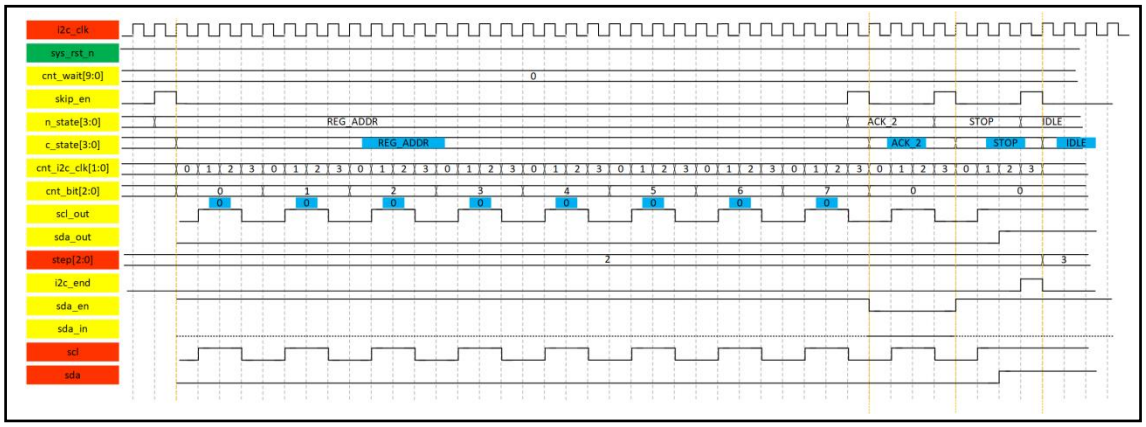

图 1.43 读取 0x00 寄存器数据（上）信号波形图（2）

本操作步骤未引入新的内部信号，对已有的信号功能不再赘述。

2.4 step3：读取 0x00 寄存器数据（下）

读数据采用的指令格式见图 1.40，根据指令格式后半部分绘制的状态转移图如图 1.44 所示：

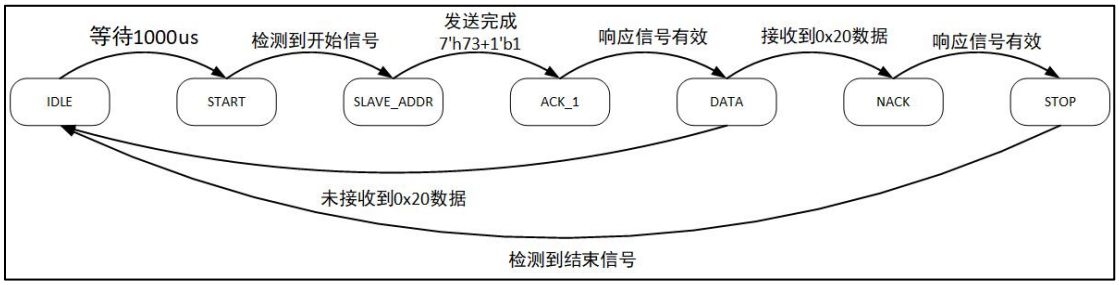

图 1.44 读数据状态转移图

由读数据状态转移图可知，前面三个状态跳转和前面几步操作都是一样的。需要注意的是，从 SLAVEA_ADDR 状态跳转到 ACK_1 状态，跳转条件为发送完成从设备地址+读位；DATA 状态此时就不是由主机控制，而是由从机控制了，并且当且仅当读取到的数据为“0x20”时，才能跳转到下一个 NACK 状态，由主机向从机发送为高电平的应答信号；如果在 DATA 状态未接收到“0x20”数据，则代表唤醒操作失败，需要回到 IDLE 状态并重新开始执行唤醒操作。

根据图 1.44 状态转移图绘制的信号波形图如下所示：

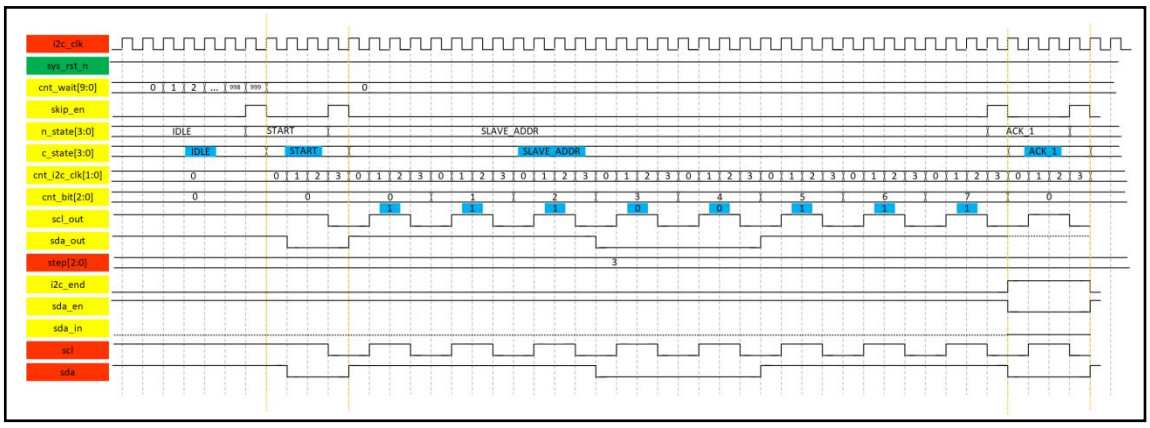

图 1.45 读数据信号波形图（1）

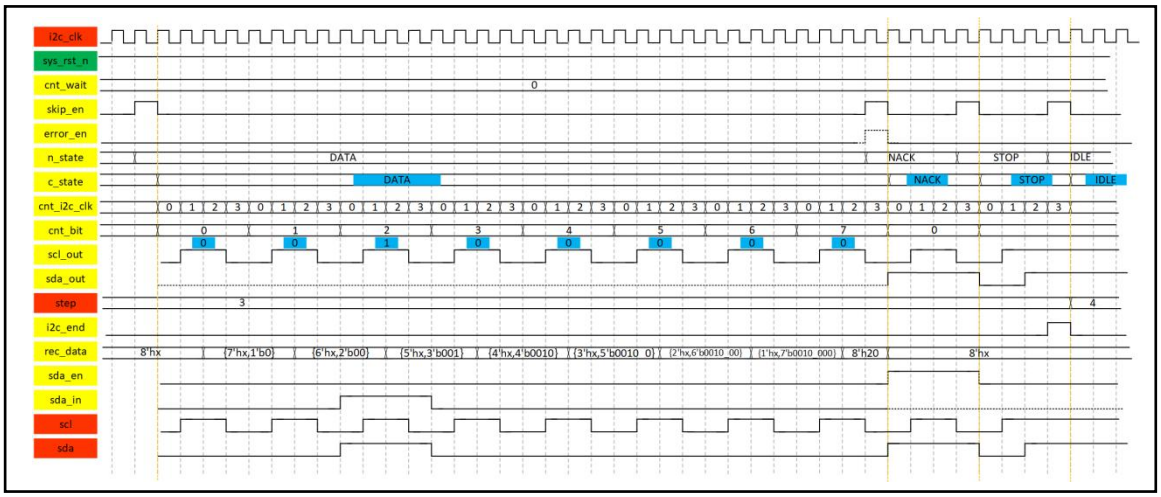

图 1.46 读数据信号波形图（2）

在笔者绘制上图信号波形图时，设定默认接收到的数据是“0x20”，skip_en 正常拉高；error_en 为虚线，表示只有当接收到的数据不是“0x20”，该信号才会拉高。

根据信号波形图，对读数据操作新引入的内部信号说明如表 1.6 所示：

表 1.6 读数据操作新引入的内部信号说明表

信号名称	信号类型	信号位宽	信号说明
error_en	reg	1	接收 0x20 数据错误信号
rec_data	reg	8	0x20 数据接收寄存信号

2.5 step4：配置寄存器组

配置寄存器组操作步骤也是需要使用到前面激活 Bank0 用过的写操作指令，执行该步骤时，I2C 控制模块需要与 paj7620u2 配置模块产生数据交互，见图 1.29。当配置模块将待配置的 1byte 寄存器地址（或需要进行激活 Bank 的 Bank 地址，下文统一简称为寄存器地址）及 1byte 数据，共计 16 位的 cfg_data 发送给 I2C 控制模块去执行时，同时也需要发送一个 i2c_start 信号，用来指示 I2C 控制模块由 IDLE 到 START 工作的开始。

