第 5 章 SoC 芯片设计

SoC 架构的芯片是当代数字芯片的绝对主流，不论做什么领域的芯片，SoC 类型的芯片都是首选。因为它具备强大的灵活性，使芯片变成了人们可以自由操作的设备。目前的 SoC 生态建设已相当完备，统一的烧录器、编译器、编程工具、编程语法架构规范等，得到了广泛认同和应用，甚至于芯片设计工程师在完全不懂业务的情况下仍然可以设计出芯片，而嵌入式软件工程师可以在这颗芯片上实现各种各样复杂的功能。这便是 SoC 芯片把持其主导地位的主要原因。本章就来为读者揭晓 SoC 芯片的工作方式和设计方法。

5.1 SoC 架构

SoC 架构由处理器、总线及外围设备组成。处理器是其核心和大脑，它控制着芯片中全部的外围设备，因而称为中央处理器(Central Processing Unit，CPU)。总线相当于芯片的脊髓，而外围设备(Peripheral)相当于芯片的五脏。大脑通过脊髓及其延伸出来的神经细胞与五脏相连，同样地，CPU 通过总线与外设相连。连接的目的是一方面对外设的参数进行配置，即写设备；另一方面对外设的状态进行读取，即读设备。

CPU 常见的有 x86 架构、ARM 架构、51 单片机、RISC-V，其中 x86 一般不开放授权，ARM IP 开放授权，并得到了广泛采用，51 单片机适用于简单、低成本的芯片，而 RISC-V 一方面具备 51 单片机的开源免费、结构简单等优势，另一方面性能又远高于 51 单片机，目前已有部分产品从 ARM 改为 RISC-V。

总线类型常见的有 AXI、AHB、APB 等。

常见外设有只读存储器(ROM)、静态存储器(SRAM)、UART、I^2C、计数器(Timer)、脉冲宽度调制器(PWM)、Flash 控制器等。

5.2 关于 CPU 的一些概念

7min

在 SoC 架构中，CPU 一方面需要对外设进行读写，另一方面自己内部也要进行运算，例如四则运算、逻辑判断、移位操作等。只有将这两方面的操作相结合，芯片才能完成特定的

功能。CPU 执行什么操作、以何种顺序执行操作是由指令来控制的。CPU 按顺序读取指令，并按顺序执行。指令为一串二进制数，人们经常说的 CPU 的指令长度，即为这组二进制数的长度，比较常见的是 8 位、16 位、32 位、64 位。指令越长，能执行的操作越复杂。同样是一个动作，用 64 位指令的 CPU 需要一条指令即可完成，对于 8 位指令的 CPU 可能需要 8 条指令，假设每条指令执行的速度相同，则前者执行该任务耗时是后者的八分之一，但如果执行一个简单动作，只需一条 8 位指令，此时，64 位指令就显得体态臃肿。可见，对于简单应用，使用长指令会浪费存储空间，对于复杂应用，使用长指令却能提高执行效率。

CPU 还有指令集的概念，分为复杂指令集（Complex Instruction Set Computer，CISC）及精简指令集（Reduced Instruction Set Computer，RISC）。x86 属于复杂指令集，ARM、RISC-V 属于精简指令集。复杂指令集的指令数目多，而且单条指令可以完成的行为也比较复杂。精简指令集的指令数目相对少，而且单条指令可以完成的行为比较简单，一个复杂的动作需要拆分为若干个简单动作才能完成。复杂指令集中的指令没有固定的长度，而精简指令集都有着固定长度的指令。复杂指令集中的一条指令代表一系列复杂的动作，如果使用精简指令集表示相同动作，就需要多条指令。如果将复杂指令集中的一条指令比作一块巨石，则精简指令集中的一条指令就是一块小石子。遇到适合巨石安放的位置，一块巨石可以代替许多小石子，但合适的位置并不多，多数时候用小石子填到合适的位置是最方便的。人们为了增加巨石的适应力，制造了很多巨石，将这些巨石打磨成不同的形状，以便适应不同的安放位置，而小石子的种类却不需要很多，因为它们足够细小，找到适合它们的位置比较容易。这就是复杂指令集中指令的条目多，而精简指令集指令条目少的原因。人们之前之所以使用巨石的思路，是因为每块小石子的安装速度太慢，即一个任务执行的速度太慢，不如制造一块巨石直接安装上去效率高。随着芯片工艺的进步、速度的提高，巨石的不方便性和小石子的灵活性逐渐显现。实践证明，两种指令集都各有所长，所以 Intel 的 x86 架构中实际是把巨石在其内部也打碎成小石子进行流水处理的，而 ARM 指令集中小石子的种类也越来越多，石头越来越大，最终，两种指令集的概念将趋于模糊。由于 ARM 的体系结构已经发展得过于复杂，石头的种类多，大小不一，只有大型公司可以将其理解消化，还需要花钱购买，而 RISC-V 的规模还处在中小型公司可以维护的程度，也就是石头都还比较小，种类也比较少，因此得到了广泛关注。用芯片实现创新科技，有两种实现方法，一种是将创新放在外设上，作为 CPU 的一个设备，CPU 本身使用通用形态，另一种是将创新直接融入 CPU 内部，通过修改、增加指令集，使 CPU 在执行某些特殊功能时速度加快，RISC-V 为中小公司进行第二类创新创造了机会。

软件编译器负责将 C、C++ 等高级语言，或汇编语言等低级语言编译成二进制指令文件。它的作用如同一个翻译。复杂指令集的指令数量多，相当于一种词汇量比较多的语言，这里称为 A 语言，精简指令集相当于词汇量比较少的语言，这里称为 B 语言。编程人员以统一的语法形式（C、C++ 或汇编等）输入编译器中，由编译器分别翻译为 A 语言和 B 语言。可以看出，A 语言的翻译需要记忆的词汇量多，翻译难度较大，同时，每个单词的含义比较复杂，最终翻译的结果，其语句的数量较少，而 B 语言的翻译需要记忆的词汇量少，翻译难

度小,同时,每个单词的含义简单,需要很多单词才能表达清楚意思,最终翻译的结果语句的数量多。同样可以看出,A 语言翻译的门槛高,要求具有很高的水平,不同翻译之间的水平差距小,而 B 语言翻译的门槛低,但同时,不同的翻译,得到的结果质量差别也较大,有的言简意赅,有的则啰啰唆唆。上述比喻基本概括了两种指令集在编译器设计方面的特征。A 语言的翻译者主要是微软,它的 Visual Studio 将其他语言翻译为 A 语言的效率非常高。B 语言的翻译者,例如 Keil、gcc 等,水平各不相同,Keil 编译的结果从大小上和效率上都优于 gcc。虽然指令集的大类只有两种,但指令集细分也有多种,例如,同为精简指令集,ARM 和 RISC-V 的指令就不同,ARM 自己的众多产品中也有着不同的指令集,51 单片机也有自己的指令集,每种指令集都相当于一种语言,因而需要相应的编译器进行翻译。

CPU 内部识别指令有两种方式,一种是通过微码,另一种是直接硬件识别。微码识别即在 CPU 内部实现一个可编程的有限状态机,在这种状态机中,配置的比特不同,状态机的状态走向也不同,因而本质上就是一个较为复杂的有限状态机。微码由指令分解而来,配置到状态机中。复杂指令集多用这种方式。直接硬件识别在 Verilog 中就是简单的 if 语句或者 case 语句,判断指令是否为硬件中设定的二进制数,若是,则走对应的流程。精简指令集多用这种方式。

CPU 内部进行指令执行需要用到一些提供给编程者进行读写或运算操作的寄存器,例如 ARM Cortex M0 中就有 6 个这样的寄存器,它们与前文设计阶段使用的寄存器并无区别。通常说的 CPU 位数,例如 8 位 CPU、32 位 CPU 等,指的就是寄存器的位宽,而不是指令长度,因为对于复杂指令集来讲,指令长度是不定的,但是 Intel 或 AMD 的 CPU 属于 32 位或 64 位是明确的。寄存器位数越多,进行大数据计算就越容易,因为数据不需要截位,可以直接放在寄存器中进行计算,如果位数少,则当遇到大数据时为了防止溢出,需要将数据截断,进行分批计算,最后合并,一条计算指令就会变成多条指令。

CPU 的地址位宽决定了 CPU 能对多大的空间进行数据读写。这些空间未必有实际的存储器存在,它仅代表 CPU 的一种寻址能力。在设计 SoC 芯片时,挂接的外设及 CPU 内部的一些模块寄存器都会占用地址,这些地址并不连续,可以分开。那么,当 CPU 访问没有存储器实体的地址时,所取得的数据不就是错误的吗?确实如此,有两种方法用于处理这种情况,一种是在总线上挂接一个假设备,该假设备也是由 Verilog 编写的模块。凡是 CPU 寻到无效地址位置时,都会接入该假设备,由它反馈一个无效标志。另一种是在软件编程时,避免操作无效地址,将设备声明为结构体,编程时只操作这些结构体,而不是直接操作地址,有助于避免错误的发生。实际上,CPU 获得错误数据并不可怕,可怕的是将该数据用于计算,这是最应该避免的。CPU 的地址位宽和数据位宽(寄存器位宽)、指令长度并没有绑定关系,根据 CPU 寻址的需要,可以自由决定地址位宽。

CPU 的时钟频率,即其内部寄存器的驱动频率,决定了一条指令的执行速度。指令的执行速度常用每秒能执行几百万条指令(Million Instructions Per Second,MIPS)来衡量。很多 CPU 都是一拍时钟就执行完一条指令,但其实 CPU 并不是在一拍之内做了取指令、执行指令、保存结果等各种工作,而是通过流水线的方式(详见 2.22 节),在一拍之内完成了

第 1 条指令的执行、第 2 条指令的译码和第 3 条指令的读取，这就是 ARM Cortex M0 中的三级流水，最终用户看到的效果却是一拍执行了一条指令。高端的 CPU 由于具有很高的时钟频率，必须将上述三级流水再多分几级，否则就会因为组合逻辑太长导致建立时间不够。值得一提的是，MIPS 的衡量和测试标准是宏观的，因此，不能单纯地以 CPU 的时钟频率推导 MIPS，因为 CPU 在实际运行时，并不能做到每个时钟周期都能取出一条指令，指令存储在不同的位置，可能存储在 Flash 这类非易失性器件中，也可能存储在 SRAM 或 DDR 中，读取这些设备的速度是不同的。每一拍都能取到想要的指令是极其理想的假设，因为程序语言有很多不确定性，例如在 C 语言中，当遇到 if 和 else 语句时走哪个分支？while 循环在什么情况下跳出？指令是按顺序存的，但程序语言的这种不确定性会导致取指令不按顺序取。由此，CPU 的设计者引入了缓存(Cache)，它的设计思路是先预测下面的几条可能被执行的指令，在 CPU 没有读取之前，先从速度慢的存储介质(如 DDR)中读到速度快的存储介质中(如 SRAM)，当 CPU 读取的指令已经在 Cache 中了，称为命中，可以在一个时钟周期内将指令送达 CPU，若不在，则没有命中，再从原来指令存储的位置进行读取。SRAM 虽然读写快，但是面积大，因此不能承担很多存储任务，需要分层次地对预测的指令进行存储，使用最快的 SRAM 作为第一级 Cache，其后级的 Cache 使用面积更小但速度较慢的存储介质，常见的高性能 CPU 均有多达 3 级 Cache，而低性能 CPU 可能没有 Cache。可见，Cache 设计的关键因素在于预测，这是对程序行为，或称对人的行为的预测，目前这个领域仍然吸引了大量的学者从事研究，并没有一个绝对完美的解决方案。

值得一提的是，一个程序或一项任务的执行效率，并不单纯取决于 MIPS 指标，还有它的指令集。若 MIPS 高，但编译出来指令条数多，与 MIPS 低，而指令条数少，最终的执行效果是一致的。

如果一个性能最佳的 CPU 都无法满足应用需求，则需要考虑两个及以上的 CPU 协同配合来完成应用任务。例如手机芯片中通常的做法是使用几个高性能的 CPU 核运行游戏，而低性能核用于普通的应用，原因是高性能核心的功耗高，用它处理一般应用需求会缩短待机时间。多核并行处理，可想象为一套硬件被复制为两份，并将一个任务分成两半，由两套硬件各自处理其中的一半，最终达到缩短任务执行时间的目的。单核 CPU 是按照顺序处理指令的，虽然它也可以处理并行的线程和进程，但这个所谓的并行其实是由操作系统调度产生的幻觉，实际执行仍然是串行的。单核变成多核后，真正从硬件上做到了并行，需要考虑的就是如何分工的问题。ARM 在设计 CPU 和总线时，就已经考虑到该问题，把 CPU 之间通信的接口和协议规定齐备，可以避免 CPU 之间执行指令的冲突。

5.3 简单 SoC 结构及存储器类型

不论是复杂的多核 SoC，还是简单的单核 SoC 结构，都存在设计上的共性，本节以一个简单的 SoC 结构为例，来对该共性进行阐释。

一个简单的 SoC 结构如图 5-1 所示。整个架构的核心仍然是 CPU，它的总线上有几个

必要的设备,分别为 ROM、RAM 和 NVM 控制器。这 4 个设备在任何 SoC 芯片中都是必要的,其中,ROM 负责存储芯片上电时 CPU 应该执行的指令,一般称 ROM 上存储的程序为 Boot 或 Bootrom。RAM 用于存储主要的业务指令,并且在运行过程中暂存中间数据。NVM 控制器是非易失性器件的控制器。RAM 在芯片掉电后其内容就会丢失,而 ROM 在掉电后内容还在,但一般只存储上电程序,难堪大任,因而芯片中一般需要非易失性器件(Non-Volatile Memory,NVM)负责在芯片掉电后保持对指令的记忆。常见的非易失性器件如 Flash、MTP、OTP、EEPROM 等。这些器件一般需要特殊设计的控制器才能读写,因而在总线上挂的常常是控制器,在控制器的后面挂 NVM 本体。有时,NVM 并未集成在片内,即 Die 内,而是作为另一个单独的 Die,可以同 SoC 芯片一起焊接在电路板上,也可以将两个不同的 Die 放在一个封装里,作为一颗芯片。

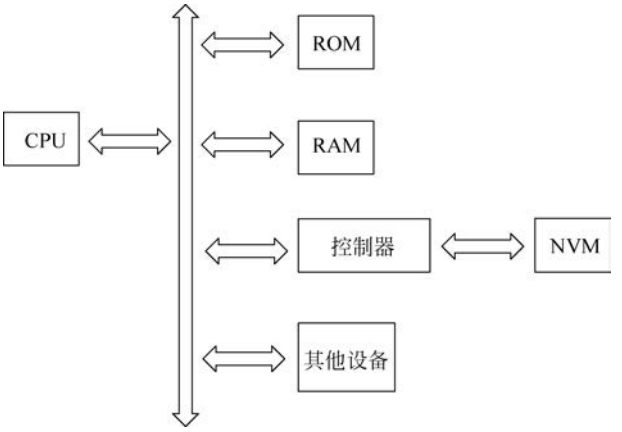


图 5-1 简单 SoC 结构

可见,一颗 SoC 芯片其实是一台微缩的计算机,有 CPU、有 Boot、有内存条(RAM)、有硬盘(NVM),最后由总线和通用输入/输出口(GPIO)设备充当主板功能。

RAM 分为 SRAM 和 DRAM,其中 SRAM 是静态随机存储器,它结构简单操作方便但是面积大,DRAM 是动态随机存储器,它需要周期性地刷新才能保持记忆,操作相对复杂。高速的 DRAM 都使用同步技术,称为 SDRAM,在计算机内存条中使用的芯片是 DDR,它也是一种 SDRAM,但与其他 SDRAM 的区别在于它是双沿采样的,因此速率提高了一倍。设计 SoC 芯片时,一般会放入一块或大或小的 SRAM,作为数据、堆栈的存储器使用,而 DRAM 类型却未必会用到,仅当数据存储需求很大,完全使用 SRAM 成本过高时才使用。下例即是一个 SRAM 的例化实例,包括时钟 CLK、片选信号 CEN(低有效)、写使能信号 WEN(每个比特有独立的使能,低有效)、地址线 A、输入数据线 D、输出数据线 Q 等。当 CSN 为低时,选中该 SRAM,此时观察 WEN,若其中某些比特值为 0,则将 D 中相应比特写入 A 所指向的地址,在写的同时,Q 端也会在一到二拍内,将 A 地址上原有的数据输出到接口上。本例中 WEN、D 和 Q 都是 32 位的,用户的程序,如 C 语言中的 int、char、short,可以体现在 WEN 上,譬如要对 4 字节中的某个进行写操作,则其他 3 字节的使能都为 1,只有需

要改写的字节使能为 0。信号 addr 的位宽根据 SRAM 的具体容量来定，此例中 addr 的位宽为 20 位，可知该 SRAM 的容量为 2～4MB。

```
sram      u_ram
(
    .CLK  (clk          ),//i
    .CEN  (csn          ),//i
    .WEN  (wr_en_n      ),//i[31:0]
    .A    (addr         ),//i[19:0]
    .D    (wdat         ),//i[31:0]
    .Q    (rdat         ) //o[31:0]
);
```

SRAM 一般由工具生成，工具由 Foundry 或 IP 供应商提供。ARM 公司提供的一个 SRAM 生成工具界面如图 5-2 所示，可以设定 SRAM 的字数量、字长、宽度形状、所属的金属层等。字长设定为与 CPU 数据总线长度相同，字的数量根据应用需要而定，至于其他物理量，如 Top 层、纵线、横线分别应分配到哪个金属层，需要听取后端工程师的建议，以及将整个芯片的设计层次综合考虑，主要是找到使用率较低的层次实现 SRAM。Multiplexer Width 决定了最终生成的 SRAM 的形状，生成时建议将这些选项都试一遍，选出生成形状最方正、面积最小的来使用。工具最终可以生成 gds2 和 lef 文件，供后端布局布线使用，cdl 网表文件用于 LVS 一致性对照，以及 Verilog 文件用于前仿和后仿。

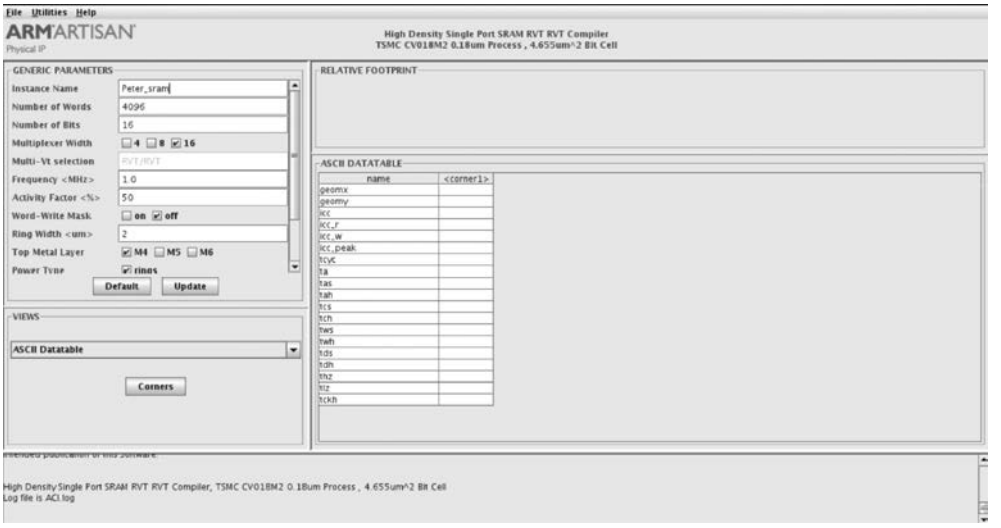


图 5-2 SRAM 生成工具界面

在一个设计中，常常会遇到多个模块需要 SRAM 的情况，但每块 SRAM 容量需求都较少。这时，主要有两种解决方案，一种是按照需求，生成多个容量较小的 SRAM，零散地分布在芯片中，另一种是将它们集中在一起，形成一个大的 SRAM。在工程中，往往使用第 2 种方案，在总容量相同的情况下，一块 SRAM 比分散的多块 SRAM 面积小。集中生成的问

题在于,分散的 SRAM 每个都有自己独立的读写接口,而一块 SRAM 只有一个读写接口,因此需要编写一个专门的模块,称为仲裁器(Arbiter),用来协调各模块读写 SRAM 的需求,这也增加了设计的难度。

ROM 是只读存储器,CPU 无法向其内部写入内容,只能读取内容。ROM 中的内容,即 Boot,由软件人员编写完成后,生成二进制文件。用 ROM 的生成工具读取该文件,最后生成一个内容无法修改的 ROM 硬件。在 ROM 内部,二进制文件直接与电平相关,出现比特 1 时硬件上就连到数字电压 VDD 上,出现比特 0 时硬件上就连到 GND 上,因而 ROM 的结构十分简单。在芯片制造中,ROM 的硬件连线总是集中在少数的几个金属层上,这为修改 ROM 的内容提供了方便。虽然当需要修改内容时仍然要重新流片,但只需改动涉及 ROM 的少数掩模,修改的成本较低。

一个 ROM 的例化接口如下例所示,与 SRAM 相同的是,它也有 CLK、CEN、A 和 Q,但没有写数据和写使能的输入,因为它是只读器件。此例中给出的地址是 9 位,说明 ROM 的大小在 1～2KB 内。

```
rom      u_rom
(
    .CLK (clk      ), //i
    .CEN (csn      ), //i
    .A   (addr     ), //i[8:0]
    .Q   (rdat     )  //o[31:0]
);
```

ROM 的生成工具界面如图 5-3 所示。与 SRAM 的生成一样,仍然可以设定字长、字数及形状。Boot 文件名填写在 Testcode File 中。

Boot 文件常以.rcf 为扩展名,每行放一个字,逐行排列,一个字长为 32 比特的.rcf 文件片段如下。常用的嵌入式软件编译工具,如 Keil,编译后会生成.bin 和.hex 两种类型的文件,前者是二进制文件,由于是直接用 0 和 1 表示的,不是用 ASCII 码表示,所以用文本编辑器打开后显示为乱码,后者是 ASCII 码表示的十六进制文件,它们都不是.rcf 的样式,需要使用脚本工具将其转换为例子中的格式才能使用。

```
00000001000000000000000110111101
00000001000000000000000110111111
00000001000000000000000110111111
00000001000000000000000110111111
00000001000000000000000110111111
00000001000000000000000110111111
00000001000000000000000110111111
00000001000000000000000110111111
00000001000000000000000110111111
```

在 NVM 存储器中,Flash 最为人们所熟知,它的存储容量很大,可擦除也可读取。虽然擦除速度较慢,但读取速度很快。单次编程器件(One-Time Programmable,OTP)是一种

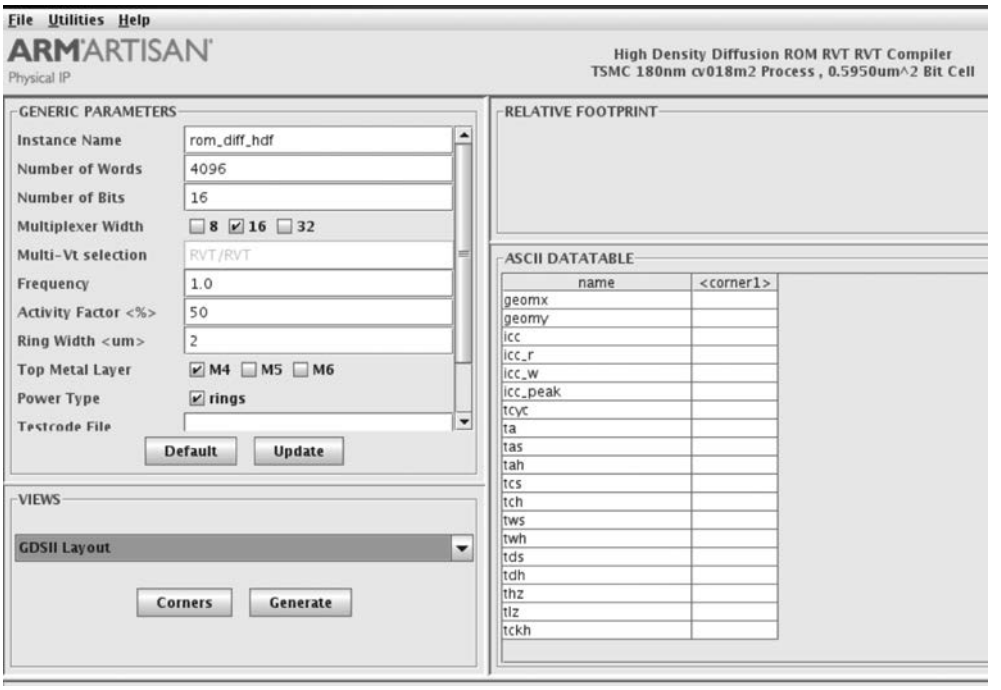


图 5-3 ROM 生成工具界面

只允许一次编程的记忆器件，若第 1 次烧写的程序有错，则无法修改，芯片就无法销售，因此它只适合于产品定位清晰，代码已经被检验为完全正确的场景。在过去的低成本芯片中，OTP 经常使用，但随着用户的需求越来越复杂化、多样化，OTP 已很难满足用户的需要，正逐步被多次编程器件（Multiple-Time Programmable，MTP）所取代，后者是一种可以多次擦写的记忆器件。对于用户而言，Flash 与 MTP 在使用上并无差别，只是 Flash 更容易使用更小的面积实现更大的容量，擦写和烧录的次数也更多。对于设计者而言，两者的工艺不同，控制方式也不同。MTP 往往可以在普通的流片工艺下构建，而 Flash 的构建则需要特殊高压工艺，这样为了使用 Flash，整个芯片的其他电路也都必须搬移到该工艺上，这种工艺上的搬移也被称为转厂，其对于数字电路的影响较小，但对于模拟电路的影响较大，其原有设计、仿真结论需要重新审核，甚至要重新设计。若既不想转厂，又想使用 Flash 作为 NVM，则可以选用片外 Flash，这样，SoC 芯片本身使用普通工艺进行流片，形成一个 Die，Flash 只用专属工艺，形成另外一个 Die。两个 Die 可以各自封装，并焊接在电路板上，也可以封装在一起，作为一颗芯片进行销售。若使用同一个封装，则需要选购大小合适的 Flash Die，不同 Flash 厂商的实现工艺不同，大小不一，另外，引脚的排布也不同，因此，连线的难易程度、是否能被一个合适的封装所容纳，是需要仔细考虑的问题。分离的 Flash 大多采用高速 SPI 接口，因此 SoC 芯片需要设计一个 SPI 主设备控制器，挂接在总线上，以便 CPU 随时通过它读取 Flash 中的内容。Flash 分为 NOR 和 NAND 两类，NAND 是主流类型，它具有成本低、擦写速度快、容量大等优点，但若将它作为 NVM，运行程序时必须先将整个程

序搬移到 RAM 中才能运行,因为它没有单字节寻址的能力,只能使用页寻址,而 NOR Flash 虽然成本较高、擦写速度慢,但读速度快,最主要的是,它可以随机存取,不必一读就是一整页,因而当它作为 NVM 时,可以使用 XIP 技术直接在 Flash 中取指运行,不必复制到 RAM 中。MTP 在实现相同存储容量的情况下面积较大,但它对工艺的要求较低,因而有一部分芯片用它作为 NVM 记忆体。在一些芯片实现中,使用 ROM 作为 NVM,这样实现确实最为简单,但代价是明显的。ROM 的程序无法轻易修改,这就要求厂商必须保证其交付流片时使用的程序即为产品使用的最终程序,而这一要求是很难达到的,同时,这样做也失去了 SoC 架构固有的灵活性优势,把产品降格为固化的 ASIC 形式,舍本而逐末,诚不可取也。

注意 本书中程序、指令等名词混用,目的是让初识 SoC 芯片的读者有一个清晰直观的认识。严格地说,程序即为流程及步骤,该流程被编译为若干条指令,因而可称程序为完成规定步骤的指令集合。不可简单称为指令集,因为指令集是指一款 CPU 所支持的全部指令的集合。单独说指令,无法准确表示是全部的指令还是单条指令,因而本书中多使用程序一词表示需要 CPU 执行的任务。

5.4 SoC 芯片中常用的外围设备

上述 ROM、RAM、NVM 及其控制器是 SoC 运行必不可少的设备,如果缺少了它们,则 SoC 无法运行。其他设备虽然并非 SoC 运行的必要条件,但可以帮助 SoC 完成特定的任务。本节介绍一些常见的外设,它们在大多数 SoC 芯片中会用到。

(1) UART 可以使芯片与计算机等设备连接,通过计算机的串口工具与芯片进行通信。连接一般有两根线,分别为 Tx 和 Rx,前者是信号发送线,后者是信号接收线,没有统一的时钟线。UART 是没有主从关系的,即在连接中,没有主设备(Master)和从设备(Slave)之分,通信双方的地位是平等的,因而双方各有一套 Tx 和 Rx,其中一方的 Tx 接另一方的 Rx。实际上,一个固定的通信方向上只用了一根通信线,即所谓 1-wire 方式。市场上很多 SoC 芯片标称的 1-wire 接口,实则是 UART 接口的变体。UART 的通信机制简单,靠固定频率的电平直接表示比特,其频率常用波特率表示,在一字节发送中有起始位、校验位和结束位等,用以标识字节的边界。由于没有时钟,无法具备同步电路快速传输的优势,只能异步传输,即通信双方各自在内部产生并使用自己的时钟,这两个时钟之间频率相等,但没有相位关系,属于异步时钟。要做到时钟频率相等,则收发两端都要配置相同的频率,但由于时钟产生精度的问题,双方的实际频率可能有所差别,偶尔在传输中出现的乱码即是由该原因造成的。例如使用芯片的 32MHz 晶振产生波特率为 460 800 的 UART 时钟,须将 32MHz 分频 69.444 倍,实际只分频整数倍,即 69 倍,因此实际产生的 UART 时钟波特率为 463 768,这种频率偏差对于短数据通信的 UART 来讲是允许的,而如果对于大数据量 Burst 通信,例如 WiFi,则必须经过采样频偏校准。通常情况下,UART 传输频率只能达到

几百 kHz，用以满足最基本的读写要求。计算机与芯片的 UART 之间一般需要电平转换，芯片电平标准一般是 3.3V，而计算机串口会输出多种电平，如 12V 等。

(2) 串行设备接口(Serial Peripheral Interface，SPI)是一种常用的低速通信接口，用于芯片间通信。它具有主从关系，由 Master 发出时钟，Slave 与其同步，并且 Master 还会发出接口操作命令，即操作码(Operation Code，Opcode)，用来表示传输的目的是读还是写。最简单的 SPI 接口由 SCLK、CSn、MOSI、MISO 4 根线组成，其中，SCLK 是时钟线，CSn 是片选信号，MOSI 是 Master 输出给 Slave 的单向信号，MISO 是 Slave 输出给 Master 的单向信号。一个 Master 可以挂接多个 Slave，其中 SCLK、MOSI、MISO 3 根信号线是共享的，可以认为是总线，只有 CSn 是每个 Slave 拥有的独立信号。默认情况下，全部的 CSn 信号线均为高电平，当 Master 想跟其中一个 Slave 进行通信时，会拉低该 Slave 的 CSn，而其他 Slave 在此过程中保持静默。SCLK、MOSI、MISO 等 3 根信号线在默认情况下也通常为高电平，但并非强制要求。由于 SPI 使用同步通信机制，通信速率较快，其时钟通常可达到 50MHz 以上，100MHz 左右的也甚为常见。为了进一步提高通信速率或提高通信线的利用率，出现了很多 SPI 变体，如一根数据线既可以作为 MOSI，又可以作为 MISO，由单向线变为双向线，或者两根、4 根双向线同时通信。当 Master 向 Slave 发出信号时，由 4 根信号线同时传输，而当 Slave 向 Master 回复信号时，刚才的 4 根线其传输方向发生翻转，仍然同时传输数据。一般将两根信号线同时传输数据的方式称为 DSPI，将 4 根线同时传输数据的方式称为 QSPI。

(3) I^2C，也称 IIC，也是一种常用的芯片间通信接口，其特征已包含在其名字中，即两根线进行的通信，分别称 SCL 和 SDA。SCL 是时钟线，SDA 是数据线。由于存在时钟线，可知此设备也是有主从之分的，Master 是时钟的发出者，Slave 接收时钟，并在该时钟的驱动下采集 Master 的数据或向 Master 发出数据。I^2C 的接口简单，SCL 一般为单向线，即从 Master 指向 Slave。SDA 一般为双向线，不论 Master 还是 Slave，只要发出数据均由它负责传输。有些接口设计将 SCL 设计为双向线，它给 Slave 一个控制 SCL 的机会，可以将 SCL 下拉到低电平，以便通知 Master“Slave 尚未准备就绪”，不能立即传输。这种 SCL 需要 Master 和 Slave 都具备相应的处理机制才能发挥作用。值得注意的是，与其他芯片引脚不同，I^2C 接口的引脚一般是开漏的(Open Drain，OD)，即信号从 MOS 管的漏极直接输出。这种 OD 引脚的特点是只有输出低电平的能力，没有输出高电平的能力，要想输出高，只能接上拉电阻，因此，若看到 OD 引脚上的电平为高，此高电平不是引脚输出的结果，而是引脚处于高阻态，由上拉电阻输出的，只有引脚上的电平为低时，才是它真正的输出。使用 OD 的特性可以在一个 I^2C 接口上挂载一个 Master 和若干个 Slave，当 Master 将 SCL 和 SDA 的电平拉低时，接口上的所有设备都能够感受到，当 Master 所传输的 ID 是其中的一个 Slave 的，则由该 Slave 辨别出来，并响应，而其他 Slave 设备也会辨别出来并保持沉默，直到该通信结束。若使用普通引脚，则当线路上同时有两个设备，一个输出高电平，另一个输出低电平时，该线路将变为不定态。常用的 I^2C 时钟速率是 100kHz 和 400kHz，也有一些 I^2C 接口设计时钟频超过 1MHz。

(4) Timer,即计时器,负责计时,到达预定时刻后,会发出中断,通知 CPU 做后续处理。一颗 SoC 芯片往往会挂载多个 Timer,以满足同时计时的需要。例如,要实现某协议,需要先对事件 A 计时,在 A 计时尚未结束时,又需要对事件 B 计时。这就需要有两个硬件 Timer。在实际中,复杂的协议需要更多的 Timer。若使用软件计时,则计时的精准度是不够的,因为软件需要处理各种不同的任务,除顺序处理的任务外,还有不时而至的中断需要响应,因而纯软件计时经常会被其他任务打断,导致计时不准确,在需要严格定时的场景下,就要用到硬件 Timer。Timer 的本质是计数器,计数可以从 0 递增到预定值,也可以从预定值递减到 0。有些计数器可以设置先从 0 递增到预定值,然后递减到 0。开启 Timer 的目的主要是为了到达某个时刻能够向 CPU 发起中断,中断发起的位置可以与 Timer 重载时刻一致,也可以任意选取中间时刻。计时可以是不需要 CPU 操作就可重复进行的周期性计时,也可以是一次性计时,如停止后不能自动发起的 Oneshot 计时(详见第 4.1 节)。SoC 芯片中的若干 Timer,有着不同的计时精度要求和用途。普通 Timer 使用与 CPU 相同的时钟作为驱动,CPU 的时钟可以来自不同的源头,而 Timer 并不关心,它只是单纯使用 CPU 时钟,这样设计的好处在于简单方便,缺点是当 CPU 发生频率变化时,例如休眠时,若要记录与原来相同的时间,则 Timer 的设置必须改变,有时 CPU 进入深度休眠,时钟完全被切断,则 Timer 也无法使用。为了弥补普通 Timer 的缺点,SoC 芯片中还会集成实时 Timer,它的时钟源头是固定的(例如内部振荡器或晶振),不随 CPU 时钟的变化而变化,这样,它就可以按照固定的频率持续计时,其内部比普通 Timer 复杂,因为需要异步握手,使 CPU 的配置同步到该时钟源。实时 Timer 可以作为万年历使用,不论芯片是否休眠,它都一直在工作,对于具有操作系统(Operation System,OS)的芯片来讲是很有必要的。Timer 还可能会集成输入脉冲宽度统计功能,从芯片外输入的电平信号,进入 Timer,对它的高低电平宽度进行统计,并在电平切换时发出中断信号。

(5) 脉冲宽度调制器(Pulse Width Modulation,PWM)。可以产生宽范围频率的方波,占空比(Duty)可以设定。现实中很多应用都需要用到方波,如无线充电、电机驱动等,因而 SoC 芯片特别是通用 MCU 也常常会挂载 PWM 发生器。有些设备需要多路 PWM 共同驱动,这些 PWM 不是异步的,而是需要有特定的相位关系,因此,PWM 发生器往往会集成多路 PWM 逻辑,支持多路 PWM 同时发出。最典型的多路 PWM 应用是一对 PWM 以相位翻转的形式发出,即第 1 路 PWM 为高时,第 2 路 PWM 为低,反之亦然,这种 PWM 常被用于驱动模拟大功率 MOS 桥电路。除了周期和占空比,PWM 还有一个特征称为死区时间(Deadzone),它的作用是在多路 PWM 同步输出时,可以任意设定这些 PWM 的相位,使其满足驱动要求,如果仅有一路 PWM,则死区时间没有意义。仍然以驱动 MOS 桥电路为例,用于驱动的两路 PWM 不应该完全反向,因为 MOS 桥有一定的开关时间延迟,即使在数字上完全反向,在模拟上也并不能立即对 MOS 进行开关切换,那么在两路 MOS 管上就有机会同时导通,从而发生短路故障甚至烧毁芯片,因此需要将两路 PWM 的上升沿稍稍延后,让下降沿先出现,上升沿再出现,中间形成时间差,使两个 MOS 管在开关切换过程中有一段时间是同时断开的,从而避免短路风险。PWM 本质上仍然是一个计数器,内部计数到一

些数值时输出高，计数到另一些数值时则输出低。一般来讲，有几路 PWM，其内部就应该有几个计数器，但若是多路同步的 PWM，应该共用一个计数器，以便保持同步。由于内部结构与 Timer 类似，在 SoC 设计中，常在 Timer 中加入 PWM 功能，即 Timer 既可以计时，又可以用于产生 PWM，而且一个 Timer 可产生一对同步的 PWM。常用的 MCU，例如 STM32 系列，在其说明书中常常有输出一对互补 PWM 的功能，所谓互补就是指上述翻转 180°的 PWM。