如下前 8 位数据为寄存器地址，后 8 位数据为向寄存器内配置的数据：

```
1.  {8'hEF,8'h00}
2.  {8'h37,8'h07}
3.  {8'h38,8'h17}
4.  {8'h39,8'h06}
5.  {8'h42,8'h01}
6.  {8'h46,8'h2D}
7.  {8'h47,8'h0F}
8.  {8'h48,8'h3C}
9.  {8'h49,8'h00}
10. {8'h4A,8'h1E}
11. {8'h4C,8'h20}
12. {8'h51,8'h10}
13. {8'h5E,8'h10}
14. {8'h60,8'h27}
15. {8'h80,8'h42}
16. {8'h81,8'h44}
17. {8'h82,8'h04}
18. {8'h8B,8'h01}
19. {8'h90,8'h06}
20. {8'h95,8'h0A}
21. {8'h96,8'h0C}
22. {8'h97,8'h05}
23. {8'h9A,8'h14}
24. {8'h9C,8'h3F}
25. {8'hA5,8'h19}
26. {8'hCC,8'h19}
```

```
27. {8'hCD,8'h0B}
28. {8'hCE,8'h13}
29. {8'hCF,8'h64}
30. {8'hD0,8'h21}
31. {8'hEF,8'h01}
32. {8'h02,8'h0F}
33. {8'h03,8'h10}
34. {8'h04,8'h02}
35. {8'h25,8'h01}
36. {8'h27,8'h39}
37. {8'h28,8'h7F}
38. {8'h29,8'h08}
39. {8'h3E,8'hFF}
40. {8'h5E,8'h3D}
41. {8'h65,8'h96}
42. {8'h67,8'h97}
43. {8'h69,8'hCD}
44. {8'h6A,8'h01}
45. {8'h6D,8'h2C}
46. {8'h6E,8'h01}
47. {8'h72,8'h01}
48. {8'h73,8'h35}
49. {8'h74,8'h00}
50. {8'h77,8'h01}
51. {8'hEF,8'h00}
```

当 i2c_ctrl 模块完成对应寄存器配置时，即从 IDLE 状态跳转到 STOP 状态，也会产生一个操作完成信号，该信号为 paj7620u2 配置模块的开始工作信号 cfg_start，用来告诉该模块，i2c_ctrl 模块已经完成对数据的配置了，paj7620u2 配置模块可以打包新的数据发送给控制模块。

综上所述，两个模块不断进行数据的打包与开始配置指令的发送、数据的配置与配置完成指令的发送，当把所有的寄存器内部数据配置完成，则代表该操作步骤已经结束，传感器正在进行手势数据采集的功能，此时只需要我们把传感器采集到的手势数据从手势数据寄存器内读出并利用外设作出指示即可。

2.6 step5：读取手势数据（上）

手势数据寄存器地址为 0x43，寄存器功能见图 1.6，该步操作与 step2 使用的指令格式一致，因此不再赘述具体实现步骤。

需要注意的是，在数据手册上，除了单次读指令格式外，还存在连续读指令格式，如图 1.47 所示：

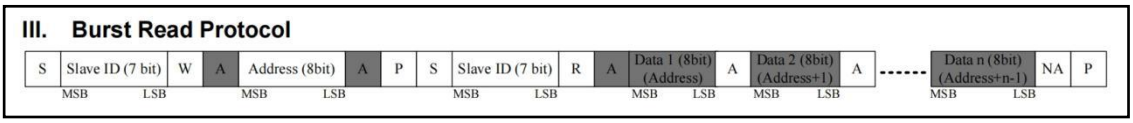

图 1.47 连续读指令

在设计时我们依然是使用单次读指令，笔者此前测试过连续读指令，发现读出的数据存在大量错误的情况，因此连续读指令可以把它忽略掉，读取手势数据不采用此指令。

2.7 step6：读取手势数据（下）

该步操作与 step3 使用的指令格式一致，不再赘述具体实现步骤。

本步骤中需要注意引入的步骤信号 step 的变化，通过前面绘制的信号波形图可知，step 在每个操作步骤结束后都会自增 1，表示进行到下一操作步骤。但是在读取手势数据时，需要重复读取而不是只读取一次，因此当 step 从 0 自增到 6 时，就一致保持 6 不会再改变了，表示状态会一直维持在读取手势数据这个操作步骤。