(6) 看门狗(WatchDog，WDT)，也是一种 Timer，它的主要用途是当 CPU 程序运行错误后死机时进行自动复位。在程序正常运行时，隔一段时间就需要对看门狗的计数器进行一次重载，即重新配置。一旦停止重载，当看门狗内部计数器倒计时到 0 时就会发起复位操作，即通过一根复位信号线引发总线复位，此时，凡是使用总线复位线作为自身复位信号的触发器都将被复位，复位释放后，程序将从 Boot 开始重新运行。由于看门狗担负着维持系统稳定的责任，其驱动时钟一般使用固定的时钟源，不与 CPU 时钟混用。

(7) 通用输入/输出口(General-Purpose Input/Output，GPIO)，不仅是一个接口，而且是一个设备，它具备诸多功能。不仅能读到它的输入值，以及按照要求输出高低电平，还具备多种中断功能，例如针对输入信号的上升沿、下降沿、高电平、低电平等状态产生的中断。需要注意的是，芯片中有两种不同的 GPIO 概念，其中一个是指本节提到的这类设备，另一个是芯片的通用引脚。作为通用引脚的 GPIO，实际上是 I/O 器件的一部分，如图 1-3 所示。需要严格区分这两个概念的原因是，作为引脚的 GPIO，并不单纯只连接 GPIO 设备，由于封装尺寸小型化，引脚资源紧张，一个引脚往往会连接多种设备进行功能复用。因而，一个 GPIO 引脚，既可以作为 GPIO 设备的输入/输出引脚，也可以成为 UART 或 I^2C 设备的接口，如图 5-4 所示。

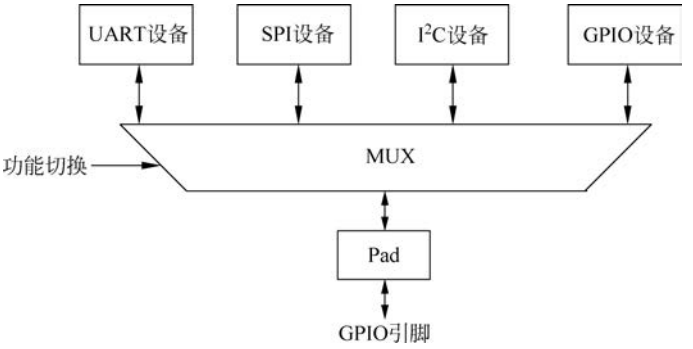


图 5-4 GPIO 引脚及其所接设备

5.5 SoC 内部程序的运行过程

17min

简单的 SoC 芯片需要运行两个程序，先运行上电启动程序，即 Boot，它存放在 ROM 中，然后启动主要业务程序，这里称为 App。

Boot 程序是由软件人员编写并固化在 ROM 中的程序,它的主要作用是引导 App 顺利执行。在编写 Boot 时,须涉及如下 3 个方面:

(1) 对芯片组中的其他芯片进行复位。

(2) 检查 NVM 是否在预定位置上。

(3) 如何进入烧写模式,如何对 NVM 进行烧写。

上述第一点中的芯片组是指单独一颗芯片或一个 Die 无法独立完成任务,需要两颗或以上的 Die 共同完成任务的情况下,任务涉及的若干芯片就组成了一个芯片组。当该芯片组中的主控芯片执行 Boot 时,需要对组中其他芯片也进行复位操作。这样做的原因是 Boot 只在两种情况下被执行,一种是在上电复位时,另一种是在意外复位时。上电复位的时刻,即电路板刚刚通电的时刻。此时,芯片组中的所有芯片都是刚上电,因此它们一起复位,是否在主控芯片的 Boot 中执行复位操作其实并无区别,但如果主控芯片由于意外情况而复位,则不能允许仅仅是主控芯片发生复位,而其他芯片保持原有的工作状态继续工作,因为原有的工作状态很可能对于最初的工作并不合适,例如,某个芯片已经进入了一种高速处理模式,而上电时要求它先进入低速处理模式,根据主控芯片的指挥才能进入高速模式,当主控芯片复位后,该芯片仍保持高速模式就不符合上电规程的要求了,需要将其复位到低速模式。主控芯片发生意外复位的原因有很多,例如程序指向了一个无效地址,取回了一条错误的假指令,就是俗称的跑飞,芯片的电压、电流、温度超出或低于设计范围,导致取指和执行不正常等,这些原因会导致 CPU 对程序的执行发生错误。为了使芯片在发生上述意外情况后能自行复位,一般会在总线上挂一个看门狗电路,它的功能就是当程序执行错误时对总线发起复位。如何判断程序执行错误呢?看门狗的机制是程序必须周期性地对看门狗电路进行配置,俗称喂狗。不论程序的主要目的是执行何种功能,周期性地喂狗都是必要的,当 CPU 停止喂狗时,会导致喂狗超时,看门狗电路就会判断有意外发生,从而复位总线及总线上的设备,Boot 重新执行。除看门狗会引发复位外,CPU 本身还可能出现 LockUp 状态,它是一个信号,当 CPU 正在处理 NMI(不可屏蔽中断)中断服务程序时发生错误,或者在硬件错误(HardFault)处理中又再次发生了错误时,该信号会置 1,此时,在 SoC 设计时往往也用 LockUp 信号线作为复位线,将 CPU 复位。此外,通过软件配置寄存器的方式也可以实现总线手动复位,例如 ARM 的 Cortex M0 处理器中,SCB 寄存器组的 AIRCR 字段,若将其第 16 到第 31 位由默认值为 0xfa05 改为 0x05fa,并将第 2 位置为 1,就会发起总线复位。上述 3 种复位情况只要其中一种发生,就可以实现复位,Boot 将重新运行,总线上所挂的外设中多数寄存器都将被复位为初始态,同时,由于 Boot 中复位芯片组的动作,也会导致芯片组内的其他芯片复位。对其他芯片进行复位的方法,需要根据实际情况而定,例如,有些 Flash 是 SPI 接口,对它发出某些特殊的 SPI 命令就能完成 Flash 复位。

第二点是要求检查 NVM 是否在位置上,原因是某些芯片,其 NVM 记忆体可能呈现多种形态,可能是 Die 内集成的,也可能在 Die 外。若是在 Die 外,例如需要焊接在电路板上,则可能在芯片上电运行 Boot 时没有焊接,此时芯片无法运行 App,因为 App 的程序存储在 NVM 中,因此,需要 Boot 对是否存在 NVM 做检查。一般,Boot 会从 NVM 的一个特定位

置读取一串特定数据，若数据与预期一致，则说明 NVM 存在。有两种情况可能导致检查失败，第 1 种是 NVM 不存在，第 2 种是 NVM 存在，但它是未经烧写的，其内部的所有比特都是初始值。

第三点是要在硬件上确定如何进入 NVM 烧写模式，以及烧写模式下 NVM 怎样烧写。这里的烧写模式，仅指批量生产芯片时对 NVM 进行烧写的方式。研发芯片的最终目的是售卖，而芯片一经导入供应链，其需求量巨大，一般以每月百万片为计量单位，也称为 KK，即 10 的 6 次方。这就要求芯片厂商具备快速进行量产测试的能力，而将 App 烧写到 NVM 中也是量产测试中的一个环节。在生产时，测试机台会发出规定信号，该信号可以是 App 的二进制数据流，按照 SoC 芯片规定的烧写格式，灌入芯片引脚中。数据流可以是符合 NVM 烧写标准的，直接进入 NVM，其他电路只起通道作用。也可以在芯片内部进行翻译，例如数据流通过串口或 1-wire 方式，只用一根线输入，再在芯片内部转换为符合 NVM 标准的烧写格式。对于使用 SPI 接口的 Die 外 Flash，一般会将 GPIO 引脚复用为 SPI 接口。测试机台产生 SPI 信号，顺着 GPIO 输入芯片内部，在片内仅提供通道作用，信号直接到达 Flash，从而完成烧写。这种方式适合 Flash 与 SoC 属于不同的 Die，却合封在一起的情况，若是分开封装的，则烧写 NVM 的工作会单独交由 Flash 测试流程完成，与 SoC 无关。当烧写方式确定后，需要在 Boot 中创建一种条件，当条件满足时，Boot 引导芯片进入量产烧写模式，否则 Boot 会引导芯片进入 App 继续运行。一种简单的进入烧写模式的条件是将芯片中的一个 GPIO 引脚在芯片内部下拉，则当它在电路板上悬空时，CPU 读该 GPIO 的输入值仍然为 0。在 Boot 运行时，检查该 GPIO 引脚的输入值，当其值为 0 时，说明当时用户并未对该 GPIO 进行特殊处置，是正常模式，继续进入 App 运行，若读出其值为 1，则说明用户在电路板上将该 GPIO 接到了电源上，进而说明用户打算烧写 NVM，此时，Boot 会对烧写涉及的 GPIO 引脚进行功能切换，转换为测试机台数据能够灌入的模式。这里需要注意的是，NVM 的寻址方式都比较复杂，因而都带有控制器，CPU 不直接与 NVM 发生关系，而是通过控制器传达。当 GPIO 引脚与 NVM 连通时，就需要屏蔽掉控制器的作用，否则系统设计会出现多重驱动的风险。一个简单的切换电路设计结构如图 5-5 所示，图中 NVM 与

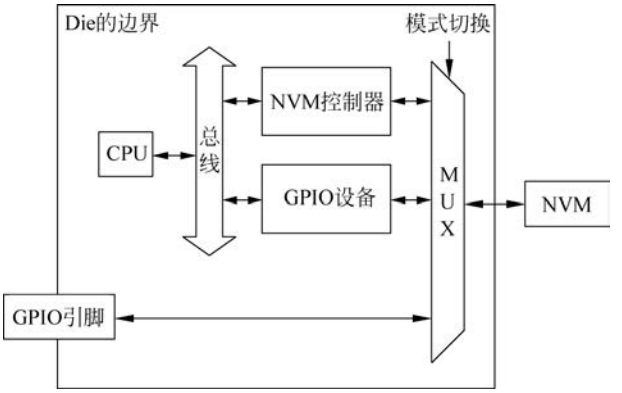


图 5-5 NVM 烧写模式和普通模式的切换

SoC 芯片本身是分开流片的,不在同一片 Die 上。当芯片正常运行业务时,NVM 控制器与 NVM 相连,当进入量产烧写模式后,GPIO 引脚与 NVM 直接联通,使量产测试机台能构造出烧写用的数据流通过 GPIO 引脚灌入 NVM。图中还提供了 GPIO 设备与 NVM 的连接,使 CPU 可以通过对 GPIO 设备的编程来操作 NVM 的读写。

综上,Boot 程序的基本流程如图 5-6 所示。

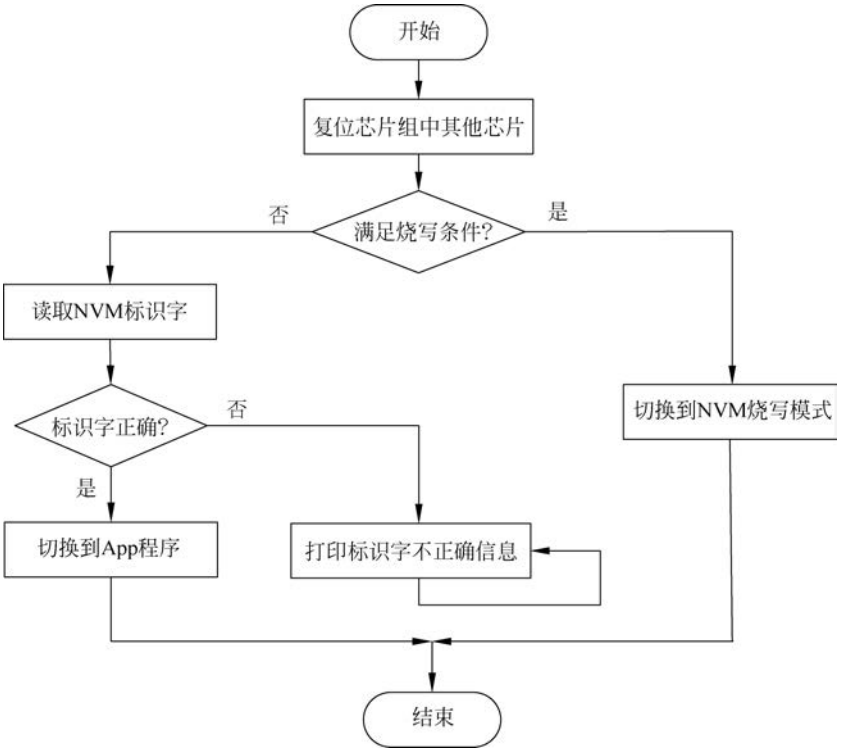


图 5-6 Boot 程序的基本流程

从 Boot 切换到 App 的过程,有 3 种实现方式,对于 App 来讲是 3 种运行方式。

(1) 将 App 从 NVM 中复制到 RAM 中运行。

(2) App 在片内 NVM 中运行。

(3) App 在片外 NVM 中运行(XIP 方式)。

第 1 种方式的切换步骤是由 Boot 程序指挥将 NVM 中存储的 App 程序复制到片内 RAM 中,然后将默认运行地址从 ROM 改为 RAM。这种方式的优点是操作简单、App 运行速度快、流片工艺适应性强等,缺点是需要较大的 RAM 空间,必须能容纳 App 的全部程序,对于功能较为复杂、程序庞大的应用来讲,采用这种方法成本非常高。在小型 SoC 芯片中,应用程序较少,可使用此方法,为进一步简化设计,RAM 使用的是 SRAM,如第 5.3 节所述,它相对于 DDR 来讲操作更为简单,但面积更大。

第 2 种方式针对 Die 内集成 NVM 的芯片。若用此方式,则 Boot 程序会指挥将默认运

行地址从 ROM 改为 Die 内 NVM。这种方式的优点是节省 RAM 资源、App 运行速度也比较快，缺点是需要特定的工艺，以及需要购买 OTP、MTP 等 IP，这些 IP 也会增加面积成本。Die 内 NVM 的读取速度有一定的上限，而对 SRAM 的读取可以与 CPU 同频，因此，当 CPU 时钟频率较高时，第 2 种方式的指令执行速度不及第 1 种方式。

第 3 种方式又称 XIP 方式，它针对 NVM 在 Die 外的情况，而且此处的 NVM 特指 NOR Flash。若用此方式，则 Boot 程序会指挥将默认运行地址从 ROM 改为 Die 外 NVM。这种方式的优点是节省 RAM 资源，也没有特定的工艺要求，无须采购 NVM IP，流片成本最低，缺点是 App 运行速度最慢，并且产品质量控制较难。运行速度慢是因为 NOR Flash 是通过 SPI 接口与 SoC 芯片相连的，这是一种串行连接，即使该接口使用快速的 QSPI 并将 SPI 的频率提升到最高的 104MHz，其实际指令读取速率仍然无法与片内 NVM 这类并行连接的速度相比，而且，CPU 取指时一般不会发起 Burst 操作，即读取连续地址上存储的指令，这与程序中的 if 跳转和 for 循环有关，因此 QSPI 无法发挥出它的优势。使用 SPI 读取 NOR Flash 时，一般会先发出 Opcode，即操作命令，然后传输需要读取的地址、模式字等信息，最后才会返回读取的数据。真正的 XIP 方式是省略 Opcode，从而减少 8 个比特的传输时间，即便如此，地址和模式字等仍然需要传输，加之无法使用 Burst，使传输效率只有 40%～60%，即使使用 104MHz 的 SPI 频率，实际指令读取速度也只能达到 6～8 MIPS，因此，XIP 方式只适合于执行慢速应用。另外，由于一个产品被拆分为两个 Die，在量产测试时就需要分别测试两个 Die 的质量，若合封在一个封装里，则对打线，即对 Die 上引脚的连接有较高要求，而且封装类型也受到限制，这就是产品质量较难控制的原因。

上述 3 种方式都需要 Boot 程序指挥将默认地址从 ROM 改为相应的目标器件地址。这种目标地址可灵活改动的特性不是软件赋予的，而是在 RTL 设计中通过地址解析器实现的。地址解析器在 SoC 架构中的位置如图 5-7 所示。连接 CPU 与外设的总线由多条不

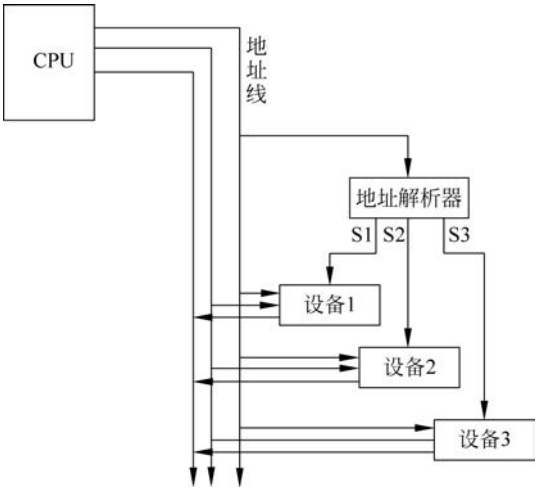


图 5-7 总线连接中地址解析器的位置

同功能的线路组成,这些线路,一部分由 CPU 指向外设,另一部分由外设指向 CPU,其中,地址线就是从 CPU 指向外设的线路。在 SoC 架构中,每个外设都有自己唯一的地址,外设内部也会包含寄存器,或者 RAM,它们也有自己的地址,这样可以保证 CPU 能准确地对某个特定外设的特定寄存器进行读写操作,不会牵连到其他设备。若 CPU 有对某个设备的读写需求,就会向总线发送该设备的地址,该地址能被总线上的所有设备接收,从而可以分析出 CPU 希望操作哪个设备。对地址的分析有两种方式,一种是分布式,一种是集中式。分布式即在每个设备内部建立一套解析逻辑,以分辨 CPU 是否想操作自己,集中式是使用一个统一的地址解析器进行解析,它不会具体到每个设备的每个内部寄存器地址,但它能分析出 CPU 当前希望操作哪个设备。如图 5-7 所示的集中式设计方法较为常见。

在地址解析器中,可以建立一套从 ROM 重映射(Remap)到目标存储器件地址的机制,以完成从 Boot 到 App 的切换。一个典型的地址解析 RTL 代码如下例所示。此例中,假设芯片中的存储设备有 3 种,分别是 ROM、RAM 和 Flash,它们都使用 32 位地址线进行统一编址。当 CPU 发出的地址为 32'h0100???? 时,表示选中 ROM,这里的"?"表示任意十六进制数,也代表 ROM 内部的子地址。当地址为 32'h2000???? 时,表示选中 RAM。当地址为 32'h02?????? 时,表示选中 Flash。这里,"?"的个数可以从 3 种存储器的实际大小来决定,例如 ROM 的子地址有 16 位,说明 ROM 包含 2^{16}B,即 64KB,RAM 也是 64KB,而 Flash 的子地址有 24 位,说明它的最大寻址范围是 16MB。上述是这 3 个存储器的绝对地址,重映射主要是指相对地址,即 CPU 不发出类似 32'h01000001 这样的绝对地址,而是只发出 32'h0 这样的简单地址,则它究竟希望操作哪个设备呢?代码中的 (addr[31:16]==16'h0) &(remap==2'd1) 逻辑用来解析这部分内容,即当用于识别设备的地址前缀全部为 0 时,通过 remap 信号线来判别哪个设备是默认设备。remap 本身可以通过其他设备上的寄存器进行配置,该信号上电后的默认值为 1,即以 ROM 为默认设备,若 App 在 RAM 中运行,则 remap 值由 1 变为 2,若在 Flash 上运行,则 remap 值由 1 变为 0。工程中,将形如 32'h0100000a 的地址称为绝对地址,它由基地址 32'h01000000 和偏移地址 a 组成,基地址用于选定设备,偏移地址用于选定设备中的某个存储器件,该器件的物理介质可以是寄存器也可以是 Flash 等 NVM,单位是字节。如果基地址为全 0,则根据 remap 值选定设备。