代码清单

```
1.  module  i2c_ctrl
2.  (
3.      input   wire            sys_clk     ,   //系统时钟，频率为 50MHZ
4.      input   wire            sys_rst_n   ,   //系统复位，低电平有效
5.      input   wire            i2c_start   ,   //i2c_ctrl 模块开始工作信号
6.      input   wire    [5:0]   cfg_num     ,   //配置完成的寄存器个数
7.      input   wire    [15:0]  cfg_data    ,   //待配置的寄存器(以地址区分)和向对
    应寄存器配置的数据
8.
9.      output  wire            scl         ,   //串行时钟信号
10.     output  reg             i2c_clk     ,   //I2C 驱动时钟
11.     output  reg     [2:0]   step        ,   //配置步骤信号
12.     output  reg             cfg_start   ,   //配置模块开始工作信号，打包数据传递
    给 i2c_ctrl 模块
13.     output  reg     [7:0]   po_data     ,   //输出的 8 位手势数据
14.
15.     inout   wire            sda             //串行数据信号
```

```
16. );
17.
18. parameter   CNT_CLK_MAX     =   6'd25   ;   //产生 1MHZ 分频计数最大值
19. parameter   CNT_WAIT_MAX    =   13'd1000;   //上电等待和发送唤醒指令后等待时
    间
20. parameter   SLAVE_ID        =   7'h73   ;   //从设备 ID 号
21. parameter   IDLE        =   4'd0    ,   //空闲状态
22.             START       =   4'd1    ,   //开始状态
23.             SLAVE_ADDR  =   4'd2    ,   //主机发送从设备 ID 号(地址)+1'b0/1'b1
    状态
24.             ACK_1       =   4'd3    ,   //从机向主机发送响应状态
25.             REG_ADDR    =   4'd4    ,   //主机发送寄存器地址状态
26.             ACK_2       =   4'd5    ,   //从机向主机发送响应状态
27.             DATA        =   4'd6    ,   //主机向从机发送配置的数据状态/从机向主
    机发送手势数据状态
28.             ACK_3       =   4'd7    ,   //从机向主机发送响应状态
29.             NACK        =   4'd8    ,   //主机向从机发送响应状态
30.             STOP        =   4'd9    ;   //结束状态
31.
32. reg     [5:0]   cnt_clk     ;   //分频计数器，计数最大值为 25-1
33. reg     [3:0]   n_state     ;   //三段式状态机次态
34. reg     [3:0]   c_state     ;   //三段式状态机现态
35. reg     [12:0]  cnt_wait    ;   //等待计数器，计数最大值为 1000-1
36. reg             skip_en     ;   //状态跳转信号
37. reg             error_en    ;   //错误信号，没有接收到 0x20 该信号拉高
38. reg     [1:0]   cnt_i2c_clk ;   //i2c_clk 时钟个数计数器
39. reg     [2:0]   cnt_bit     ;   //接收到的比特数据计数器
40. reg     [7:0]   rec_data    ;   //接收到的 0x20 数据寄存器
41. reg     [7:0]   po_data_reg ;   //输出的手势数据寄存
42. reg             scl_out     ;   //主机产生输出至外围 I2C 设备的 SCL 信号
43. reg             sda_out     ;   //主机产生输出至外围 I2C 设备的 SDA 信号
44. reg             i2c_end     ;   //结束信号，STOP 状态最后一个时钟周期拉高
45. reg     [7:0]   slave_addr  ;   //SLVAE_ADDR 需要发送的数据寄存
46. reg     [7:0]   reg_addr    ;   //REG_ADDR 需要发送的数据寄存
47. reg     [7:0]   wr_data     ;   //DATA 状态需要主机发送的数据寄存
48. wire            sda_in      ;   //从设备发送给主设备的数据
49. wire            sda_en      ;   //主机控制 SDA 总线使能信号
50.
51. //scl
52. assign  scl     =  scl_out  ;
53. //三态门
54. assign  sda_in  =  sda  ;
```

```
assign  sda_en  =  ((c_state == ACK_1)||(c_state == ACK_2)||(c_state == ACK_3)||((c_state == DATA)&&((step == 3'd3)||(step == 3'd6)))) ? 1'b0 : 1'b1  ;

assign  sda     =  (sda_en == 1'b1) ? sda_out : 1'bz  ;

//cfg_start
always@(posedge i2c_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        cfg_start  <=  1'b0  ;
    else
        cfg_start  <=  i2c_end  ;

//slave_addr、reg_addr、wr_data
always@(*)
    case(step)
        3'd0    :begin
                    slave_addr  <=  {SLAVE_ID,1'b0}  ;
                    reg_addr    <=  8'h0  ;      //未使用
                    wr_data     <=  8'h0  ;     //未使用
                 end
        3'd1    :begin
                    slave_addr  <=  {SLAVE_ID,1'b0}  ;
                    reg_addr    <=  8'hEF  ; //发送 0xEF  Bank 地址
                    wr_data     <=  8'h00  ; //向 bank 地址里面写入 0x00 数据，激活 bank0
                 end
        3'd2    :begin
                    slave_addr  <=  {SLAVE_ID,1'b0}  ;
                    reg_addr    <=  8'h00  ; //发送 0x00 寄存器地址
                    wr_data     <=  8'h00  ; //未使用
                 end
        3'd3    :begin
                    slave_addr  <=  {SLAVE_ID,1'b1}  ;
                    reg_addr    <=  8'h00  ; //未使用
                    wr_data     <=  8'h00  ; //未使用
                 end
        3'd4    :begin
                    slave_addr  <=  {SLAVE_ID,1'b0}  ;
                    reg_addr <=  cfg_data[15:8]  ;
                    wr_data     <=  cfg_data[7:0]   ;
                 end
        3'd5    :begin
                    slave_addr  <=  {SLAVE_ID,1'b0}  ;
                    reg_addr    <=  8'h43  ;
```

```
                    wr_data     <=  8'h0   ;  //未使用
                end
        3'd6    :begin
                    slave_addr  <=  {SLAVE_ID,1'b1}  ;
                    reg_addr    <=  8'h00  ; //未使用
                    wr_data     <=  8'h00  ; //未使用
                end
        default :begin
                    slave_addr  <=  8'h0  ;
                    reg_addr    <=  8'h0  ;
                    wr_data     <=  8'h0  ;
                end
    endcase

//cnt_clk
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        cnt_clk  <=  6'd0  ;
    else  if(cnt_clk == CNT_CLK_MAX - 1'b1)
        cnt_clk  <=  6'd0  ;
    else
        cnt_clk  <=  cnt_clk + 1'b1  ;

//i2c_clk
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        i2c_clk  <=  1'b0  ;
    else  if(cnt_clk == CNT_CLK_MAX - 1'b1)
        i2c_clk  <=  ~i2c_clk  ;
    else
        i2c_clk  <=  i2c_clk  ;

//状态机第一段：描述现态和次态关系
always@(posedge i2c_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        c_state  <=  IDLE  ;
    else
        c_state  <=  n_state  ;

//状态机第二段：描述状态跳转情况
always@(*)
    case(c_state)
        IDLE        :   if(skip_en == 1'b1)
                            n_state  =  START  ;
```

```
140.                            else
141.                                n_state  =  IDLE  ;
142.          START       :     if(skip_en == 1'b1)
143.                                n_state  =  SLAVE_ADDR  ;
144.                            else
145.                                n_state  =  START  ;
146.          SLAVE_ADDR  :     if((skip_en == 1'b1)&&(step == 3'd0))
147.                                n_state  =  STOP  ;
148.                            else  if(skip_en == 1'b1)
149.                                n_state  =  ACK_1  ;
150.                            else
151.                                n_state  =  SLAVE_ADDR  ;
152.          ACK_1       :     if((skip_en == 1'b1)&&((step == 3'd3)||(step == 3'd
    6)))
153.                                n_state  =  DATA  ;
154.                            else  if(skip_en == 1'b1)
155.                                n_state  =  REG_ADDR  ;
156.                            else
157.                                n_state  =  ACK_1  ;
158.          REG_ADDR    :     if(skip_en == 1'b1)
159.                                n_state  =  ACK_2  ;
160.                            else
161.                                n_state  =  REG_ADDR  ;
162.          ACK_2       :     if((skip_en == 1'b1)&&((step == 3'd1)||(step == 3'd
    4)))
163.                                n_state  =  DATA  ;
164.                            else  if(skip_en == 1'b1)
165.                                n_state  =  STOP  ;
166.                            else
167.                                n_state  =  ACK_2  ;
168.          DATA        :     if((skip_en == 1'b1)&&((step == 3'd1)||(step == 3'd
    4)))
169.                                n_state  =  ACK_3  ;
170.                            else  if(skip_en == 1'b1)
171.                                n_state  =  NACK  ;
172.                            else  if(error_en == 1'b1)
173.                                n_state  =  IDLE  ;
174.                            else
175.                                n_state  =  DATA  ;
176.          ACK_3       :     if(skip_en == 1'b1)
177.                                n_state  =  STOP  ;
178.                            else
179.                                n_state  =  ACK_3  ;
180.          NACK        :     if(skip_en == 1'b1)
```

```
                            n_state  =  STOP  ;
                        else
                            n_state  =  NACK  ;
        STOP        :   if(skip_en == 1'b1)
                            n_state  =  IDLE  ;
                        else
                            n_state  =  STOP  ;
        default     :   n_state  =  IDLE  ;
    endcase

//状态机第三段：对各信号进行时序逻辑赋值
always@(posedge i2c_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        begin
            cnt_wait    <=  13'd0   ;
            skip_en     <=  1'b0 ;
            error_en    <=  1'b0 ;
            cnt_i2c_clk <=  2'd0 ;
            cnt_bit     <=  3'd0 ;
            i2c_end     <=  1'b0 ;
            step        <=  3'd0 ;
        end
    else
        case(c_state)
            IDLE        :begin
                            if(cnt_wait == CNT_WAIT_MAX - 1'b1)
                                cnt_wait  <=  13'd0  ;
                            else
                                cnt_wait  <=  cnt_wait + 1'b1  ;
                            if((i2c_start == 1'b1)&&(step == 3'd4))
                                skip_en  <=  1'b1  ;
                            else  if((cnt_wait == CNT_WAIT_MAX - 2'd2)&&(st
   ep != 3'd4))
                                skip_en  <=  1'b1  ;
                            else
                                skip_en  <=  1'b0  ;
                        end
            START       :begin
                            cnt_i2c_clk  <=  cnt_i2c_clk + 1'b1  ;
                            if(cnt_i2c_clk == 2'd2)
                                skip_en  <=  1'b1  ;
                            else
                                skip_en  <=  1'b0  ;
                        end
```

```
224.              SLAVE_ADDR,REG_ADDR
225.                              :begin
226.                                  cnt_i2c_clk  <=  cnt_i2c_clk + 1'b1  ;
227.                                  if(cnt_i2c_clk == 2'd3)
228.                                      cnt_bit  <=  cnt_bit + 1'b1  ;
229.                                  else
230.                                      cnt_bit  <=  cnt_bit  ;
231.                                  if((cnt_i2c_clk == 2'd2)&&(cnt_bit == 3'd7))
232.                                      skip_en  <=  1'b1  ;
233.                                  else
234.                                      skip_en  <=  1'b0  ;

235.                               end
236.              ACK_1,ACK_2,ACK_3
237.                              :begin
238.                                  cnt_i2c_clk  <=  cnt_i2c_clk + 1'b1  ;
239.                                  if((cnt_i2c_clk == 2'd2)&&(sda_in == 1'b0))
240.                                      skip_en  <=  1'b1  ;
241.                                  else
242.                                      skip_en  <=  1'b0  ;
243.                               end
244.              DATA            :begin
245.                                  cnt_i2c_clk  <=  cnt_i2c_clk + 1'b1  ;
246.                                  if(cnt_i2c_clk == 2'd3)
247.                                      cnt_bit  <=  cnt_bit + 1'b1  ;
248.                                  else
249.                                      cnt_bit  <=  cnt_bit  ;

250.                                  if((cnt_i2c_clk == 2'd2)&&(cnt_bit == 3'd7)&&(s
    tep == 3'd3)&&(rec_data == 8'h20))
251.                                      skip_en  <=  1'b1  ;
252.                                  else  if((cnt_i2c_clk == 2'd2)&&(cnt_bit == 3'd
    7))
253.                                      skip_en  <=  1'b1  ;
254.                                  else
255.                                      skip_en  <=  1'b0  ;

256.                                  if((cnt_i2c_clk == 2'd2)&&(cnt_bit == 3'd7)&&(s
    tep == 3'd3)&&(rec_data != 8'h20))
257.                                      begin
258.                                          error_en  <=  1'b1  ;
259.                                          step      <=  3'd0  ;
260.                                      end
261.                                  else
```

```
                                begin
                                    error_en  <=  1'b0  ;
                                    step      <=  step  ;
                                end
                            end
                NACK        :begin
                                cnt_i2c_clk  <=  cnt_i2c_clk + 1'b1  ;

                                if((cnt_i2c_clk == 2'd2)&&(sda_out == 1'b1))
                                    skip_en  <=  1'b1  ;
                                else
                                    skip_en  <=  1'b0  ;

                            end
                STOP        :begin
                                cnt_i2c_clk  <=  cnt_i2c_clk + 1'b1  ;
                                if(cnt_i2c_clk == 2'd2)
                                    skip_en  <=  1'b1  ;
                                else
                                    skip_en  <=  1'b0  ;
                                if(cnt_i2c_clk == 2'd2)
                                    i2c_end  <=  1'b1  ;
                                else
                                    i2c_end  <=  1'b0  ;
                                if((i2c_end == 1'b1)&&(step <= 3'd3))
                                    step  <=  step + 1'b1  ;
                                else  if((i2c_end == 1'b1)&&(step == 3'd4)&&(cf
    g_num == 6'd51))
                                    step  <=  step + 1'b1  ;
                                else  if((i2c_end == 1'b1)&&(step == 3'd5))
                                    step  <=  step + 1'b1  ;
                                else
                                    step  <=  step  ;
                            end
                default     :begin
                                cnt_wait    <=  13'd0   ;
                                skip_en     <=  1'b0 ;
                                error_en    <=  1'b0 ;
                                cnt_i2c_clk <=  2'd0 ;
                                cnt_bit     <=  3'd0 ;
                                i2c_end     <=  1'b0     ;
                                step        <=  3'd0 ;
                            end
            endcase
```

```
303.
304. //状态机第三段：对 po_data_reg 时序逻辑赋值
305. always@(posedge i2c_clk or negedge sys_rst_n)
306.     if(sys_rst_n == 1'b0)
307.         po_data_reg  <=  8'h0  ;
308.     else
309.         case(c_state)
310.             DATA    :   if((step == 3'd6)&&(cnt_i2c_clk == 2'd1))
311.                             po_data_reg  <=  {po_data_reg[6:0],sda_in}  ;
312.                         else
313.                             po_data_reg  <=  po_data_reg  ;
314.             default :   po_data_reg  <=  po_data_reg  ;
315.         endcase
316.
317. //状态机第三段：对 po_data 时序逻辑赋值
318. always@(posedge i2c_clk or negedge sys_rst_n)
319.     if(sys_rst_n == 1'b0)
320.         po_data  <=  8'h0  ;
321.     else
322.         case(c_state)
323.             DATA    :   if((step == 3'd6)&&(cnt_i2c_clk == 2'd3)&&(cnt_bit
    == 3'd7)&&(po_data_reg != 8'h0))
324.                             po_data  <=  po_data_reg  ;
325.                         else
326.                             po_data  <=  po_data  ;
327.             default :   po_data  <=  po_data  ;
328.         endcase
329.
330. //状态机第三段：对 rec_data 时序逻辑赋值
331. always@(posedge i2c_clk or negedge sys_rst_n)
332.     if(sys_rst_n == 1'b0)
333.         rec_data  <=  8'h0  ;
334.     else
335.         case(c_state)
336.             DATA    :   if((step == 3'd3)&&(cnt_i2c_clk == 2'd1))
337.                             rec_data  <=  {rec_data[6:0],sda_in}  ;
338.                         else
339.                             rec_data  <=  rec_data  ;
340.             default :   rec_data  <=  rec_data  ;
341.         endcase
342.
343. //状态机第三段：对 scl_out 组合逻辑赋值
344. always@(*)
345.     case(c_state)
```

```
346.          IDLE         :   scl_out  =  1'b1  ;
347.          START        :   if(cnt_i2c_clk == 2'd3)
348.                               scl_out  =  1'b0  ;
349.                           else
350.                               scl_out  =  1'b1  ;
351.          SLAVE_ADDR,REG_ADDR,DATA,ACK_1,ACK_2,ACK_3,NACK
352.                       :   if((cnt_i2c_clk == 2'd0)||(cnt_i2c_clk == 3'd3))
353.                               scl_out  =  1'b0  ;
354.                           else
355.                               scl_out  =  1'b1  ;
356.          STOP         :   if(cnt_i2c_clk == 2'd0)
357.                               scl_out  =  1'b0  ;
358.                           else
359.                               scl_out  =  1'b1  ;
360.          default      :   scl_out  =  1'b1  ;
361.      endcase
362.
363. //状态机第三段：对 sda_out 组合逻辑赋值
364. always@(*)
365.      case(c_state)
366.          IDLE         :   sda_out  =  1'b1  ;
367.          START        :   if(cnt_i2c_clk == 2'd0)
368.                               sda_out  =  1'b1  ;
369.                           else
370.                               sda_out  =  1'b0  ;
371.          SLAVE_ADDR   :   sda_out  =  slave_addr[7 - cnt_bit]  ;
372.          REG_ADDR     :   sda_out  =  reg_addr[7 - cnt_bit]  ;
373.          DATA         :   sda_out  =  wr_data[7 - cnt_bit]  ;
374.          ACK_1,ACK_2,ACK_3
375.                       :   sda_out  =  1'b0  ;
376.          NACK         :   sda_out  =  1'b1  ;
377.          STOP         :   if((cnt_i2c_clk == 2'd0)||(cnt_i2c_clk == 2'd1))
378.                               sda_out  =  1'b0  ;
379.                           else
380.                               sda_out  =  1'b1  ;
381.          default      :   sda_out  =  1'b1  ;
382.      endcase
383.
384. endmodule
```