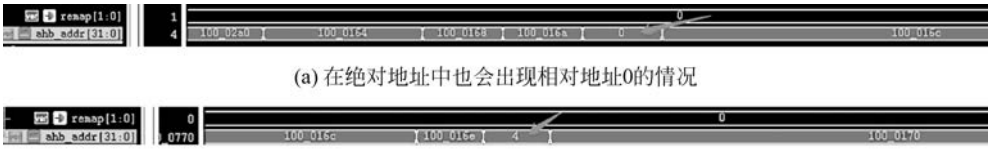
```
assign rom_sel =
    (addr[31:16] == 16'h0100)|((addr[31:16] == 16'h0)&(remap == 2'd1));

assign ram_sel =
    (addr[31:16] == 16'h2000)|((addr[31:16] == 16'h0)&(remap == 2'd2));

assign flash_sel =
    (addr[31:24] == 8'h02)|((addr[31:24] == 8'h0)&(remap == 2'd0));
```

读者可能会产生这样的疑问:"CPU 发出的地址若只使用绝对地址,是否可以不需要考虑重映射的问题呢?"产生这样的疑问,是因为 Keil 等编译器中可以选择指令的地址形式,如图 5-8 所示,IROM1 即为程序编译后存储的位置,也是 CPU 取指的位置,这里填写的

是 0x1000000，即 ROM 的地址，因而 CPU 在取用 ROM 中存储的 Boot 程序时，都会使用以 0x1000000 为基址的绝对地址进行访问。若 IROM1 处填写 0，则 CPU 会直接用偏移地址。这么说来，IROM1 填写设备基址就可以忽略重映射步骤了吗？

图 5-8　在 Keil 设定程序编译后存储的地址

实际上，即使在 Keil 编译时指明了绝对地址，也需要通过重映射切换到默认存储设备。原因是，在从 Boot 切换到 App 的过程中，CPU 会请求地址 0 和地址 4，并不包含基地址，如图 5-9 的(a)和(b)所示，在均为绝对地址寻址的情况下，(a)的中间过程中出现了地址 0，(b)出现了地址 4，而这两处地址必须指向新转换的存储体。



(a) 在绝对地址中也会出现相对地址0的情况

(b) 在绝对地址中也会出现相对地址4的情况

图 5-9　软件切换过程中出现的特殊地址

要实现软件切换，除了在 RTL 设计中需要做地址解析和重映射以外，在 Boot 软件中也需要加入切换指令，这类底层命令往往使用汇编语言编写。一个切换例子如下，__asm 表示它是用汇编语言编写的，可以放在以 C 语言为主的程序中，调用时可以直接使用函数名 mem_switch，与调用普通 C 函数无异。该函数的作用是将 CPU 内部的主栈指针 MSP 赋值为镜像文件中地址 0 所存储的内容，并且将镜像文件中地址 4 存储的内容赋值为 CPU 内部的 PC 指针。

```
__asm void mem_switch(void)
{
    MOVS  R0,#0
    LDR   R1,[R0]
    MOV   SP, R1
    LDR   R1,[R0,#4]
    BX    R1
}
```

一个编译好的镜像文件如图 5-10 所示。可见,其地址 0 的数值为 0x200017F8,其地址 4 的数值为 0x010001A9。当程序运行时,CPU 首先读取地址 0 的值,即 0x200017F8,将其赋值给其内部的 MSP,这样,在程序运行过程中遇到局部变量需要存储或读取,都可以从该地址开始。0x20000000 被 ARM 指定为 RAM 的基地址,因此可知栈的位置在 RAM 中。在对 MSP 赋值完毕后,CPU 会读取地址 4 的值,即 0x010001A9,并将该值赋给其内部的 PC 指针。0x010001A8 是复位入口地址,即不论何种原因使 CPU 复位,都会从该地址上的指令开始执行。PC 指针的目的就是告知 CPU,下一步需要从哪个地址取指令。需要注意的是,地址 4 上的数值是 0x010001A9,而实际指针指向的是 0x010001A8,最低位比特 1 被忽略,是因为该比特具有特殊含义,为 0 代表 ARM 指令集,为 1 代表 Thumb 指令集。综上,在完成了对地址 0 和地址 4 的读取后,CPU 才能正常执行后面的程序。

```
200017F8 010001A9  010001B5 010001B7
00000000 00000000  00000000 00000000
00000000 00000000  00000000 010001B9
00000000 00000000  010001BB 010001BD
010001BF 010001BF  010001BF 010001BF
```

图 5-10　镜像文件示例

不论是运行 Boot 程序还是 App 程序,都需要提供两个存储空间,一个是用于存储指令的代码空间,另一个是用于存储数据的数据空间。在编译器中,一般用 IROM 表示代码空间,用 IRAM 表示数据空间,如图 5-8 所示。实际上,IROM 并非只存储代码(这里指的是编译后的二进制指令,称为 Code),还包括 RO Data 和 RW Data,前者是程序中指定的常数或字符串(在程序运行过程中保持不变),后者是程序中被初始化的全局变量(在程序运行过程中还会发生变化),因此,一般将 IROM 空间表示为 Code+RO Data+RW Data。IRAM 中包含了 RW Data 和 ZI Data(也称 BSS),后者是未经初始化的全局变量,一般,这种变量也会使其初始化为 0,这就是 ZI 名称的由来(Zero Initialization)。可见,RW Data 是 IROM 和 IRAM 的公共部分。RW Data 在 IROM 中占据空间是因为需要存储它的初始值,而在 IRAM 中占据空间是因为它在程序运行过程中数值会发生变化。在 CPU 执行正式任务之前,会先将 RW Data 从 IROM 存储器复制到 IRAM 存储器中,并在 IRAM 存储器中开辟 ZI Data 空间,将其中的字节全部初始化为 0。这一过程是 CPU 根据指令完成的,在 RTL 硬件中无须做任何特殊处理,在硬件设计上,只需保证 IROM 和 IRAM 的空间满足上述几个分段落的要求,具体要求可以在编译文件中查到,例如 Keil 在编译后会产生 Listings 文

件夹，其中在.map 文件中就写有本程序对 IROM 和 IRAM 的需求信息，如下例所示。

```
Total RO  Size (Code + RO Data)                 1156 (   1.13kB)
Total RW  Size (RW Data + ZI Data)               144 (   0.14kB)
Total ROM Size (Code + RO Data + RW Data)       1172 (   1.14kB)
```

IRAM 中还包含堆(Heap)和栈(Stack)两部分。当在主程序 main 或子函数中声明局部变量时，这些变量值都存储在栈中，因此，有些程序会出现栈溢出的情况，这是由于局部变量超出了 IRAM 中栈的容量，一些找不到原因的 Bug 即是由该原因引起的。当然，如果在局部变量声明前加入了 const 关键字，则它会被编译到 RO Data 中，若加入 static 关键字，则会被编译到 ZI Data 中。若在函数中使用 malloc、new 等动态内存分配的命令，则开辟的空间会被安排在堆中。在程序编译前，需要先分配合适的堆和栈，在 Keil 工程中，一般在.s 文件中分配它们，如下例所示，这里的 Stack_Size 指栈大小，分配了 16 字节，Heap_Size 指堆大小，没有分配空间，因此此程序中不允许使用动态内存分配命令。综上，IRAM 由 4 部分空间组成，分别是 RW Data、ZI Data、栈空间、堆空间。

```
Stack_Size          EQU       0x00000010
                    AREA      STACK, NOINIT, READWRITE, ALIGN = 3
Stack_Mem           SPACE     Stack_Size
__initial_sp

Heap_Size           EQU       0x00000000
                    AREA      HEAP, NOINIT, READWRITE, ALIGN = 3
__heap_base
Heap_Mem            SPACE     Heap_Size
__heap_limit
```

通过以下 C 语言的示例可以辅助理解数据的存储区域，其中，myArray 为全局数组，并声明为常数 const，因此属于 RO Data，与代码一起存储于 IROM 中；aa 为全局变量，并有初始值，因此它属于 RW Data，既存在于 IROM 中，又存在于 IRAM 中；bb 为全局变量，无初始值，因此它属于 ZI Data，只存在于 IRAM 中；cc 虽然是局部变量，但它前面有 const 关键字，说明是常数，属于 RO Data；dd 虽然是局部变量，但它前面有 static 关键字，说明它是静态变量，属于 RW Data；ee 是局部变量，不管它是否有初始值，该值均存储于 IRAM 的栈中。指针 ff 指向一段堆空间，该空间存在于 IRAM 中。程序编译得到的指令，作为 Code，存储于 IROM 中。

```
const char myArray[3] = {7, 8, 9};          //RO Data
int aa = 10;                                //RW Data
int bb;                                     //ZI Data

int main(void)
{
    const  int cc = 11;                     //RO Data
    static int dd;                          //RW Data
```

```
    int ee;                          //Stack Data
    int *ff;                         //该指针指向 Heap 中的一段地址

    ff = (int *)malloc(20);          //在 Heap 中动态分配一段 20 字节的区域
    ...
}
```

镜像文件(Image)是程序被编译后生成的,一般会生成二进制非 ASCII 码和十六进制 ASCII 码两种格式,可以等同于 IROM,但比上文的 IROM 多一个中断向量表,该表存放在该文件的起始处,以地址 0 为起始,包括图 5-10 所示的栈指针地址、各中断服务程序存放的位置等。文件的后半部分为 IROM 中的 Code、RO Data、RW Data。所谓中断向量表,就是罗列了每个中断对应的服务程序入口地址的表格。中断由外设发给 CPU,用于引发 CPU 的注意,并做出相应的处理。每个中断,在程序中都有相应的处理措施,称为中断服务程序,这些程序都属于 Code,它们分散存储在镜像文件的不同位置,因而需要一张表格把每个中断服务程序的位置标识出来。

Boot 程序的镜像文件固定在 ROM 中,流片后不能改动,App 程序的镜像文件烧写在 NVM 中,流片后可根据 NVM 的特性进行修改,最常用的是可以任意擦写的 Flash 存储方式。ROM、RAM、NVM 都有对应的基地址,在镜像文件中显示的地址仅为偏移地址,当 CPU 要读取镜像中的一个数据时,需要将对应设备的基地址加上偏移地址,组成完整地址后再发到总线上,若仅包含偏移地址,则具体指向哪个存储器取决于上文所配置的重映射的编号。例如,镜像文件中的地址 5,若文件存储在 ROM 中,则在 CPU 视角下它的地址是 0x01000005,若文件存储在 RAM 中,则在 CPU 视角下它的地址是 0x20000005,若文件存储在 Flash 中,则在 CPU 视角下它的地址是 0x02000005。这些基地址已经在编译时指定了,如图 5-8 所示。

在编译 Boot 程序时,虽然 IROM 指定的是 ROM 地址,但 IRAM 仍然需要指定到 RAM 地址,因为只有 RAM 才是最适合快速读写的存储介质。同样,App 程序也需要将 IRAM 指定到 RAM 地址中。那么,Boot 的 IRAM 地址和 App 的 IRAM 地址需要分开吗?答案是不需要,RAM 特别是 SRAM,单位面积十分昂贵,使用时需要特别节约。理论上,Boot 的 IRAM 数据用于保证 Boot 顺利运行,当 Boot 结束运行并切换到 App 后,Boot 的 IRAM 失去了存在的意义,因而可以被 App 的 IRAM 覆盖并进行空间重用。需要注意的是,对于第 1 种 App 运行方式,即将代码从 NVM 复制到 RAM 中运行的方式,应避免 App 的代码将 Boot 的 IRAM 数据覆盖,因为此时 Boot 仍然在运行,仍然需要这些数据,所以不能被覆盖。一般,若代码从 RAM 的起始处复制,则 Boot 在指定 IRAM 时需要指定到 RAM 空间的末尾处,以避免被覆盖。指定 IRAM 的起始位置在图 5-8 的 IRAM1 处,其大小可以使用 RW Data + ZI Data + Stack Size + Heap Size 来计算。

注意 Stack 在很多资料中被翻译为堆栈,这种翻译容易使初学者误认为是堆(Heap)和栈(Stack),因此,这里推荐称为栈,相应地,将 SP 称为栈指针。

5.6 程序的分散加载

4min

SoC 芯片中程序的运行，一个镜像文件除了可以在指定的单个存储介质中运行外，还可以分散到多个存储介质中运行，例如，main 函数在 NVM 中运行，但其中一些子函数在 RAM 中运行。这样的好处是，如果 RAM 的速度快，而 NVM 的速度慢，程序中的某些功能要求运行响应速度足够快，而 NVM 无法满足这样的要求，但是，RAM 的空间不够，无法装下整个镜像文件。此时，就可以将那些对速度要求较高的功能映射到 RAM 中，而将其他要求不高的功能仍然放在 NVM 中，这样就实现了 RAM 面积与运行速度之间的平衡。这种将一个完整的程序分别从不同存储介质中取指的方式称为分散加载，也叫动态加载。

在 Keil 中进行分散加载的方法如图 5-11 所示。在 Options 中选择 Linker 标签，在 Scatter File 一栏中填入文件名，扩展名为.sct。

图 5-11 在 Keil 中设置分散加载文件

分散加载文件 abc.sct 的一个示例如下。此例假设 NVM 的基地址是 0x02000000，RAM 的基地址是 0x20000000。首先，将程序映射到 NVM 中，大小是 64KB(0x10000)。将除了 ram_app.o 文件以外的其他文件都放在 NVM 中，包括镜像文件开头的中断向量表。将 ram_app.o 单独映射到 RAM 中，大小是 1KB(0x400)。IRAM 部分也映射到 RAM 中，但这段区域被映射到 ram_app.o 后面，即地址 0x20000400 开始的位置。文件中，app_exe、ram_exe 等段名称均可以更改，关键是基地址、大小、括号中的内容要正确。.o 文件是程序编译后生成的目标文件，一般一个 C 语言文件就对应一个.o 文件，因而对于需要映射到 RAM 中的函数 ram_app，需要单独建立一个文件 ram_app.c，这样它就可以生成一个单独的.o 文件，并单独被映射。ram_app 函数也可以与其他程序放在一起，不过此时，需要加入

一些编译器控制声明,使编译器能辨认出它的特殊性,例如,在函数前面加一句__attribute__((section("ram_exe"))),也可以达到同样的效果。

```
LR_IROM1 0x02000000 0x10000
{
    app_exe 0x02000000 0x10000
    {
        * .o (RESET, + First)
        * (InRoot $ $Sections)
        .ANY ( + RO)
    }

    ram_exe 0x20000000 0x400
    {
        ram_app.o ( + RO)
    }

    RW_IRAM2 0x20000400 0x1400
    {
        .ANY ( + RW + ZI)
    }
}
```

注意,不要误以为 ram_app 单独被烧写到 RAM 中。实际上,整个镜像文件,包括 ram_app,仍然被烧写在 NVM 中,但在执行镜像文件时,CPU 会先将 ram_app 复制到 RAM 的 ram_exe 区域(使用 VCS 等仿真工具对 SoC 芯片进行仿真时能够清晰地看到这一过程),然后才开始执行。这一复制过程是 CPU 根据镜像文件的指令自动完成的,在 RTL 方面不需要做任何控制。

5.7 SoC 芯片程序的烧写方式

SoC 芯片除了运行程序外,程序的烧写也是一个必须重视的问题。这里要烧写的程序主要指 App,即对 NVM 的烧写,因为 Boot 已经固化在 ROM 中,不需要烧写。一般的 SoC 芯片需要提供 4 种以上烧写方式,分别如下:

(1) 量产烧写方式。

(2) 调试烧写方式。

(3) App 运行时的烧写方式。

(4) 在线升级烧写方式。

量产烧写方式已在 5.5 节中介绍过,这里介绍另外 3 种方式。

调试烧写方式是嵌入式软件人员修改软件程序后采取的烧写方式,它与量产烧写方式的区别在于后者是对单独的芯片进行烧写,芯片尚未焊接,仅仅放在机台的 socket 插座上,因此全部引脚都可以用于烧写,自由度较高,而调试烧写一般是芯片已经焊接在印制板上,

只有少量引脚可以用于烧写，因此这种烧写方式需要更加紧凑的设计，相应的代价是烧写速度可以比量产烧写慢一些。

ARM CPU 中包含一个调试专用模块 DAP，其作用是向用户提供 Debug 接口，可以使 CPU 复位和暂停、读写 SoC 内部各存储空间等。可以将 DAP 理解为用户干预 CPU 的接口，当用户调试时，用户本人就是另一个 CPU，芯片内部的 CPU 被用户取代，用户通过操作 DAP，从 CPU 手中夺取了总线的控制权，可以对总线进行任意操作。用户操作 DAP 的协议方式有两种，分别是 JTAG 和 SWD，它们的协议都比较复杂，一般用户不会自己构造协议的波形，而是通过仿真器间接实现对 DAP 的控制。仿真器是一种硬件，它内部包含对 JTAG 和 SWD 协议的解析逻辑和波形发生逻辑。它的一端连接计算机的 USB 接口，另一端连接 SoC 芯片电路板的 JTAG 或 SWD 接口，起到将 USB 信号与 JTAG/SWD 互换的作用。常用的仿真器有德国 SEGGER 公司的 JLink、ARM 公司自己出的 Ulink 等。

JLink 面向 SoC 芯片一侧的接口如图 5-12 所示，从图中可以看出 JTAG 和 SWD 分别需要哪些引脚。JTAG 需要连接 VCC 和 GND 两根信号线，以及 TMS、TCLK、TDI、TDO，而 TRST、RESET、RTCK 等 3 根信号线是可选的，其中，TMS 是模式设置线，方向是从仿真器指向 CPU；TCLK 是仿真器向 CPU 提供的调试时钟信号；TDI 是仿真器向 CPU 输入的数据；TDO 是 CPU 返给仿真器的数据；TRST 是仿真器向 CPU 发出的复位信号，只复位 DAP，低电平有效；RESET 也是仿真器向 CPU 发出的复位信号，会复位整个芯片，即发起总线复位，低电平有效；RTCK 是 CPU 给仿真器反馈的时钟，因为仿真器给 CPU 的时钟 TCLK 一般与 CPU 的内部时钟是异步的，而有些 SoC 设计，为了提高 JTAG 的通信速率，要求 TCLK 与 CPU 的内部时钟同步，此时就需要 RTCK 反馈给仿真器，以便它能即时调整 TCLK 的频率和相位，与 CPU 同步。SWD 的引脚与 JTAG 的引脚是复用的，其中，SWCLK 是仿真器向 CPU 输出的时钟信号；SWDIO 是仿真器与 CPU 交互的双向数据线；通过 SWO 和 CPU 可反馈一些诸如 MIPS、中断源等信息；RESET 也用于复位整个 SoC 系统。使用 SWD 时，SWCLK、SWDIO 是必需的，再加上 GND 连接，即可进行通信，而 JTAG 连线数量至少为 6 根，因而越来越多的调试选择使用 SWD 接口。