代码第 53 至 56 行是通过 sda_en 描述 sda 的状态，当 sda_en 为高电平时，表示主机要占用 SDA 通信总线，此时就把主机产生的 sda_out 赋值给 sda 即可；

当 sda_en 为低电平时，表示主机此时要放弃对 SDA 通信总线的控制权，主机给 sda 信号赋高阻态表示放弃控制。

sda_in 表示的是从机传输给主机的数据，在主机接收从机响应状态，或者是接收从机传递的手势数据时，需要用到此信号。

代码第 58 至 63 行是将 i2c_end 进行打拍操作，即可得到配置模块的开始工作信号，但此信号在配置模块具体什么时候起作用还需要结合 step 信号一起判断；

代码第 65 至 108 行是对 SLAVE_ADDR、REG_ADDR、DATA 状态主机需要发送到从机的数据寄存，便于之后赋值给 i2c_sda；

代码第 110 至 126 行是产生新的驱动时钟 i2c_clk，用该时钟来生成与 I2C 通信相关的信号；

代码第 135 至 189 行是状态机第二段，只要理清每个步骤，从 step0 到 step6 状态跳转信号需要跳转的状态，即可完成此段代码的编写；

代码第 191 至 302 行是状态机第三段，这里需要着重说明一下第 256 行至 265 行，这里如果接收到的数据不是 0x20，就代表唤醒操作是失败的，此时不仅是错误信号 error_en 需要拉高，还需要将 step 清零，重新执行唤醒操作；

代码第 304 至 315 行是对输出数据寄存赋值，因为我们希望得到的是一个稳定持续的手势数据，而得到 8 位数据必须要一位一位进行拼接，将 DATA 状态下的 8 位数据全部拼接完成才能得到完整的手势数据，因此 po_data_reg 只是起到的拼接数据作用，拼接完成最后一位才是需要的手势数据，所以要在结尾将最后拼接得到的数据重新赋值一下，才得到正确的手势数据；