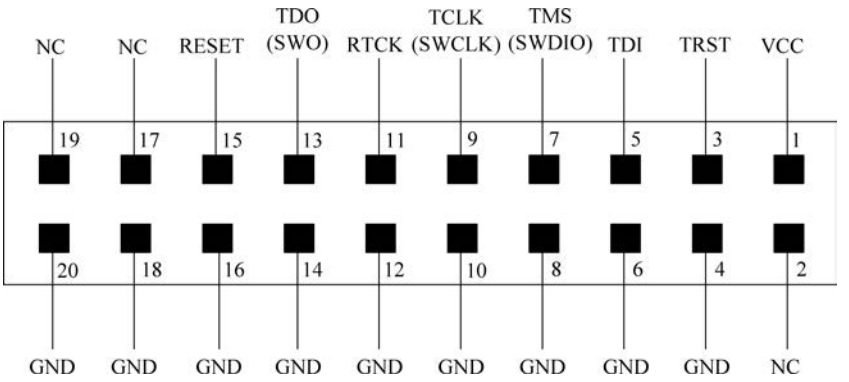


图 5-12 JLink 仿真器的引脚分布

使用调试方式对 NVM 进行烧写,也同样使用 JTAG 或 SWD 进行,思路是利用这些协议能够通过总线对任意地址进行读写的特性,将镜像文件的字节依次写入 NVM 中。如果使用 SWD,则最少只需占用芯片上的 3 个引脚(SWCLK、SWDIO、GND)就可以对 NVM 进行烧写。上文已提到,CPU 往往不能直接对 NVM 进行操作,需要通过 NVM 控制器进行间接操作。因而,烧写的地址和数据在进入总线后,也是先经过控制器,由它翻译为 NVM 可识别的数据格式后才能真正进行烧写。若控制器的操作过程比较复杂,或效率低,也可以考虑在烧写时绕过控制器,直接产生 NVM 需要的信号进行烧写。在控制器与直接产生的信号之间,通过 MUX 进行选择。在图 5-5 的 3 选 1 MUX 中,GPIO 设备可以直接操作 NVM 的设计,也就是出于这样的考虑,图中 NVM 是外部 NOR Flash,通过 SPI 接口与 SoC Die 相连。这种 Flash 是按页烧写的,一页是 256 字节,烧写时,一页数据放在一个 SPI 帧中才能保证烧写效率最高,烧写一页约需要一毫秒,若将同一页的不同字节分开烧写,则每次烧写都需要消耗一毫秒,烧写速度明显下降。若使用 Flash 控制器,先存储一页内容,然后烧写,需要开辟额外的 RAM,并使用 DMA 方式,这样增加了 Flash 控制器的面积和设计难度。使用 GPIO 设备产生 SPI 波形是一个折中的方案,由于该 Flash 的烧写速度主要取决于等待的一毫秒时间,而不是 SPI 时钟频率,或 SPI 输入数据的并行度,这样就降低了 GPIO 模拟 SPI 波形的难度,可以使用普通的兆级速率,单方向 SDI 和 SDO 这样的简单方式即可满足页烧写的要求。如果使用 JLink 仿真器,则可以使用配套工具 SEGGER Embedded Studio 进行烧写机制的编程。该程序会通过 SWD 或 JTAG 接口搬移到 SoC 芯片的 RAM 中,并被运行,运行的效果是图 5-5 中的 MUX 被切换为 GPIO 设备与 NVM 相连,并且凡是从总线上获得的数据,就通过 GPIO 设备模拟成 SPI 格式发送到 NVM 中。

很多 SoC 芯片设计厂商还会自研烧写器,该器件的核心也是一片较高性能的 MCU,它会模拟 SWD 或 JTAG 接口信号,将其内部存储的镜像文件转换为这些接口信号并烧写到放在 Socket 插座上的 SoC 芯片中。这样就不需要使用 JLink 或 ULink,也节省了连接计算机的时间开销和操作开销,烧写速度较快,因而也可以作为一种量产烧写方式使用。

在 App 运行时,有时也有烧写 NVM 的需要,例如产品厂商希望长期保存一些信息,但该信息在出厂时并不确定,而是由用户的使用习惯决定的,这些信息放在寄存器里或 RAM 里都会丢失,只能保存在 NVM 中。对于这种情况,使用 NVM 控制器作为烧写媒介就比较合适,因为 NVM 内部的 Code 在 CPU 运转时会通过 NVM 控制器被不断读取(Code 运行在 RAM 上的除外),因此该控制器的管辖权不应被剥夺,烧写时,也只能通过控制器烧写。需要注意的是,严禁烧写的数据破坏镜像文件区域,应烧写在 NVM 的空白区域中,一旦镜像文件被破坏,在本次运行或下次重新上电运行时,会发生无法恢复的错误。

在线升级烧写是针对具有以太网口或无线联网能力的芯片,使来自网络的数据流进入芯片中,对 NVM 进行烧写。其处理难度主要在于芯片中的网络部分,其他设置与调试烧写方式基本相同。其优点是一旦产品需要修改或升级,无须召回即可联网更新,最常见的就是手机或平板电脑在线更新 ROM 的方式,在消费电子中将 NVM 中存储的镜像也称为 ROM。这种方式对于无网络能力的芯片不适用。

5.8 SoC芯片的参数校准

芯片内部的参数并不是在设计时就完全确定的，例如内部RC振荡器的设计频率和实际运行频率存在偏差，内部各电源域通过LDO或DCDC供电，供电电压未必会达到设计时的精度要求，对于电压、电流、温度的检测，也有不准确的可能，这是模拟电路上的不确定性，其根源来自EDA的仿真模型与实际版图性能的偏差、制造工艺的偏差、封装散热、实际使用方式与设计时的想象不一致等，因此，模拟工程师在设计时也不会将电路做死，而是会留一些可供调节的参数。数字电路中虽然电路仿真与实际不一致的情况几乎不存在，但对于与物理特性有关的性能参数，仍然需要调节，特别是通信芯片，实际通信环境与算法仿真环境有时相差很大，在芯片中需要准备多种预案及可调参数，这些都要做成电路摆放到芯片中，在使用时，根据不同情况选择启动合适的算法电路和参数，以求达到满意的效果。这些不确定性，在芯片设计时无法预料，补救措施是量产测试时的校准步骤及面向用户升级驱动程序。

RC振荡器的频率、芯片内部的供电电压等参数的校准，可以在产线上测试，并由机台直接烧录到NVM中(仍然需要找一个预定的空白区域)，而一些需要人工实验测试才能确定的参数，如果能在量产前确定，则可以通过机台烧写，若赶不上量产测试，则只能通过在线烧写或在产品生产过程中通过调试烧写方式进行。

校准参数有3种方法。第1种方法是作为单独的数值烧写在NVM的空白区域，当软件运行时，会自动读取该区域，并将该数值配置到相应的寄存器中以便起效。这种方式适用于芯片之间存在差异，无法做到每个芯片都使用同一个参数的情况。同时，也说明虽然不同芯片间在该参数上存在差异，但在一颗芯片内部，不会因为时间、温度等环境因素导致芯片的该参数需要重新校准。只有上述两个条件都具备，才使用第1种参数改法。第2种方法是直接修改软件程序，将实验后确定的参数写进去，做成新的镜像文件，替换NVM中原有的镜像文件。这种方式适用于不同芯片间没有差异，能够通用一个参数的情况。第3种方法是实时校准。有些参数，不仅不同芯片之间存在差异，同一芯片在不同时间、不同温度下也存在差异。对于这种情况，必须在芯片内部设计校准专用电路，芯片上电后，校准电路也开启，随时跟踪芯片状态，并计算出合适的参数配给寄存器。这类参数中比较典型的是无线射频电路中的IQ失配补偿参数、载波泄露补偿参数，以及天线非线性放大预失真等。

第1种校准方法有时也会使用第3种校准方法代替。原因是第1种方法对机台的测试环境、测试精度有要求，并且还占用测试时间，而测试时间又是芯片成本的一部分。为了避免参数不准确并节省成本，对于第1种参数，也会在芯片中设计专门的校准电路，但复杂度低于第3种校准，因为它没有实时校准的需求，仅上电完成一次校准即可。实时校准要求在保留正常数据业务的前提下进行校准，因此对校准提出了更高要求，而上电一次性校准电路不需要保留正常业务通道，它可以在上电时将需要校准的参数通过MUX切换到校准专用

通道,校准完毕后再切回到正常业务通道,因而设计较为简单。一个典型的例子如 RC 振荡器频率校准使用的 AFC 电路。

5.9 SoC 芯片的上电异常保护

芯片上电时,由于芯片插头的电平不稳定,存在纹波,可能会导致芯片能够上电,但上电后状态不正常。特别是消费电子,在用户频繁拔插电源的情况下,由于手在插电时会发生抖动,电源接通其实是时断时续的,会有一定概率导致芯片功能异常,因此,在芯片设计时,需要考虑到这类情况,引入一些机制,使芯片在异常后可以自动复位,进入正常状态。主要的保护机制有两种,一种叫欠压保护(Under Voltage LockOut,UVLO),另一种是上文介绍的看门狗设备。

UVLO 是模拟电路。上电时,供电电压可能发生剧烈波动,如图 5-13 中 VDD 所示。UVLO 是对 VDD 的测量,并且使用两个门限来衡量 VDD 的大小,一个是复位门限,另一个是恢复门限。当 VDD 低于复位门限时,上电复位信号(Power On Reset,POR)就会变低,使整个芯片复位,当 VDD 高于恢复门限时,POR 就会升高,芯片正常工作,图中 VDD 上水平的两条虚线就是这两个门限,它们的设置不应相同,其差别称为回差,回差可以防止 VDD 震荡导致的 POR 震荡。图中恢复门限与 VDD 交汇于 A 点,使刚上电的 POR 变为高电平,但是紧接着 VDD 突降,与复位门限交汇于 B 点,POR 变为低电平,导致芯片复位,然后 VDD 又升高,与恢复门限交汇于 C 点,POR 变高,芯片复位结束,后面的 D、E 两点虽然也是 VDD 与恢复门限的交汇点,但是由于此时 POR 是高,因而对 POR 没有影响,也说明 VDD 在这两个点的抖动尚不够剧烈,设计中认为它不会造成芯片内部逻辑的紊乱,因而可以忽略。UVLO 的加入,使原本不够敏感的 POR 变得比较敏感,从而增加了 POR 复位的机会及复位的时间宽度,对保证芯片内部的稳定有一定意义。

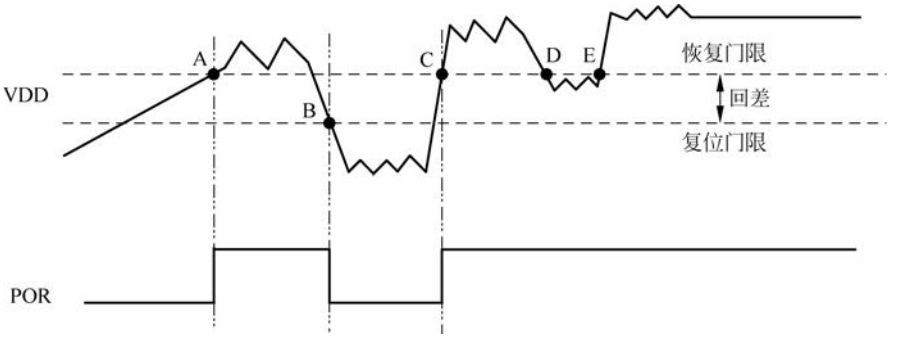


图 5-13 UVLO 对复位的控制

看门狗设备的功能已在 5.4 节中介绍过,这里不再赘述。在实际应用中,看门狗有一定的局限性,例如,在 Boot 中一般不会打开看门狗,因而如果芯片锁死在 Boot 中,同时 UVLO 设计或设置不当,芯片就会一直死机而得不到复位,另外,看门狗本身是基于时钟

的，如果上电异常，时钟波形混乱，则无法满足看门狗内部触发器的建立时间和保持时间，也会造成看门狗电路功能异常，从而失效。因而，UVLO 是保护电路功能的第一道屏障，而看门狗更适合防范由于驱动软件操作失误而引发的异常，如 HardFault 等软件问题。

一般，UVLO 通用于 SoC 芯片和非 SoC 芯片，而看门狗用于 SoC 芯片。

5.10 ARM Cortex-M0 介绍

ARM 的 Cortex-M0 是一款结构简单且功能方面足以满足基本程序运行需求的 CPU，比较适合广大初学者学习。M 编号意味着它是 ARM 公司为移动可穿戴设备设计的处理器，M0 是这一系列处理器中的最低配版本，此外，还有 R 系列是在实时性方面进行了优化的版本，以及 A 系列所代表的高性能处理器版本。目前，Cortex M0 处理器已经加入了 ARM 的 DesignStart 项目，初学者可以免费下载到它的评估版本，并可在 FPGA 上综合，以便学习它的操作流程、工作原理和处理特性。其核心代码仅两万余行，使用低端 FPGA 也能运行。下文将 Cortex M0 称为 CM0。

不像复杂的 CPU 内部有多级流水，CM0 内部流水只有 3 级，分别是取指、解析和执行。所谓流水操作，就是前面的一条指令正在执行的同时，后面的指令正在被解析，而再后面的指令正在被读取。这样，取指电路、解析电路、执行电路都可以得到充分利用，中间没有等待和空闲，除非取指太慢导致前面的解析和执行都空闲了，而指令还没取过来。因而取指的速度决定了 CPU 的运行效率，在 5.5 节中介绍了多种程序运行方式，就是为了做到速度与成本之间的平衡，以适应不同产品的需求。CM0 的 3 级流水是用寄存器划分的，说明其内部执行一条指令需要打 3 拍。对于较高端的 CPU，由于时钟频率高，建立时间的要求比较苛刻，因此在寄存器之间不能有太长的组合逻辑，需要在 CM0 基础上将组合逻辑再拆分，中间用寄存器隔开，从而造成了流水较长的状况。由此也可以看出，3 级流水决定了 CM0 不能使用太快的时钟。

CM0 的架构被称为 ARM V6-M，其实所谓架构就是指里面的指令集，一旦指令集确定，则对应的内部解析电路也就确定了，因而才称为架构。ARM V6-M 架构包括 Thumb-1 的 16 位指令集及部分 Thumb-2 的 32 位指令集，总共有 56 个指令，其中 50 个为 16 位指令，其余 6 个为 32 位指令，它们分别是：BL、DMB、DSB、ISB、MRS、MSR。16 位指令与 32 位指令混用是 Thumb 指令集的特性，可以避免一个简单的动作却占用了一条长的指令，从而降低了执行效率和存储效率。以 16 位指令为主的方式，可以使 CPU 的 32 位总线一次性读入两条指令，其中一条被执行，另一条被缓存，相当于一个小型的 Cache。Thumb 本意为大拇指，大拇指由关节控制可以进行对称折叠，象征着 32 位和 16 位指令可以自由切换，这就是 Thumb 名称的由来。关于纯 32 位的 ARM 指令集和混合长度的 Thumb 指令集的由来及演变，牵扯到 ARM IP 的演进例程，这里不再赘述。需要记住的是，CM0 只支持 Thumb 指令集，因而在某些数据和控制中需要选择 ARM 或 Thumb 时，应选择 Thumb。

如图 5-10 的地址 4,其最低比特为 1,代表 Thumb 指令,在 CM0 中,镜像文件地址 4 的最低位保持 1 不变。

CM0 的地址线是 32 位,因此寻址空间是 32 位,即最多能支持 2^{32}B=4GB 空间的寻址。其内部包含 AHB-Lite 总线解析逻辑。外围设备可以设计为总线要求的接口,并挂在该总线上,每个外设都拥有独立的地址,这样 CM0 能够掌握每个设备的每个寄存器。

CM0 内部的寄存器不需要通过总线访问,而是直接访问的,因此操作速度快(省去了总线握手环节)。这些寄存器分为两类,一类称为寄存器组,另一类称为特殊寄存器。

寄存器组中共有 16 个寄存器,命名为 R0~R15,其中,R0~R12 是普通寄存器,运行指令时用得最多的是 R0~R7,而 R8~R12 在执行特殊任务时才会用到。

R13 存储的是主栈指针 MSP 和进程栈指针 PSP,目的是帮助 CM0 找到栈的位置,并将软件中的局部变量保存到栈中或从栈中读取。在 CM0 中,主要使用 MSP,例如在第 5.5 节介绍的开机复位过程及从 Boot 到 App 的切换过程中,都是将镜像文件的地址 0 中的数据写到 MSP 中,让 CPU 知道 main 函数对应的栈位置。PSP 只在使用操作系统时才会用到,因为操作系统中系统和应用运行在不同的栈上。由于 CM0 的性能限制,较少使用操作系统,至多使用类似 RTOS 这样的微型操作系统,因而 PSP 并不常用。CM0 内部既有 MSP 寄存器又有 PSP 寄存器,但给用户使用时却只提供了一个接入点,也就是 R13。至于 R13 究竟指的是 MSP 还是 PSP,取决于特殊寄存器 CONTROL 的配置。CM0 的运行状态有两种,开机上电正常运行时称为线程模式(Thread Mode)和在中断时会自动进入的中断模式(Handler Mode)。MSP 和 PSP 只有在线程模式下才可选,在 CPU 处理中断时,栈指针固定使用 MSP。

注意 很多资料中将 Handler Mode 翻译为处理者模式,这虽然是直译,但没有体现该模式的用途。在软件中,中断服务程序一般命名为 Handler,例如 UART_Handler 是 UART 中断服务程序的函数名,因此,Handler 这里就是指中断服务程序,Handler Mode 应翻译为中断模式,这也体现了它的功能。

R14 是链接寄存器 LR,它用来记录子函数在母函数中的位置,以便使 CPU 在执行完子函数后能顺利回到母函数并继续执行下一条程序。

R15 是程序计数器 PC,用来告诉 CPU 下一条指令存储的地址,CPU 知道该地址后才会向总线发起读该地址的操作,以索取指令。

特殊寄存器包括 3 个,分别是 xPSR、PRIMASK 和 CONTROL。

xPSR 是程序状态寄存器,它其实是 3 个寄存器的合并。这 3 个寄存器分别是应用状态寄存器 APSR、中断状态寄存器 IPSR、执行状态寄存器 EPSR。APSR 记录了 CPU 计算结果的情况,包括是否为负数(用 N 表示该状态位)、是否为 0(用 Z 表示该状态位)、是否发生了无符号数运算的溢出(用 C 表示该状态位)、是否发生了有符号数运算的溢出(用 V 表示该状态位)。IPSR 用来显示当前正在执行的中断的编号(中断号)。CM0 中每个中断有自己的编号,从-14 到 31,可用 6 比特表示。EPSR 中只有一个有意义的比特,用来表示当