代码第 317 行至 328 行是对输出数据赋值，在拼接数据完成后，即 cnt_bit 到最后一位时，才能够将 po_data_reg 赋值给 po_data，并且一定要注意的是，当采集到的数据为全 0 时，就代表没有识别到手势，此时不能将 po_data_reg 赋值给 po_data，否则这个变化过程会特别快，如果利用 LED 灯作指示，LED 灯是持续保持熄灭的状态。

因此，我们需要更换设计思路，当采集到的 8bit 手势数据不为全 0，则代表有手势数据，可以赋值；反之不进行赋值。这样可以保留原先的手势数据不清零，使 LED 灯常亮指示。

代码第 343 行至 384 行是对主机产生的输出到外围设备的串行时钟信号和串行数据信号赋值，需要注意的是第 371 至 373 行，这里随着 cnt_bit 从 0 到 7 不

断自增，把 slave_addr、reg_addr、wr_data 从高位到低位依次赋值给了 sda_out，因为三个变量会随着 step 信号变化而变化，所以对 scl_out 和 sda_out 代码编写完成后，就可以不用修改这部分了。

3. PAJ7620U2 寄存器组配置模块

3.1 模块框图

模块框图如图 1.48 所示：

图 1.48 PAJ7620U2 寄存器配置模块框图

据模块框图，对模块端口信号说明如表 1.7 所示：

表 1.7 paj7620_cfg 模块端口信号说明

信号名称	信号端口类型	信号位宽	信号说明
i2c_clk	input	1	I2C 驱动时钟，频率为 1MHZ
sys_rst_n	input	1	系统复位，低电平有效
step	input	3	操作步骤信号，不同操作步骤对应不同配置状态跳转
cfg_start	input	1	寄存器配置模块开始工作信号
cfg_num	output	6	寄存器配置模块打包完成的数据个数
cfg_data	output	16	寄存器配置模块打包的待配置的寄存器地址和向该地址内写入的数据
i2c_start	output	1	I2C 模块开始工作信号

3.2 信号波形图

根据模块框图，绘制的信号波形图如图 1.49 所示：

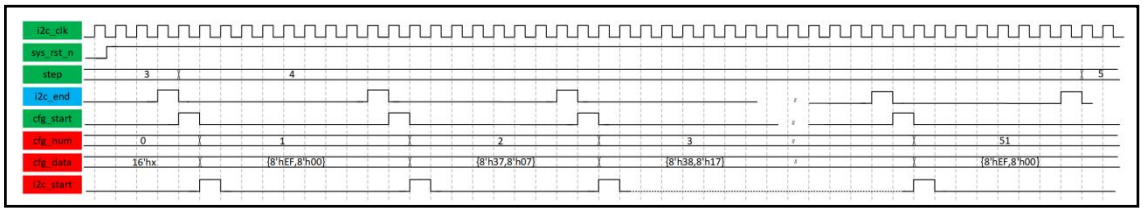

图 1.49　PAJ7620U2 寄存器组配置信号波形图

i2c_end 填充为蓝色表示该信号在该模块只作为辅助指示，不参与赋值操作。由信号波形图可得，只有当 step == 4 时，i2c_end 打拍信号 cfg_start 才在该模块内部起作用。

代码清单

```
1.  module  paj7620_cfg
2.  (
3.      input   wire            i2c_clk     ,   //I2C 驱动时钟，频率为 50MHZ
4.      input   wire            sys_rst_n   ,   //系统复位，低电平有效
5.      input   wire    [2:0]   step        ,   //操作步骤信号
6.      input   wire            cfg_start   ,   //配置模块开始工作信号
7.
8.      output  reg     [5:0]   cfg_num     ,   //寄存器配置的个数
9.      output  wire    [15:0]  cfg_data    ,   //待配置的寄存器
10.     output  reg             i2c_start       //i2c_ctrl 模块开始工作信号
11. );
12.
13. wire    [15:0]  cfg_data_reg[50:0]  ;   //数组，寄存待配置的寄存器地址和数据
14.
15. //cfg_num
16. always@(posedge i2c_clk or negedge sys_rst_n)
17.     if(sys_rst_n == 1'b0)
18.         cfg_num  <=  6'd0   ;
19.     else  if((cfg_start == 1'b1)&&(step == 3'd4))
20.         cfg_num  <=  cfg_num + 1'b1  ;
21.     else
22.         cfg_num  <=  cfg_num  ;
23.
24. //i2c_start
```

```
always@(posedge i2c_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        i2c_start  <=  1'b0  ;
    else  if((cfg_start == 1'b1)&&(step == 3'd4))
        i2c_start  <=  1'b1  ;
    else
        i2c_start  <=  1'b0  ;

//cfg_data
assign  cfg_data  =  (cfg_num != 6'd0) ? cfg_data_reg[cfg_num - 1'b1] : 16'd0  ;

//cfg_data_reg
assign  cfg_data_reg[00]    =   {8'hEF,8'h00}  ;    //激活 Bank0
assign  cfg_data_reg[01]    =   {8'h37,8'h07}  ;
assign  cfg_data_reg[02]    =   {8'h38,8'h17}  ;
assign  cfg_data_reg[03]    =   {8'h39,8'h06}  ;
assign  cfg_data_reg[04]    =   {8'h42,8'h01}  ;
assign  cfg_data_reg[05]    =   {8'h46,8'h2D}  ;
assign  cfg_data_reg[06]    =   {8'h47,8'h0F}  ;
assign  cfg_data_reg[07]    =   {8'h48,8'h3C}  ;
assign  cfg_data_reg[08]    =   {8'h49,8'h00}  ;
assign  cfg_data_reg[09]    =   {8'h4A,8'h1E}  ;
assign  cfg_data_reg[10]    =   {8'h4C,8'h20}  ;
assign  cfg_data_reg[11]    =   {8'h51,8'h10}  ;
assign  cfg_data_reg[12]    =   {8'h5E,8'h10}  ;
assign  cfg_data_reg[13]    =   {8'h60,8'h27}  ;
assign  cfg_data_reg[14]    =   {8'h80,8'h42}  ;
assign  cfg_data_reg[15]    =   {8'h81,8'h44}  ;
assign  cfg_data_reg[16]    =   {8'h82,8'h04}  ;
assign  cfg_data_reg[17]    =   {8'h8B,8'h01}  ;
assign  cfg_data_reg[18]    =   {8'h90,8'h06}  ;
assign  cfg_data_reg[19]    =   {8'h95,8'h0A}  ;
assign  cfg_data_reg[20]    =   {8'h96,8'h0C}  ;
assign  cfg_data_reg[21]    =   {8'h97,8'h05}  ;
assign  cfg_data_reg[22]    =   {8'h9A,8'h14}  ;
assign  cfg_data_reg[23]    =   {8'h9C,8'h3F}  ;
assign  cfg_data_reg[24]    =   {8'hA5,8'h19}  ;
assign  cfg_data_reg[25]    =   {8'hCC,8'h19}  ;
assign  cfg_data_reg[26]    =   {8'hCD,8'h0B}  ;
assign  cfg_data_reg[27]    =   {8'hCE,8'h13}  ;
assign  cfg_data_reg[28]    =   {8'hCF,8'h64}  ;
assign  cfg_data_reg[29]    =   {8'hD0,8'h21}  ;
assign  cfg_data_reg[30]    =   {8'hEF,8'h01}  ;    //激活 Bank1
```

```
68. assign  cfg_data_reg[31]    =   {8'h02,8'h0F}  ;
69. assign  cfg_data_reg[32]    =   {8'h03,8'h10}  ;
70. assign  cfg_data_reg[33]    =   {8'h04,8'h02}  ;
71. assign  cfg_data_reg[34]    =   {8'h25,8'h01}  ;
72. assign  cfg_data_reg[35]    =   {8'h27,8'h39}  ;
73. assign  cfg_data_reg[36]    =   {8'h28,8'h7F}  ;
74. assign  cfg_data_reg[37]    =   {8'h29,8'h08}  ;
75. assign  cfg_data_reg[38]    =   {8'h3E,8'hFF}  ;
76. assign  cfg_data_reg[39]    =   {8'h5E,8'h3D}  ;
77. assign  cfg_data_reg[40]    =   {8'h65,8'h96}  ;
78. assign  cfg_data_reg[41]    =   {8'h67,8'h97}  ;
79. assign  cfg_data_reg[42]    =   {8'h69,8'hCD}  ;
80. assign  cfg_data_reg[43]    =   {8'h6A,8'h01}  ;
81. assign  cfg_data_reg[44]    =   {8'h6D,8'h2C}  ;
82. assign  cfg_data_reg[45]    =   {8'h6E,8'h01}  ;
83. assign  cfg_data_reg[46]    =   {8'h72,8'h01}  ;
84. assign  cfg_data_reg[47]    =   {8'h73,8'h35}  ;
85. assign  cfg_data_reg[48]    =   {8'h74,8'h00}  ;
86. assign  cfg_data_reg[49]    =   {8'h77,8'h01}  ;
87. assign  cfg_data_reg[50]    =   {8'hEF,8'h00}  ;    //激活 Bank0
88.
89. endmodule
```

代码第 13 行表示定义了一个数组，数据位宽为 16，数组元素个数为 51 个；

代码第 15 至 22 行是对 cfg_num 信号赋值，用来指示配置的寄存器个数，每检测到配置模块开始工作信号，该信号就会自增 1；

代码第 24 至 31 行是对 i2c_start 信号赋值，该信号相当于是在 step == 4 时，对 cfg_start 进行打拍后得到的，这样可以让 cfg_num、i2c_start、cfg_data 三个信号同步在同一时刻时钟上升沿；

代码第 36 至 87 行是对所有待配置的寄存器的地址和数据进行声明，在结尾再激活一次 Bank0 是因为前面将 Bank1 激活了，寄存器组配置完成需要读取手势数据，而手势数据寄存器 0x43 是隶属于 Bank0 区域的，为了防止激活 Bank1 对激活 Bank0 有影响，因此设计再次激活一下 Bank0。

4. LED 灯控制模块

4.1 模块框图

模块框图如图 1.50 所示：

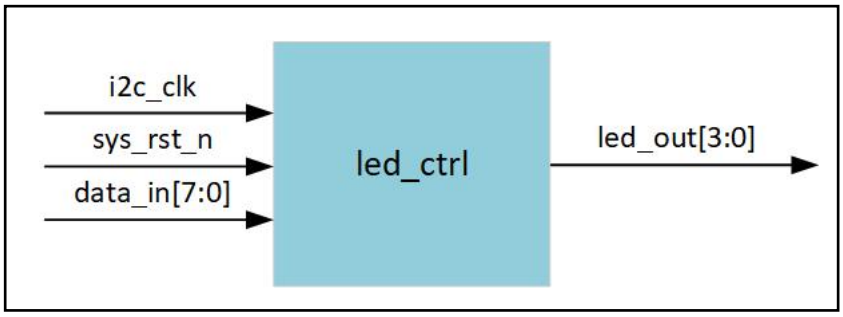

图 1.50　LED 灯控制模块框图

4.2　信号波形图

该模块逻辑功能较为简单，直接编写代码即可，无需信号波形图作辅助。

代码清单

```
1.  module  led_ctrl
2.  (
3.      input   wire            i2c_clk     ,
4.      input   wire            sys_rst_n   ,
5.      input   wire    [7:0]   data_in     ,
6.
7.      output  reg     [3:0]   led_out
8.  );
9.
10. always@(posedge i2c_clk or negedge sys_rst_n)
11.     if(sys_rst_n == 1'b0)
12.         led_out  <=  4'b0000  ;
13.     else
14.         led_out  <=  data_in[3:0]  ;
15.
16. endmodule
```

1.4.4 仿真测试

对顶层文件进行仿真测试，编写的测试代码为：

```
1.  `timescale  1ns/1ns     //定义时间参数和时间精度
2.  module  tb_ges_recognize();
3.  reg             sys_clk     ;
4.  reg             sys_rst_n   ;
5.  wire            scl         ;
6.  tri0            sda         ;   //主机不占用 SDA 总线时，一直将它置 0
7.
```

```
8.  initial
9.  begin
10.     sys_clk     <=  1'b0 ;
11.     sys_rst_n   <=  1'b0 ;
12.     #100
13.     sys_rst_n   <=  1'b1 ;
14. end
15.
16. always #10 sys_clk  <=  ~sys_clk  ;
17.
18. //参数重定义
19. defparam  ges_recognize_inst.i2c_ctrl_inst.CNT_WAIT_MAX = 13'd20  ,
20.           ges_recognize_inst.i2c_ctrl_inst.CNT_CLK_MAX = 6'd2  ;
21.
22. ges_recognize  ges_recognize_inst
23. (
24.     .sys_clk    (sys_clk    ),
25.     .sys_rst_n  (sys_rst_n  ),
26.     .scl        (scl        ),
27.     .sda        (sda        )
28. );
29.
30. endmodule
```

代码第 6 行表示主机在不占用 SDA 通信总线时，SDA 总线上的数据一直为 0。这样设计可以在主机接收从机响应状态，模拟接收到的是正确的响应信号。

代码第 18 至 20 行是重定义计数最大值参数，第 19 行是重定义上电等待时间，否则 Modelsim 仿真出来的波形很长一段都是在等待状态，不利于观察；第 20 行是重定义 i2c_clk 时钟分频计数最大值，产生周期小一点的时钟，这样设计也是便于观察。

对于仿真出的信号波形，我们主要关注的是主机发送的数据以及格式是否正确，而从机返回的数据，则需要抓取实际的信号波形进行判定。因此，Modelsim 仿真波形中，只要每个操作步骤以及状态的串行时钟、串行数据信号是按照预定格式发送的，那么使用 Modelsim 验证就是没问题的。如图 1.51 为唤醒操作信号波形图：

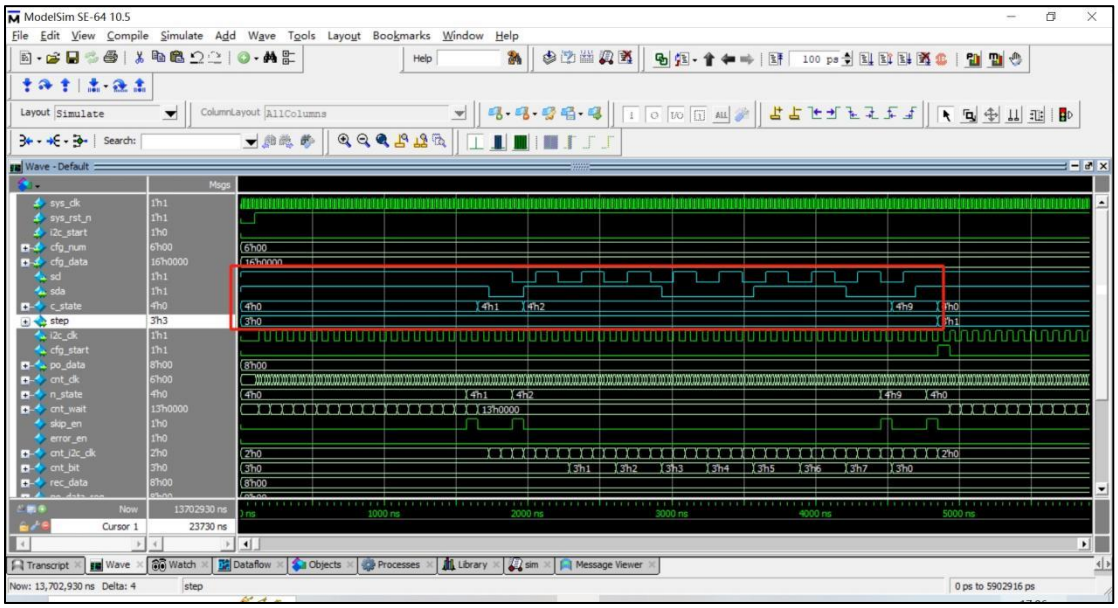

图 1.51 唤醒操作信号波形图

根据唤醒操作仿真出的信号波形图可知：①在 IDLE 状态下，等待一定时间，状态跳转到 START 状态；②在 START 状态下，scl 为高电平时 sda 产生下降沿，状态跳转到 SLAVE_ADDR 状态；③在 SLAVE_ADDR 状态下，发送的 8bit 数据为 8'b1110_0110（7'h73+1'b0），完成后跳转到 STOP 状态；④在 STOP 状态下，scl 为高电平时 sda 产生上升沿，状态重新回到 IDLE 状态。唤醒操作波形仿真验证无误。