前指令集是 ARM 指令还是 Thumb 指令，由于 CM0 只支持 Thumb 指令，因而该寄存器的值永远是 1(用 T 表示该状态位)。这 3 个寄存器可以通过读它们的统一接口 xPSR 一次性获得，xPSR 中各状态位的位置如图 5-14 所示。需要注意的是，xPSR 是状态寄存器，因此，它只能读，不能写。在芯片中，状态寄存器都是只能读不能写的。这里的读写，是针对软件用户而言的。写普通软件时，需要用到 xPSR 状态的情况较少，当两个不同的中断号共用一个中断服务程序时(例如 gpio0_Handler 和 gpio1_Handler 共用一个中断服务程序)，才需要读一下中断号，以确认当前是哪个中断在被处理。

N	Z	C	V	保留位	T	保留位	中断号
31	30	29	28		24	5	0

图 5-14 xPSR 各状态位的位置

PRIMASK 寄存器只有一个比特可用(比特 0)，负责对除了复位、NMI 和 HardFault 中断以外的其他中断功能进行总使能或总屏蔽。写 0 为不屏蔽，写 1 为屏蔽。屏蔽的好处是当 CPU 需要专注于某种事务且一段时间内不希望有任何中断打扰时，可以仅用这一个比特达到目的，这种情况发生在对软件实时处理要求较高的场合，例如，一个通信设备，偶尔需要发送数据，数据的长度和格式要求严格，软件在构造数据时，如果中间有中断处理任务，则数据的格式就会出错，此时最适合使用屏蔽策略。值得注意的是，写 0 虽然是不屏蔽，但在 CM0 的 NVIC 器件中，还有其他控制字可对中断进行屏蔽，因此，屏蔽中断的手段不只有这一种。

CONTROL 寄存器中比特 1(次低位)用于设置 R13 代表的是 MSP 还是 PSP。当其为 0 时表示 R13 读写的是 MSP，为 1 时表示 R13 读写的是 PSP。上文已经提到，当 CM0 处于中断模式时，R13 固定是 MSP，此时 CONTROL 只能读，并且显示为 0。只有当处于线程模式时才能自由配置。在 CM0 的应用中，较少使用操作系统，因而对 CONTROL 的控制也不常有。

上述 16 个寄存器组成的寄存器组，以及 xPSR、PRIMASK、CONTROL 三个特殊寄存器不需要总线就能直接访问，因此它们没有地址，它们的名字本身就是地址。例如汇编命令 MOVS R0，#0，是将 0 写入 R0 寄存器，R0 没有具体的地址。

在下载的 CM0 套件中，可以看到其文件层次结构如图 5-15 所示，其主要包括两层，内层为 CORTEXM0 层，外层为 CORTEXM0INTEGRATION 层。CORTEXM0 中既包含 CM0 内核，还包含 NVIC 和 Debug 两个模块。CORTEXM0INTEGRATION 中既包含 CORTEXM0，还包含了 WIC 和 DAP 两个模块。做 SoC 集成时，一般会例化 CORTEXM0INTEGRATION 层，并在 AHB 接口中添加需要的设备，将各设备的中断线连接在中断接口处，如此就完成了一个简单的 SoC 芯片设计。上面所讲的流水线、指令集、寄存器组与特殊寄存器都属于 CM0 核的内容。

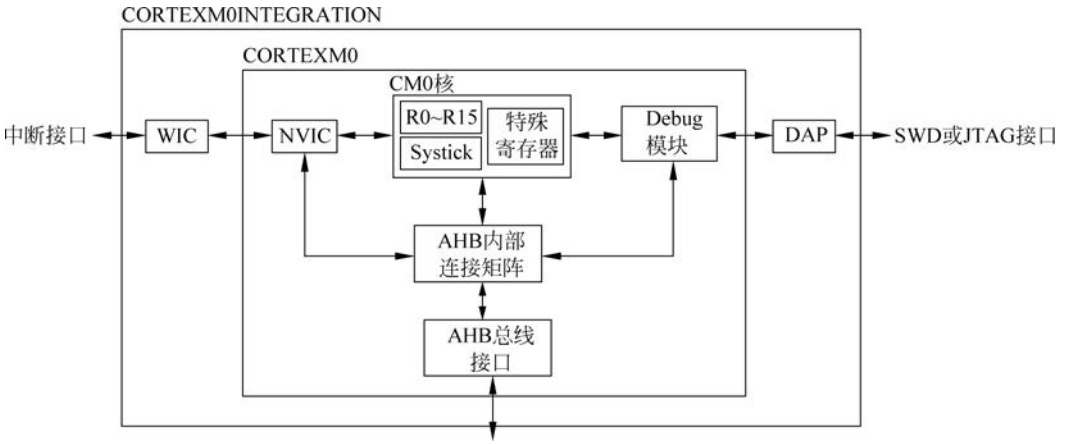


图 5-15　CM0 的内部组成模块

5.11　中断机制

内嵌向量中断控制器(Nested Vectored Interrupt Controller,NVIC)可以对 32 个普通中断进行优先级、掩码、挂起的判断。它可以决定一个中断是否能够生效(也称中断使能)、是否需要挂起等。它放置在 CORTEXM0 层中。

这里需要解释中断使能和挂起的含义。中断在芯片内部其实是普通的金属连线,在外设中产生,输入 CPU 里,平时是低电平,当外设需要 CPU 关注时,会在中断线上发出高电平,并且会 Latch 住,不会自动复位为低。此时,若该中断在 NVIC 中对应的使能位是有效的,则该高电平会成功地进入 CPU。CPU 收到该中断后,会停止获取预定的下一条指令,将当前运行的环境保存下来(将 PC 的值写到 LR 中),然后马上去查询中断向量表(镜像文件最开始的 192 字节),找到该中断编号对应的向量位置,将该位置写入 PC 中,于是再次取指令,取出来的就是该中断对应的中断服务程序的内容了。在该中断服务程序中,CPU 会对发起中断的设备中的寄存器进行读写,并将该中断清除掉(清除中断控制位在该外设的寄存器上,而不是在 CPU 里)。当中断服务程序执行完毕后,CPU 会将 LR 的数据再次写回 PC 中,这样它就可以从原来被中断打断的地方继续运行了。如果在 NVIC 中设置为不使能该中断,则尽管该中断线是高电平,CPU 仍然无动于衷,不会被打断。可见,对于中断的操作,实际上分为 3 个层次,分别是外设、NVIC 和 CM0 核。外设负责发起中断电平信号,这是中断的根源,如果用户想切断这个根源,则可以通过设置外设的寄存器,让中断电平不会变高。NVIC 把守着中断线与 CPU 之间的门户,只有 NVIC 允许中断线通过,它才能通过。最后在 CM0 内核中,还有上文介绍的 PRIMASK 寄存器把守最后一关,只有它为 0,中断才能起到真正引发 CPU 的作用,因此,一个中断要使能,需要打开设备内部中断使能、NVIC 中断使能,并保持 PRIMASK 为 0。

每个中断都有自己的优先级,优先级共分 4 级,即 0～3。数值越低,优先级越高。高优

先级的中断会先响应，即先被 CPU 处理。当 CPU 同时被两个中断打断时，它会先进入优先级高的中断服务程序进行处理，此时，低优先级的中断就处在挂起的状态，等待 CPU 处理。当 CPU 正在处理一个中断服务程序时，又来了一个中断，CPU 会看新来的中断是否具有比当前中断更高的优先级，若是，则会停止处理当前服务程序，而进入高优先级的中断服务程序中，待处理完成后再回头处理过去的中断程序。当两个中断优先级相同时，就按到来的先后顺序处理。还有一种情况是某个中断服务程序正在处理，相同的中断又到来了，即中断处理的速度赶不上中断生成的速度，此时，中断仍然会被挂起，待本次中断处理完成后再次启动此中断的处理。在 NVIC 中，中断可以通过人工挂起，即本来没有中断，但通过设置，可以制造一个虚假的中断，但这种方式并不常用。综上，挂起是指一个中断信号已经为高电平，但尚未引发 CPU 执行其服务程序，处于一种等待服务的状态。

初学者往往分不清中断(Interrupt)和异常(Exception)。在硬件中称中断，在软件中称异常。在 ARM 自己的官方文件中，硬件描述都用 Interrupt，而其软件注释中相同的中断却注释为 Exception。在软件中，异常的含义比中断更广泛，除了外设发出的引发 CPU 注意的信号，还有运行软件时出现的错误，也属于异常，但由于 CM0 的系统简单，一般也不运行操作系统，所以在这里中断基本等同于异常。

值得注意的是，CM0 中除了 32 个普通外设中断外，还有 6 个优先级较高的特殊中断，分别为复位中断、NMI 中断、HardFault 中断、SVC 中断、PendSV 中断、SysTick 中断。NVIC 只负责管理 32 个普通外设中断，其他 6 个中断在 SCB 寄存器中进行配置(详见表 5-4)。

复位中断是芯片在上电或热复位后必然会进入的，它将引导程序进入主程序 main。它的优先级是最高的。它在 CORTEXM0INTEGRATION 层没有硬件接口，属于内部中断。在下载的 designStart 包中，软件文件 startup_CMSDK_CM0.s 中已经包含复位中断服务程序 Reset_Handler 的汇编代码，一般不会去改变它。

不可屏蔽中断(Non Maskable Interrupt，NMI)在 CORTEXM0INTEGRATION 层上有接口，SoC 设计者可以将任何外接信号与该 NMI 口相连以作为中断，该中断无法被屏蔽。它的优先级仅次于复位中断，一般会将看门狗的中断或其他与芯片错误、异常有关的中断接到 NMI 口上，若没有这种特殊需求也可以下拉为 0。

HardFault 中断常在程序运行过程中发生寻址错误时触发。它的优先级仅次于 NMI 中断。它在 CORTEXM0INTEGRATION 层没有接口，属于内部中断。

SVC(Super Visor Call)中断是用户使用 SVC 指令从而人为生成的中断。它在操作系统中常用，为应用程序提供系统服务入口，在 CM0 中并不常用。它的优先级是可以配置的。在 CORTEXM0INTEGRATION 层没有 SVC 的接口，属于内部中断。

PendSV 中断是可挂起的 SVC 中断，即它虽然发出，但先挂起，待系统中完全没有中断了才会处理它。发起方式是用软件向 SCB 寄存器的 ICSR 字段比特 28 配 1。软件也可以取消挂起状态。该功能同样也常用于操作系统中，在 CM0 中并不常用。它的优先级是可以配置的。在 CORTEXM0INTEGRATION 层没有 SVC 的接口，属于内部中断。

SysTick 是 CM0 核内部的一个计时器，如图 5-15 所示。它是倒计时的，当计数到 0 时，

会发起中断。SysTick 可用于操作系统,作为系统的统一时钟使用,而在无操作系统的应用中,由于它的驱动时钟源可以是外部固定频率时钟,也可以用它作为休眠唤醒时钟使用。它的优先级是可配置的。在 CORTEXM0INTEGRATION 层没有 SysTick 中断的接口,属于内部中断。

综上,SoC 系统集成者所看到的层次一般是 CORTEXM0INTEGRATION 层。若该层没有中断接口,则该中断称为内部中断,即 CM0 自己产生的中断。如果有接口,设计者则可以将信号线引到接口上,即为外部中断。

CM0 中所有中断的中断号和优先级见表 5-1。读者应特别注意区分中断号和优先级号。中断号是一个中断在 CPU 内部区别于其他中断的标志,它不体现中断的优先顺序,仅仅是个编号,而优先级号是规定了 CPU 处理中断的优先级顺序,几个中断可以拥有相同的优先级。编号越低,优先级越高。普通中断的优先级从 0 到 3,而 3 个特殊中断(复位中断、NMI 中断、HardFault 中断)优先级是负数,说明优先级更高。

表 5-1 中断的中断号和优先级

中断类型	中断号	优先级号	是否为外部中断
复位中断	−15	−3	否
NMI 中断	−14	−2	是
HardFault 中断	−13	−1	否
SVC 中断	−5	0～3 可配	否
PendSV 中断	−2	0～3 可配	否
SysTick 中断	−1	0～3 可配	否
32 路普通中断	0～31	0～3 可配	是

中断信号,不论是由 SoC 集成者引入的外部信号,还是由 CPU 内部产生的,也可称为中断请求信号(Interrupt Request,IRQ)。中断的目的是使 CPU 中止正在进行的工作,而将工作转移到处理中断上去,因此,中断代表的是一些紧急的、突发的、无法事先预料的工作,例如手指敲击键盘的行为会引发中断,因为 CPU 事先无法预料手指何时会敲下键盘。由中断引发的新工作称为中断服务程序(Interrupt Handler 或 Interrupt Service Route,ISR)。每个中断都对应一个中断服务程序,因而 CM0 中可以有 38 个中断服务程序,它们可以看作子函数,只不过调用方式不是普通软件调用子函数的方式,而是通过中断信号线直接驱动调用。正是因为中断服务程序的特殊性,才使 CM0 分为线程模式和中断模式。中断服务程序和主程序一同被编译到镜像文件中,分布在该文件的不同地址上,如何让 CPU 方便地找到对应的中断服务程序是一个问题。在 CM0 中,用中断向量表可以解决该问题。中断向量表是一张表格,它记录了每个中断的中断服务程序在镜像文件中的起始地址。CPU 获得中断后,只需找到中断向量表中对应的地址,并将该地址写入 PC 中,让取指令地址从该 PC 开始。中断向量表生成后,保存于镜像文件的头部,其具体地址和内容见表 5-2。可见,CM0 中断向量表的大小是 192 字节。图 5-10 就是表 5-2 中地址 0 的 MSP 和地址 4 的复位中断服务程序地址的一个具体例子。需要注意的是,中断向量表也可以不是 192 字节,因为

中断的数量在 RTL 中是可以配置的，如果没有用到这么多中断，也就没必要提供这么多的中断服务向量了。

表 5-2 镜像文件中的中断向量表

镜像文件的地址(单位：字节)	内 容
0x0	MSP 的初始值(主栈地址，指向的不是镜像文件的地址，而是 SRAM 的地址)
0x4	复位中断服务程序在镜像文件中的地址
0x8	NMI 中断服务程序在镜像文件中的地址
0xC	HardFault 中断服务程序在镜像文件中的地址
0x2C	SVC 中断服务程序在镜像文件中的地址
0x38	PendSV 中断服务程序在镜像文件中的地址
0x3C	SysTick 中断服务程序在镜像文件中的地址
0x40	普通中断 0 服务程序在镜像文件中的地址
0x44	普通中断 1 服务程序在镜像文件中的地址
0x48	普通中断 2 服务程序在镜像文件中的地址
0x4C	普通中断 3 服务程序在镜像文件中的地址
0x50	普通中断 4 服务程序在镜像文件中的地址
0x54	普通中断 5 服务程序在镜像文件中的地址
0x58	普通中断 6 服务程序在镜像文件中的地址
0x5C	普通中断 7 服务程序在镜像文件中的地址
0x60	普通中断 8 服务程序在镜像文件中的地址
0x64	普通中断 9 服务程序在镜像文件中的地址
0x68	普通中断 10 服务程序在镜像文件中的地址
0x6C	普通中断 11 服务程序在镜像文件中的地址
0x70	普通中断 12 服务程序在镜像文件中的地址
0x74	普通中断 13 服务程序在镜像文件中的地址
0x78	普通中断 14 服务程序在镜像文件中的地址
0x7C	普通中断 15 服务程序在镜像文件中的地址
0x80	普通中断 16 服务程序在镜像文件中的地址
0x84	普通中断 17 服务程序在镜像文件中的地址
0x88	普通中断 18 服务程序在镜像文件中的地址
0x8C	普通中断 19 服务程序在镜像文件中的地址
0x90	普通中断 20 服务程序在镜像文件中的地址
0x94	普通中断 21 服务程序在镜像文件中的地址
0x98	普通中断 22 服务程序在镜像文件中的地址
0x9C	普通中断 23 服务程序在镜像文件中的地址
0xA0	普通中断 24 服务程序在镜像文件中的地址
0xA4	普通中断 25 服务程序在镜像文件中的地址
0xA8	普通中断 26 服务程序在镜像文件中的地址
0xAC	普通中断 27 服务程序在镜像文件中的地址
0xB0	普通中断 28 服务程序在镜像文件中的地址

续表

镜像文件的地址(单位：字节)	内　　容
0xB4	普通中断 29 服务程序在镜像文件中的地址
0xB8	普通中断 30 服务程序在镜像文件中的地址
0xBC	普通中断 31 服务程序在镜像文件中的地址

需要特别注意的是，CM0 在读取中断向量表时，用的是简单的偏移地址。例如，普通中断 31 进入了 CPU，导致 CPU 读中断向量表，此时，CPU 发出的 AHB 总线地址为 0xBC，即使该镜像文件的程序在 Keil 编译时用的是绝对地址，例如 0x20000000 这样的地址，CPU 仍然不会使用 0x200000BC 来寻址。在 Keil 中使用绝对地址编译，只能使表 5-2 第 2 列以绝对地址呈现。这一性质要求软件人员必须关注当前运行程序的介质是否与重映射的编号对应，即当前运行程序的介质是否为默认存储介质。例如，当前运行 App 程序，该程序运行在 NVM 上，NVM 的绝对地址假设是 0x02000000，并且假设当前重映射编号指向的默认地址也是该 NVM(详情参见图 5-7 及其解释)，则当中断发生时，CPU 通过偏移地址(例如 0xBC)也能顺利获得中断服务程序的入口地址，但当重映射编号指向的默认地址不是 NVM，而是 SRAM(假设其绝对地址是 0x20000000)时，CPU 会从 SRAM 读取中断向量，而 SRAM 中并没有镜像文件，于是 CPU 会发生运行错误。解决方法是在默认地址改变之后，将中断向量表复制一份到 SRAM 中，这样，CPU 就能够从 SRAM 中读到中断向量了。读者可能会疑惑为什么会出现镜像文件运行地址与默认地址不一致的情况。该情况主要发生在一些特殊应用场景下，例如，某种 NVM 在低功耗休眠状态下需要 CPU 发出一些命令才能使它恢复工作。默认情况下，程序在 NVM 中执行，但在从休眠到唤醒的过程中，CPU 必须先被中断唤醒，然后向 NVM 发一些命令让它恢复。此时，用于唤醒的中断服务程序存储在能够及时唤醒的 SRAM 中，中断向量表既在 NVM 中有一份，又在 SRAM 中有它的复制版本，在休眠之前，必须将 SRAM 作为默认设备，这样才能从 SRAM 中唤醒，否则 CPU 将向 NVM 索取中断向量，而 NVM 此时不可用，最终会导致程序挂死。

5.12　SCS 配置

在 5.10 节中已经介绍过 CM0 内部的寄存器组和 3 个特殊寄存器。在 CM0 的文献中，还常常出现系统控制空间(System Control Space，SCS)等概念，容易与寄存器组和特殊寄存器相混淆。实际上，SCS 也是由一些可配置的寄存器和一些状态寄存器组成的，它不在 CM0 核中，而是在图 5-15 中的 AHB 内部连接矩阵中，因此不能像寄存器组及 3 个特殊寄存器一样仅用自己的名称作为地址，而是在 AHB 总线上分配它的地址，其固定地址一般是 0xE000E000。该空间不是某一单独设备的配置，而是集中了 CORTEXM0 层可以通过 AHB 总线配置的所有寄存器，主要包括 3 个功能性寄存器组，分别为 SCB、NVIC、SysTick，它们的绝对地址见表 5-3。