激活 Bank0 操作信号波形图如图 1.52 所示：

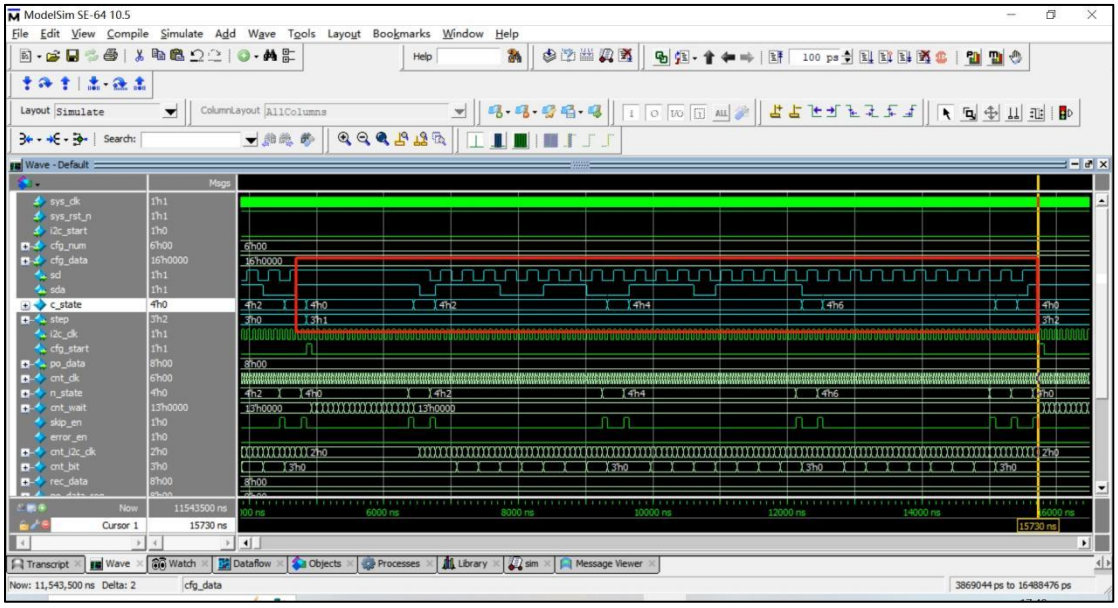

图 1.52 激活 Bank0 信号波形图

根据激活 Bank0 信号波形图可知，SLAVE_ADDR 状态发送的数据为 8’b1110_0110，REG_ADDR 状态发送的数据为 8’hEF，DATA 状态发送的数据为 8’h00，其它状态也都按照预定格式发送。激活 Bank0 波形仿真无误。

读取 0x00 寄存器数据前半部分信号波形图如图 1.53 所示：

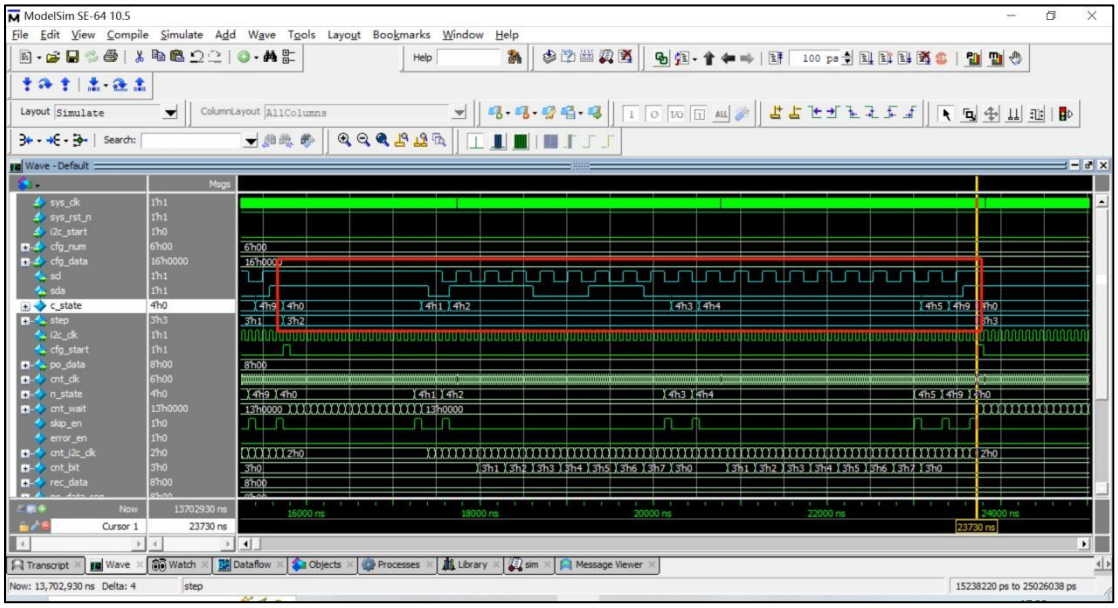

图 1.53 读取 0x00 寄存器前半部分数据信号波形图

如图 1.53 所示，SLAVE_ADDR 发送的数据为 7’h73+1’b0，REG_ADDR 发送的数据为 8’h00，并且发送数据完成后都模拟接收到了正确的响应，其它状态也没有问题。波形验证无误。

读取 0x00 寄存器数据信号波形图如图 1.54 所示：

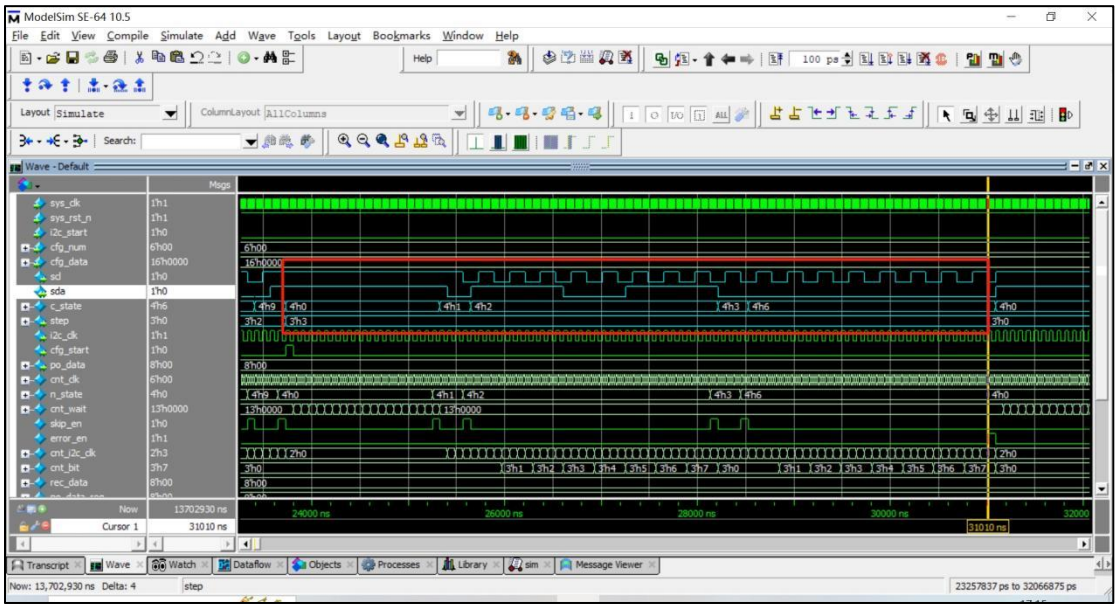

图 1.54 读取 0x00 寄存器数据信号波形图（1）

如图 1.54 所示，SLAVE_ADDR 发送的数据为 7’h73+1’b1，DATA 是由从机返回至主机的数据，因为编写测试代码时模拟接收从机返回的数据一直为 0，故接收到的数据一直为全 0，未接收到“0x20”数据，step 变为 0，重新执行唤醒操作。

此时使用 Modelsim 进行波形仿真就不合适了，使用 Signal Tap 抓取的读取 0x00 寄存器数据信号波形如图 1.55 所示：

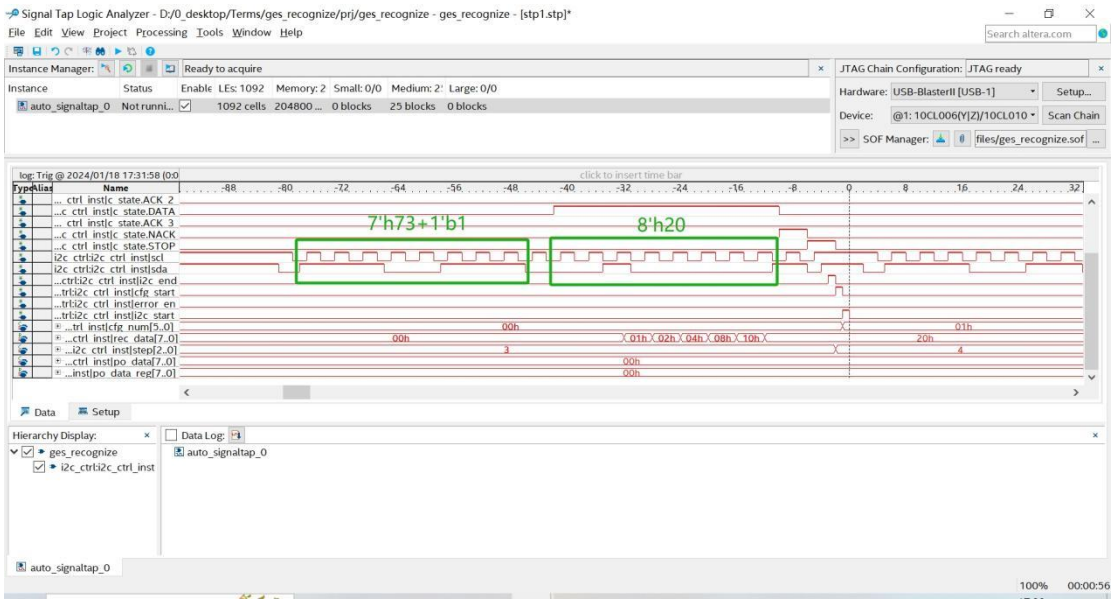

图 1.55 读取 0x00 寄存器数据信号波形图（2）

如图 1.55 所示，可以很清晰地看出，在 SLAVE_ADDR 状态下，主机发送的数据为 7'h73+1'b1，在 DATA 状态下主机接收的数据为 8'h20，接收数据为 0x20，代表唤醒操作成功，进行到下一操作步骤。

抓取后续信号波形图如图 1.56 所示：

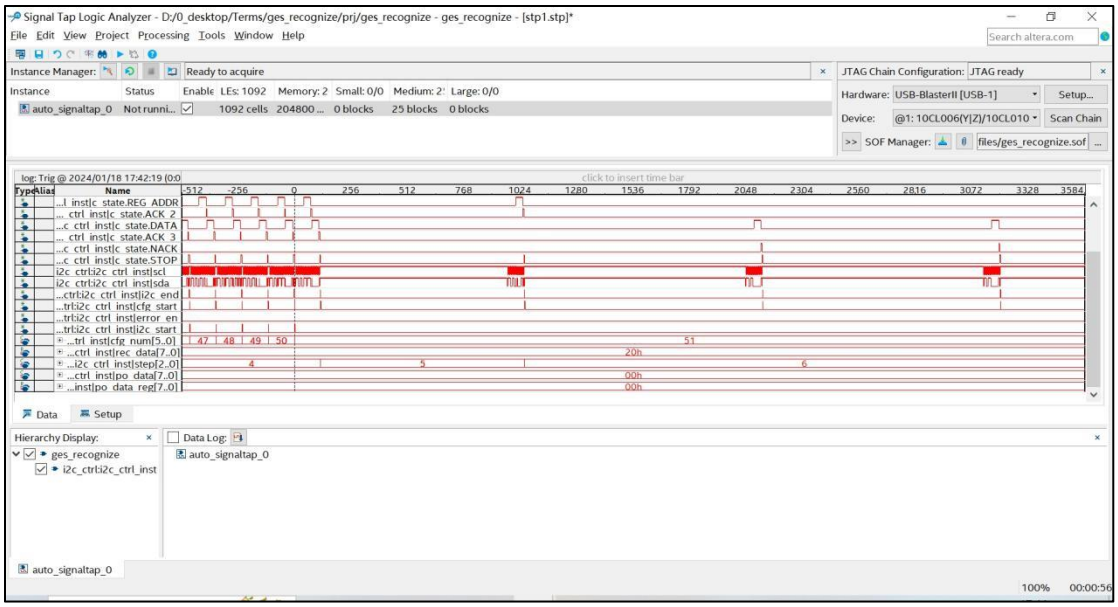

图 1.56　Signal Tap 抓取后续信号波形

如图 1.56 所示，使用 Signal Tap 抓取到的信号波形图可知，cfg_num 配置完成 51 个寄存器后，在 step=5 时主机开始指定读取手势数据寄存器 0x43 寄存器内的数据，step=6 时主机连续不断地读取手势数据，与预期设定一致。

1.4.5 上板验证

引脚绑定如表 1.8 所示：

表 1.8 引脚绑定配置

信号名称	信号端口类型	绑定引脚
sys_clk	input	E1
sys_rst_n	input	E15
scl	output	N14
sda	inout	M12
led_out[0]	output	G15
led_out[1]	output	F16

led_out[2]	output	F15
led_out[3]	output	D16

使用手势识别传感器时，红框处“小眼睛”应正对操作者，如图 1.57 所示：

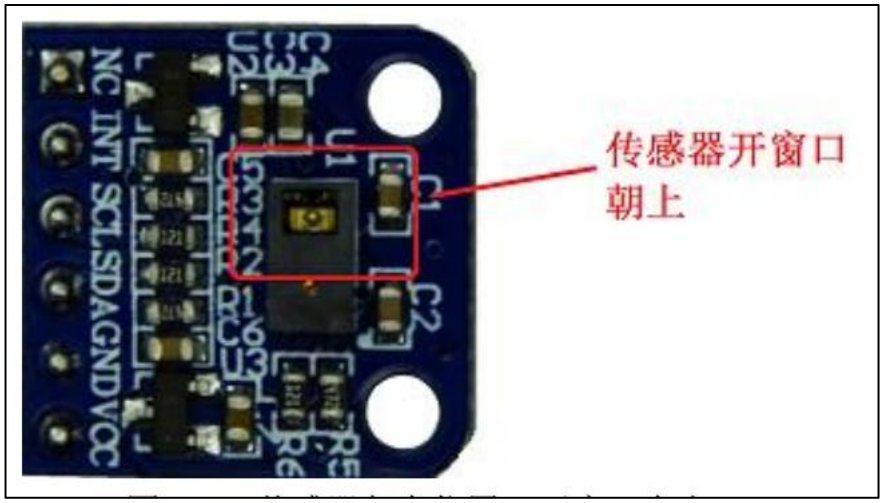

图 1.57 使用图示

将程序下载至开发板，向上挥手，效果如下：

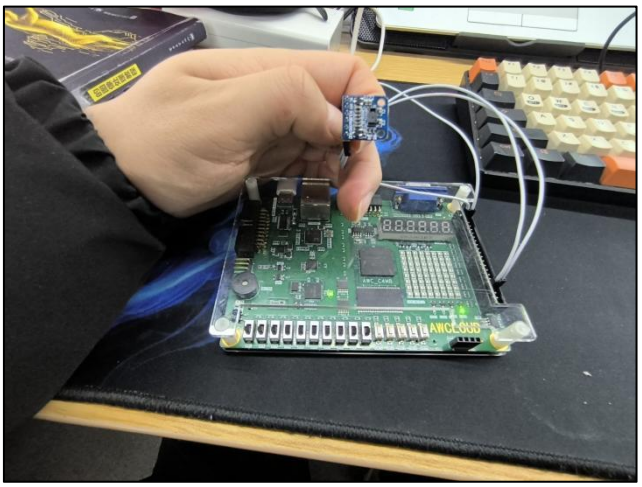

图 1.58 上板验证

如图 1.56 所示，第一个 LED 指示灯亮起，表示检测到向上挥手动作；向其余三个方向：下、左、右挥手，也有对应的指示灯亮起，上板验证成功。