系统控制块(System Control Block，SCB)可以控制或观察 NMI、SysTick、SVC、PendSV

表 5-3 SCS 内部分组及其地址

AHB 地址	功能寄存器组	AHB 地址	功能寄存器组
0xE000E010	SysTick 寄存器组	0xE000ED00	SCB 寄存器组
0xE000E100	NVIC 寄存器组		

等中断的状态，也可以对 CPU 休眠的策略进行控制。

NVIC 只能控制 32 个普通中断，因此，SCB 和 NIVC 联合起来才能完成对全部中断的配置。NVIC 的配置和状态没有放在 NVIC 器件本身，而是被放置在 SCS 中，这样可以在 NVIC 硬件中省去对 AHB 协议的解析电路。SCS 接受用户的 AHB 配置后，转换为实际的配置字，通过信号线直接配置到 NVIC 中。

SysTick 寄存器用于配置 CM0 核中的 SysTick 器件，并读取其状态，其功能在 SysTick 一节再详述。

SCB 的具体地址和含义见表 5-4。

表 5-4 SCB 具体地址及含义

AHB 地址	寄存器名	段　　名	比特位置	说　　明
0xE000ED00	CPUID	CPUID	31:0	在 CM0 中固定为 0x410CC200
0xE000ED04	ICSR	VECTACTIVE	8:0	返回当前正在运行的中断服务程序的中断号，功能与特殊寄存器 IPSR 相同
		VECTPENDING	20:12	返回当前处于挂起状态的中断号。若同时有多个中断挂起，则显示优先级最高的那个
		ISRPENDING	22	返回当前是否有中断挂起的状态
		ISRPREEMPT	23	一个挂起的中断将在下一步进入中断服务（用于单步执行时的调试目的）
		PENDSTCLR	25	取消 SysTick 中断的挂起状态
		PENDSTSET	26	制造一个 SysTick 中断，并挂起
		PENDSVCLR	27	取消 PendSV 中断的挂起状态
		PENDSVSET	28	制造一个 PendSV 中断，并挂起
		NMIPENDSET	31	制造 NMI 中断，并挂起
0xE000ED0C	AIRCR	VECTCLRACTIVE	1	清除已挂起的所有中断
		SYSRESETREQ	2	配置它可以制造一个 CPU 复位信号
		ENDIANESS	15	返回 CPU 的大小端模式，0 为小端，1 为大端。注意，这里不是配置，仅为状态的返回值，配置大小端在例化 CM0 的 parameter 中已经决定
		VECTOR_KEY	31:16	正常值是 0xFA05，若配成 0x05FA，并将 SYSRESETREQ 字段配置为 1，则会导致 CPU 复位

续表

AHB 地址	寄存器名	段　　名	比特位置	说　　明
0xE000ED10	SCR	SLEEPONEXIT	1	设置 CPU 是否可以自动进入休眠 0：不自动进入休眠，再次休眠需要用户再发命令； 1：处理完中断服务程序后自动进入休眠
		SLEEPDEEP	2	设置 CPU 睡眠深浅 0：一旦休眠将进入浅睡模式； 1：一旦休眠将进入深睡模式
		SEVONPEND	4	设置中断唤醒策略 0：只有使能的中断可以唤醒 CPU； 1：任何中断都能唤醒 CPU
0xE000ED14	CCR	UNALIGN_TRP	3	当调试时，若遇到地址不对齐的情况，就会返回一个错误。用于发现不对齐的位置
		STKALIGN	9	设置进入中断后栈的对齐方式 0：4 字节对齐； 1：8 字节对齐
0xE000ED1C	SHP[0]	SHP[0]	31:24	设置 SVC 中断的优先级
0xE000ED20	SHP[1]	PENDSVPRO	23:16	设置 PendSV 中断的优先级
		STPRO	31:24	设置 SysTick 中断的优先级
0xE000ED24	SHCSR	svcallpended	15	反馈 SVC 中断挂起标志

NVIC 的具体地址和含义见表 5-5，它只负责控制 32 路普通外部中断。

表 5-5　NVIC 具体地址及含义

AHB 地址	寄存器名	段　　名	比特位置	说　　明
0xE000E100	ISER[0]	ISER[0]	31:0	设置 32 路中断使能
0xE000E180	ICER[0]	ICER[0]	31:0	取消 32 路中断使能
0xE000E200	ISPR[0]	ISPR[0]	31:0	可以读出 32 路中断的挂起状态，也可以手动将某个中断挂起
0xE000E280	ICPR[0]	ICPR[0]	31:0	取消 32 路中断的挂起状态
0xE000E400	IP[0]	P0	7:6	设置中断 0 的优先级
		P1	15:14	设置中断 1 的优先级
		P2	23:22	设置中断 2 的优先级
		P3	31:30	设置中断 3 的优先级
0xE000E404	IP[1]	P4	7:6	设置中断 4 的优先级
		P5	15:14	设置中断 5 的优先级
		P6	23:22	设置中断 6 的优先级
		P7	31:30	设置中断 7 的优先级

续表

AHB 地址	寄存器名	段　　名	比特位置	说　　明
0xE000E408	IP[2]	P8	7:6	设置中断 8 的优先级
		P9	15:14	设置中断 9 的优先级
		P10	23:22	设置中断 10 的优先级
		P11	31:30	设置中断 11 的优先级
0xE000E40C	IP[3]	P12	7:6	设置中断 12 的优先级
		P13	15:14	设置中断 13 的优先级
		P14	23:22	设置中断 14 的优先级
		P15	31:30	设置中断 15 的优先级
0xE000E410	IP[4]	P16	7:6	设置中断 16 的优先级
		P17	15:14	设置中断 17 的优先级
		P18	23:22	设置中断 18 的优先级
		P19	31:30	设置中断 19 的优先级
0xE000E414	IP[5]	P20	7:6	设置中断 20 的优先级
		P21	15:14	设置中断 21 的优先级
		P22	23:22	设置中断 22 的优先级
		P23	31:30	设置中断 23 的优先级
0xE000E418	IP[6]	P24	7:6	设置中断 24 的优先级
		P25	15:14	设置中断 25 的优先级
		P26	23:22	设置中断 26 的优先级
		P27	31:30	设置中断 27 的优先级
0xE000E41C	IP[7]	P28	7:6	设置中断 28 的优先级
		P29	15:14	设置中断 29 的优先级
		P30	23:22	设置中断 30 的优先级
		P31	31:30	设置中断 31 的优先级

5.13 ARM Cortex-M0 的集成

用户在集成 CM0 时，可以根据自身的需求，选择例化 CORTEXM0 或者 CORTEXM0INTEGRATION。对于无特殊需求的普通用户，建议直接例化 CORTEXM0INTEGRATION 层。

一个 CM0 的简单集成示例如图 5-16 所示。除了直接例化的 CORTEXM0INTEGRATION 层外，在总线上还挂有必要的各种外设。这些外设都包含一个统一的前缀 cmsdk，这是 ARM 在 CM0 之外免费提供的一套总线设备框架，用户可以在下载的文件包中找到。

CM0 支持的总线是 AHB-Lite 总线。ROM_wrapper、SRAM_wrapper、cmsdk_ahb_gpio、cmsdk_ahb_cs_rom_table、cmsdk_mcu_sysCtrl、cmsdk_ahb_default_slave 是挂在

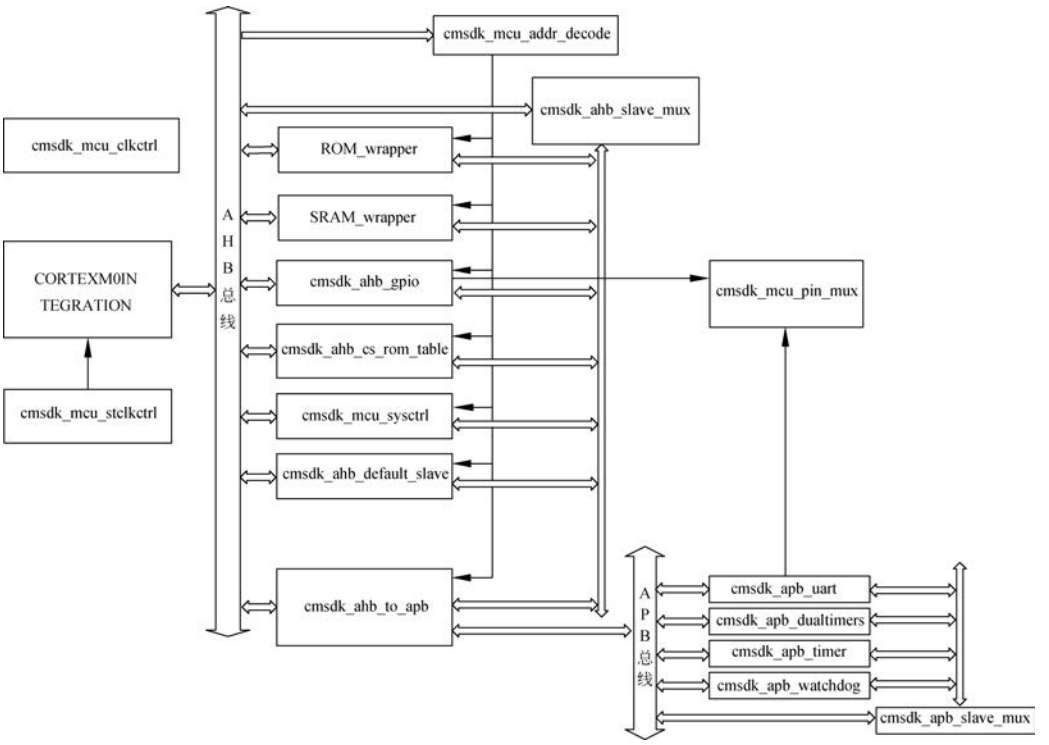


图 5-16　CM0 集成示例

AHB 总线上的设备,其中,ROM_wrapper 是 ROM 的顶层,因为按照 5.3 节介绍的方法所生成的 ROM IP 不包含 AHB 接口,因此需要在其上再包一个顶层,称为 wrapper。同理,SRAM_wrapper 内部也包含 SRAM IP。在 DesignStart 的文件包中,也可以找到名为 cmsdk_ahb_rom 的 ROM 和名为 cmsdk_ahb_ram 的 SRAM,但它们都是模型,所谓模型就是无法映射到实际电路,仅能用于仿真,也称为仿真模型。使用模型,可以使设计者搭建一个完整系统,从而验证系统中真正属于 RTL 部分的功能。在真正流片时,ROM 和 SRAM 本身不使用 Verilog 描述,而是用工具直接生成物理实体放置在版图上。本套系统也可以在 FPGA 中实现,此时,ROM 和 SRAM 需要用 FPGA 的软件开发工具生成,并例化在 RTL 里。cmsdk_ahb_gpio 是 GPIO 设备,它提供电平的输入、输出功能。cmsdk_ahb_cs_rom_table 是 ROM Table,当一款 ARM CPU 与外部仿真器(例如 JLink 和 ULink)相连并进行程序的下载、单步运行等 Debug 操作时,仿真器会先读该 CPU 的 0xF0000000 地址,即 ROM Table 地址,提取其内部信息,相当于对 CPU 进行认证,只有认证通过的 CPU 才认为是可以连接的。由于 ROM Table 设备的内容是固定在硬件上的,对于软件来讲是只读的,因此也是一个 ROM。ROM Table 的不同字段有不同含义,但对于集成来讲了解这些含义并非必要,因此这里不再赘述。cmsdk_mcu_sysCtrl 的主要功能有两个,一个是重映射序号的配置在这里,配置后,序号会通过直接连线进入 cmsdk_mcu_addr_decode 中,作为默认设

备选择的依据(其中的机制详见5.5节),第2个功能是当系统发生意外复位时,可以读取其中的复位原因寄存器,获取刚才意外发生的原因。该寄存器没有用总线时钟驱动,因而不会因总线复位而被复位。复位的原因有3种,分别是CPU内部LockUp复位、看门狗复位、软件主动配置复位。cmsdk_ahb_default_slave充当AHB总线上的默认设备。因为AHB总线的位宽是32位,即允许有4GB寻址空间,实际中各设备无法占满这些空间,必定有一些地址没有对应任何外设,此时,在设计上有两种选择,一种是当软件访问无意义的地址时置之不理,另一种是将这些地址统一收集到AHB默认设备中,以统一管理和统一回复,推荐使用第2种方法。cmsdk_mcu_addr_decode和cmsdk_ahb_slave_mux不是总线设备,而是总线管理的辅助,其中,cmsdk_mcu_addr_decode对应5.5节中的地址解析器,负责根据AHB地址选择当前操作的设备。cmsdk_ahb_slave_mux负责汇总各设备的状态信息和读出数据,然后上传给CM0。

cmsdk_ahb_to_apb模块也可以看作一个AHB设备,与上述AHB设备的控制方式相同,但它实际上是一个桥接电路,即协议转换电路,负责将AHB协议翻译为APB协议。它作为APB的主控模块,上面挂着一些APB器件。图中的APB器件有cmsdk_apb_uart、cmsdk_apb_dualtimers、cmsdk_apb_timer、cmsdk_apb_watchdog。cmsdk_apb_uart是UART,即串口设备,可以通过串口打印信息或通过串口向CM0输入信息。cmsdk_apb_dualtimers是双计时器,它虽然是一个设备,但内部有两个Timer,这两个Timer可以单独使用,每个Timer的计数位宽是16位。cmsdk_apb_timer是一个单独的Timer,其计数位宽是32位。cmsdk_apb_watchdog是看门狗设备。cmsdk_apb_slave_mux不是APB总线设备,而是APB的管理辅助,它负责选择当前操作的APB设备(类似AHB中的cmsdk_mcu_addr_decode),还负责将APB设备的输出信号进行汇总,上传给CM0(类似AHB中的cmsdk_ahb_slave_MUX)。APB设备中,也可以设计一个默认设备(类似AHB中的cmsdk_ahb_default_slave),这样的设计会更为完整,图中未画出该设备是因为cmsdk未提供。

APB设备和AHB设备并没有本质区别,它们都是CM0统属的外围器件,仅在总线接口上有所区别,其内部功能是不变的。例如,图中的UART是APB设备,也可以改成AHB设备,GPIO设备也可以改成APB设备。AHB和APB的区别在于速度,AHB传输速度较快,而APB较慢。AHB在协议上比APB复杂一些,因此,对于数据带宽较窄的设备,例如UART,其波特率一般设为115.2kHz~460.8kHz,会设计为APB设备,而用于存取数据和指令的SRAM,如果也挂在APB总线上,则会浪费系统的主频。

各设备在总线上的地址可以由集成设计人员修改。AHB设备的地址可以在cmsdk_mcu_addr_decode模块中分配和修改,APB设备的地址可以在cmsdk_apb_slave_mux模块中分配和修改。在CORTEXM0INTEGRATION层次内部的地址,如5.12节讲解的SCS地址,一般不做修改。

图5-16中的cmsdk_mcu_pin_mux模块通称为Pin Mux,即引脚的汇总,汇总后集中在引脚上,使引脚成为一个功能复合体。芯片的封装大小有限,因此每个引脚都应尽量复用多个功能。具体应用时,根据芯片用户的需要,通过软件配置该引脚的功能。在图中可以看到

GPIO 和 UART 被引入了 Pin Mux 中,说明该 Pin Mux 的一个引脚既可以当作 GPIO 输入/输出口使用,又可以当作串口使用,具体取决于软件配置。问题是,软件的配置怎样进入 Pin Mux 中? 在 DesignStart 给出的设计例子中,Pin Mux 配置的提供者是 cmsdk_ahb_gpio,即 GPIO 设备。每个引脚只有两个功能,一个是 GPIO 设备接口,另一个是 UART 接口,因此只需一位配置字。在实际的设计中,Pin Mux 的功能远比图例中的情况更复杂,一个引脚可能会包含 5～10 个不同功能,需要多位配置字。一般会设计一个专门的 Pin Mux 配置设备,挂在 APB 总线上。设计专门的配置设备也是为了将 GPIO 设备和引脚进行区分,因为引脚在版图或产品说明书中也常被命名为 GPIO,实际上它的全称是 GPIO 引脚,而在设计时常称为 Pad,它的结构如图 1-3 所示。GPIO 设备和 GPIO 引脚的概念必须严格区分,否则会造成设计和配置的混乱。

图 5-16 中的 cmsdk_mcu_clkctrl 是时钟复位控制模块,负责产生整个芯片使用的时钟信号和复位信号。DesignStart 提供的模块是一个简化版本,整个系统使用统一的时钟和复位,由总线时钟驱动总线上的每个设备,仅 cmsdk_mcu_sysctrl 模块的复位原因寄存器使用非总线时钟,以避免总线复位后原因编号消失。

图 5-16 中的 cmsdk_mcu_stclkctrl 模块是一个简化的 SysTick 时钟控制和校准模块,将其输入给 CM0,以驱动 SysTick 工作。

图 5-16 概括了 DesignStart 中提供的 cmsdk_mcu 模块的内部结构。使用者可以选择手动例化并连接图中各模块,也可以直接例化它们的顶层,即 cmsdk_mcu。一个用于仿真的 CM0 SoC 系统框架如图 5-17 所示。cmsdk_mcu 代表一个完整的基于 CM0 的 SoC 设计,其中包含了 cmsdk_mcu_system、cmsdk_mcu_pin_mux、cmsdk_ahb_rom、cmsdk_ahb_ram、cmsdk_mcu_clkctrl。cmsdk_mcu_system 包含了大部分 SoC 组件,而另外 4 个模块之所以不包含在 cmsdk_mcu_system 之内,是因为它们属于模型性质,在设计芯片或 FPGA 工程时,需要进行调整,ROM 和 RAM 需要生成,Pin Mux 需要例化 I/O 器件,时钟复位模

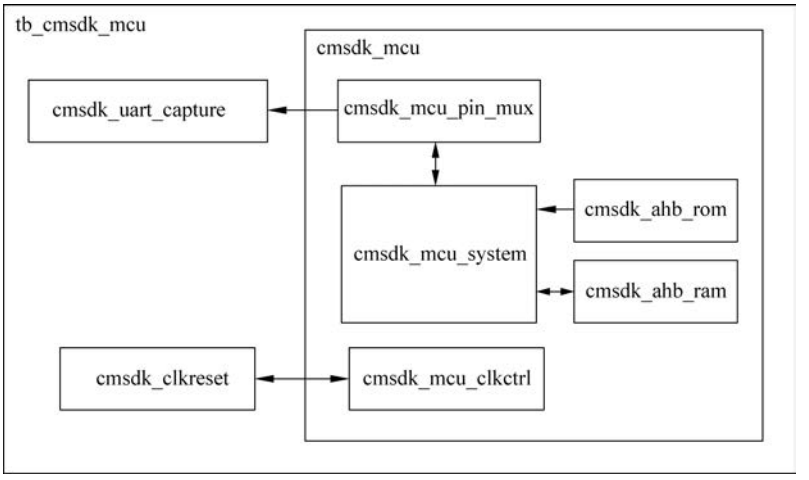


图 5-17　基于 CM0 的 SoC 系统框架

块需要重新设计。仿真顶层 tb_cmsdk_mcu 是以 cmsdk_mcu 或 cmsdk_mcu_system 为验证对象(DUT)从而搭建的平台,其中 cmsdk_clkreset 用常规方式生成了时钟和复位信号,cmsdk_uart_capture 是一个虚拟的 UART 的接收设备,可以将 cmsdk_mcu 输出的串口信息打印在仿真平台中。

外部中断信号线共有 33 根,其中 32 根为普通外部中断线,一根为 NMI 中断线。它们在 cmsdk_mcu_system 这一层次进行连接。中断信号线的连接方式如图 5-18 所示。在图 5-16 的设备中,只有 GPIO、UART、Dual Timer、普通 Timer、看门狗会起中断,其中,GPIO 的中断数量等于 GPIO 引脚数量。中断的连接不需要任何中介,直接从设备引到 CPU 顶层的 NMI 口或普通中断口即可。中断线在 CORTEXM0INTEGRATION 内部的运行方式详见 5.11 节。推荐将看门狗中断接入 NMI 口,而将其他中断接入普通外部中断口。

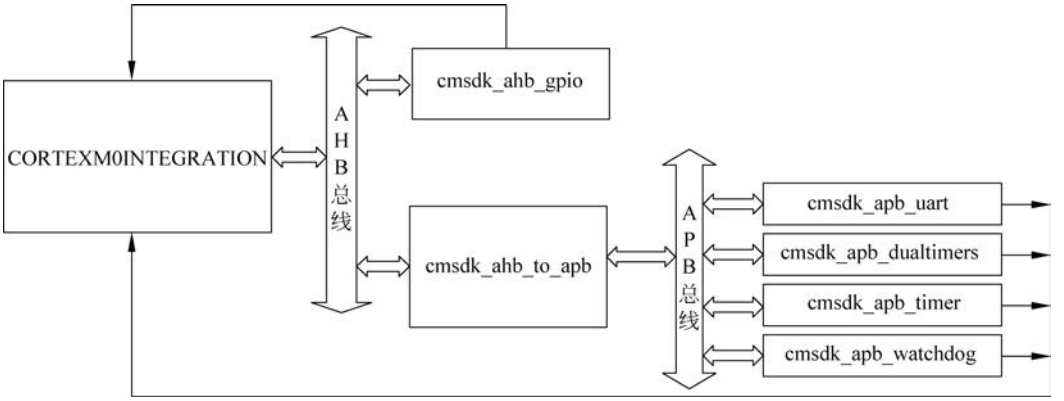


图 5-18 中断信号线的连接方式

在完整版的 CORTEXM0INTEGRATION 中会有下列设置项,即 parameter,需要集成设计人员进行配置,具体见表 5-6。需要注意的是 BE 所代表的大端模式和小端模式。大端模式(Big Endian)使用低地址存储高位字节,并且使用高地址存储低位字节。小端模式(Little Endian)正好相反,使用低地址存储低位字节,并且使用高地址存储高位字节。例如,在 C 代码中声明一个 unsigned int aa,数值为 0x12345678。如果采用大端模式,则 0x12 存储在地址 0,0x34 存储在地址 1,0x56 存储在地址 2,0x78 存储在地址 3。如果采用小端模式,则 0x12 存储在地址 3,0x34 存储在地址 2,0x56 存储在地址 1,0x78 存储在地址 0。软件在编程时,需要知道其 CPU 的大小端,然后才能进行正确的字节操作,例如软件想取数据 0x56,在大端模式下,需要去地址 2 中取,在小端模式下则去地址 1 中取。大小端在 ARM 提供的软件架构 CMSIS 中也有设置,很多软件人员会误以为是软件决定了 CPU 的大小端,实际上,在 SoC 集成时,大小端就已经通过 BE 确定了,通过软件无法修改,软件只是在已知大小端的基础上做出程序调整而已。在第 5.12 节的 SCB 寄存器中也有一个寄存器 ENDIANESS 可以反映大小端,但它是只读的,真正的根源还是在 BE 的设置上。

表 5-6 CM0 参数配置表

参 数 名	定 义
ACG	是否使用 CM0 内部时钟门控。0 表示不用门控,1 表示使用。一般选择使用
BE	选择 CPU 工作在大端模式还是小端模式。0 表示小端模式,1 表示大端模式。一般选择小端模式
DBG	是否使用 CM0 中的 Debug 模块。0 表示不使用,1 表示使用。当不使用时,将不支持仿真器代码下载及单步运行等 Debug 功能。一般选择使用
JTAGnSW	选择 Debug 接口类型。0 表示 SWD 接口,1 表示 JTAG 接口。SWD 接口连接简单,速度快,目前常用 SWD 接口
RAR	如果发生了 CPU 复位,则选择复位的波及范围。0 表示仅部分寄存器会复位,1 表示全部寄存器都复位。一般会选择 1
SMUL	选择内部乘法器的结构。0 表示运算快,但面积大。1 表示运算慢,但面积小。该选择根据实际应用而定
SYST	选择是否包含 SysTick 模块。0 表示不包含,1 表示包含。SysTick 无论在有无操作系统的应用中都有其作用,一般选择包含
WIC	选择是否包含 WIC 模块。0 表示不包含,1 表示包含。WIC 在 CM0 进入深度休眠后的唤醒中扮演着重要角色,而省电功能是 SoC 设计的必备,因此一般会选择包含
NUMIRQ	填写中断线的数量。最大是 32 根外部中断线,也可以减少。一般填写 32
WICLINES	填写引入 WIC 的中断线数量,这里比 NUMIRQ 多两根,它们是 NMI 中断和 RXEV 事件,其中的 RXEV 是在多核处理器中另一个核向本核发来的唤醒脉冲信号。在 CM0 系统中不常用。如果 NUMIRQ 填 32,则这里填 34
BKPT	填写断点的数量(Debug 器件中的配置)。CM0 最多支持 4 个断点,也可以写 0,即不支持断点
WPT	填写观察点数量(Debug 器件中的配置)。CM0 最多支持两个观察点,也可以写 0,即不支持观察点。观察点即 IDE 界面中常见的 watch 窗口,可以加入变量,以便实时观察变量的变化

5.14 通过软件验证设计

在完成了 SoC 的集成工作后还需要对设计进行仿真,即如图 5-17 那样搭建一个测试平台(TB)。SoC 芯片的验证一般通过两种方式,一种是传统的验证方法,即搭建 TB,在 TB 中加入激励,驱动 DUT 运行,并观察结果,同时在 TB 中建立参考模型,将 DUT 的输出结果与参考模型进行对照,另一种是编写嵌入式软件,将软件烧写到 NVM 中(其实就是在 TB 的开始阶段用 $readmemh 或 $readmemb 命令将软件编译得到的 HEX 文件录入 NVM 模型中),然后 CPU 会按照正常运行的方式逐条取指令。

在传统验证方法中,CPU 无法取指令并正常运行,因为用 SV 不容易表征复杂的软件运行过程。另外,SoC 的外设多种多样,需要搭建不同种类的参考模型用于验证对比,因而,该方法只适合于单一设备的验证,即将 SoC 设备进行拆分,为每个模块或每个设备搭建一套单独的 TB。在 TB 中用 SV 建立一个 AHB 或 APB 的机制,向设备中灌激励,同时监控

设备的输出。

对于 SoC 集成的整体验证,需要使用软件方式。驱动通过软件指令提供,参考模型可以在软件中实现,也可以在 TB 中实现,根据验证的需要而定。

具体的软件运行步骤可参见第 5.5 节。本节将介绍 DesignStart 中提供的软件架构,以便读者可以据此编写出可用的 Boot 程序和 App 程序。

DesignStart 中的软件框架称为 CMSIS,它提供了 cmsdk_mcu 中各设备的驱动,它所包含的文件名及作用见表 5-7。

表 5-7 CMSIS 中的文件名及作用

文 件 名	作 用
CMSDK_CM0.h	(1) 登记中断序号,以便对 NVIM 进行操作,灵活开关,该序号需要根据 cmsdk_mcu_system 内中断线的连接顺序排列。 (2) 声明各外围设备的结构体,以便随时调用 登记各外围设备的 AHB 基地址,若在 RTL 中修改了基地址,例如修改了 cmsdk_mcu_addr_decode 或 cmsdk_apb_slave_mux,则需要在本文件中进行相应修改
CMSDK_driver.h	外围设备的 SDK 函数声明
system_CMSDK_CM0.h	系统初始化相关函数声明
core_cm0.h	对 3 个特殊寄存器,以及 SCS 中 SCB、NVIC、SysTick 相关寄存器的结构体声明,还包含 SCS 中 SCB、NVIC、SysTick 应用函数
core_cmFunc.h	3 个特殊寄存器的应用函数
core_cmInstr.h	一些汇编指令的 C 语言接口,例如常用的__WFI()命令和__ISB()等
startup_CMSDK_CM0.s	使用汇编语言进行系统配置,包括以下几点: (1) 堆和栈的大小设置。 (2) 中断向量表中不同中断向量的位置和中断向量表的大小。如果修改了 RTL 里中断的顺序,在软件上不仅要修改 CMSDK_CM0.h,还要修改本文件。 (3) 登记中断服务程序的名称,默认为 Handler,允许修改。 规定中断向量的执行顺序,特别是规定了启动流程,即 Reset_Handler 的执行流程
system_CMSDK_CM0.c	系统初始化相关函数,它的头文件是 system_CMSDK_CM0.h
CMSDK_driver.c	外围设备,如 UART、Timer、GPIO 等的驱动函数。它的头文件是 CMSDK_driver.h

注意 软件 SDK 是 SoC 芯片设计厂商向芯片用户提供的软件驱动代码,它包含芯片中每个设备的寄存器功能,以及操作流程控制。它是软件应用的基础,用户调用 SDK 进行编程,可以避免讨论与硬件相关的问题,把精力更多地集中在应用的实现上。SDK 一方面可以用于解决底层硬件问题,另一方面还可以在芯片有设计缺陷时,在芯片厂商内部通过 SDK 屏蔽,从而避免将缺陷泄露给芯片用户。

除了 CMSIS 提供的 SDK 外,DesignStart 还提供了一些示例程序。它们以 CMSIS 为基础编写而成。具体文件和功能见表 5-8。这里需要解释一下 retarget 的含义。通常,当在计算机上编程并在计算机的屏幕上显示时,C 语言程序都使用 printf 进行打印,会将文字打印在屏幕上,如果要打印在文件中,则使用 fprintf 函数,CPU 将直接操作显示设备或文件设备从而完成打印,但是在 SoC 芯片嵌入式系统中,C 语言命令 CPU 打印,其实是想操作 UART 设备,让它发出串口协议电平,这就不同于普通意义上的打印了。因而,将芯片通过 UART 打印到计算机上的过程称为打印的重定向(Retarget)。

表 5-8 DesignStart 中提供的示例程序

文 件 名	作 用
bootloader. c	BOOT 程序示例
dualtimer_demo. c	Dual Timer 应用示例
interrupt_demo. c	中断应用示例
self_reset_demo. c	带电热复位示例
sleep_demo. c	CPU 休眠示例
watchdog_demo. c	看门狗应用示例
retarget. c uart_stdout. c uart_stdout. h	UART 串口打印和接收函数
apb_mux_tests. c	APB 设备选择测试程序
default_slaves_tests. c	AHB 默认设备测试程序
gpio_driver_tests. c gpio_tests	GPIO 设备相关测试
memory_tests. c	测试 SRAM、ROM、ROM Table 等存储设备的功能
timer_driver_tests. c timer_tests. c	Timer 相关测试
uart_driver_tests. c uart_tests. c	UART 相关测试

5.15 产品级芯片集成

上述集成步骤只适用于 CM0 的学习者和评估者,而如果想以 CM0 为核心打造一款 SoC 芯片,则需要在原有平台上进行诸多改进,主要列举如下:

(1) 产品级 SoC 芯片的时钟复位策略是非常复杂的,cmsdk_mcu_clkCtrl 中简单的时钟门控关系远远不能满足设计需求。SoC 芯片上电时钟和复位的设计详见 5.20 节。

(2) 存储器,包括 SRAM、ROM、NVM 等,需要使用 IP 生成器产生,并放在实际电路版图中,同时要生成一份仿真用的 Verilog 模型代码。设计者需要将它们再包一层,形成 wrapper,将 AHB 接口协议转换为 IP 可以识别的数据流。

(3) GPIO 引脚的数量需要根据实际封装大小和引脚数量进行规划。用户往往会要求

更多的 GPIO 数量以满足其各式各样的产品需求。GPIO 引脚数量将决定 GPIO 设备的输入/输出信号数量，设计者应据此修改 cmsdk_ahb_gpio 中的 GPIO 数量。

(4) 在产品中，一个 GPIO 引脚上复用的功能很多，除了 GPIO 设备本身的输入/输出信号外，还有 UART、I^2C、SPI、PWM 等多种信号复用其上，有时，模拟信号也会通过该引脚进行输入和输出，因此，cmsdk_mcu_pin_mux 不能满足实际需求。需要新建一个专属配置模块，对 Pin Mux 的功能进行配置。由于产品电路板一旦定型，Pin Mux 的设置就不会改变，因此它的配置可以使用慢速的 APB 总线完成。将需要与芯片外界沟通的信号线都接到该 Pin Mux 上。在其内部，需要例化芯片 I/O 器件，它是既支持输入又支持输出的元器件，需要在不同功能模式下对 I/O 的方向进行选通。

(5) 一般会在系统中增加 I^2C、SPI、PWM 等设备，而且数量不一定只有一个。例如配备两个 I^2C 或两个 UART 的 SoC 芯片很常见。提供的资源越多，产品开发的灵活度就越高，譬如两个 UART，可以允许用户使用其中一个进行人机交互，而另一个可以与其他芯片相连，进行单线数据传输，因此，需要芯片规划者在面积成本和挂载更多的设备之间做出平衡。这些设备使用的总线根据用户对速度的要求而定。在没有特殊要求的情况下，可以挂在 APB 总线上。

(6) SoC 设计中会有多个计时器，以便软件在实现产品功能时可以对不同的事件进行定时。例如，需要用软件实现某种通信协议，在发起一种通信帧时，开启一个计时器 A，在发起另一种通信帧时，需要再开启一个计时器 B，并保持计时器 A 的计时动作不停止。那么这种需求就需要两个计时器。当然，软件本身也有处理此问题的对策，例如用 for 循环代替硬件定时，或者只用一个计时器，每计时一小段时间，就打一个时戳，依靠这个时戳可以声明多个虚拟计时器，以满足多计时器的需求，但这些软件方式的定时都不准确，而且还容易被中断打扰，进一步增加了定时误差，因而对于时间要求严格的应用无法使用。硬件计时器的种类多样。普通计时器以总线时钟驱动，计数位宽根据需求制定，结构简单、面积小，其缺点之一是当位宽不满足要求时无法更改以适应新需求，因为一个固定计数位宽的计数器，在时钟频率固定的前提下，它的最长计时长度是固定的，缺点之二是一旦总线时钟频率变化，例如 CPU 进入休眠，则它的计数频率也会变化，因而无法做到任何情况下都准确计数。为了弥补第 1 个缺陷，可以在计时器中使用预分频，即驱动时钟进入计时器后，不直接驱动计时，而是先分频，根据需要指定分频比，然后以分频后的时钟来计数，如此，计时器的计时长度就是可调的，当需要更长时间计时时，可以将分频比调大，也可以考虑缩短计数位宽，以节省面积。为了弥补第 2 个缺陷，常常会引入一个固定的时钟源来驱动计时器，这样就不会因总线时钟频率变化而导致计时周期变化，但这会增加跨时钟握手电路，从而增加模块面积，这样的计时器称为实时时钟(Real Time Clock，RTC)。PWM 本质上也是计时器，因此，设计完的计时器还经常会集成 PWM 发生器功能，并且产生成对的互补 PWM，以满足驱动大功率晶体管的需要。计时器也可以做输入脉冲跨度计时，如果芯片应用中关心 GPIO 引脚上输入信号的脉冲宽度，则可以将宽度计时功能融入计时器设计中。对上述多种特性进行排列组合，能够得到不同种类的计时器。在 SoC 设计中，各种计时器都会挂载一些，以满足不同

的用户需求。

(7) 中断并非要将 32 个都占满,需要根据应用的需要而定。有时,GPIO 引脚数量众多,为每个 GPIO 都设置独立中断可能无法做到,此时,可以将 GPIO 合并为若干组,组内使用一个中断。当 CPU 处理中断时,再根据内部寄存器区分中断原因。其他中断的合并原则也类似于 GPIO,例如 UART 有 TX 和 RX 中断,可以合并为 UART 统一中断。另外,看门狗也可以取消中断,因为喂狗动作就是配置看门狗的寄存器,一旦停止喂,就会导致系统复位,此逻辑链条与中断无关。看门狗本身是为了检测出 CPU 死机,有些死机情况是中断正常处理,而非中断不能正常运行,此时,通过中断喂狗的方式检测不出 CPU 死机的状态。

(8) SoC 芯片分为两个大类,一类是通用型 MCU,另一类是专用芯片。对于专用芯片,往往需要设计专用的设备,例如 WiFi 通信芯片,需要设计 WiFi 模块,并挂接在总线上。这种专用模块的复杂度和面积有时甚至会超过芯片本身的 CPU。受限于设计者对应用场景的理解及芯片复杂度,专用芯片往往要解决应用来实现方案的分工问题。哪些功能由软件承担?哪些功能由专用硬件承担?专用硬件和软件如何协调分工?软件规模多大,需要占用多少 RAM 和 NVM 的面积?这些都是系统规划时需要考虑的问题。一款专用芯片的设计压力,很多时候不在 SoC 系统上,而是在于上述几个问题的决策上。

5.16 AHB 总线协议

CM0 支持的总线为 AMBA3 AHB-Lite 总线,它是一种简化的 AHB 总线,主要用于单一主控的情况,即芯片中只有一个 CPU。CM0 系统简单、性能较差,一般基于它设计的芯片不会集成多核,很适合使用 AHB-Lite 这样的简单总线。

AHB-Lite 的接口信号见表 5-9。在表格中既给出了在 CPU 视角下的信号方向,又给出了在 AHB 设备视角下的信号方向,目的是便于读者分辨。总线接口主要用于定义 CPU 与设备之间的关联信号,因此,表中的大多数信号,要么是从 CPU 输出并输入设备中,要么是从设备输出并进入 CPU 中。信号 HCLK 和 HRESETn,分别是总线的时钟和复位,不论对于 CPU 还是对于设备都是输入,因为它们是来自 CPU 和设备之外的模块,即时钟复位产生模块(图 5-16 中的 cmsdk_mcu_clkCtrl 模块),不是 CPU 与设备之间的信号。HREADY 信号对于 CPU 和设备都属于输入信号,因为它的连接方式比较特殊,在下文中将详述。HREADYOUT 是每个设备的输出,它会合并为 HREADY 再输入 CPU,因而它不是 CPU 的直接输入信号。

需要特别强调的是,对于 CPU 来讲,所有设备都只是一簇簇的存储器群落,CPU 会向这些存储器中写入数据,也会从存储器中读取数据。对于设备本身实现的功能,作为硬件的 CPU 并不关心,而这些设备功能的实现,是通过设计软件流程,驱动 CPU 对设备存储器进行读写从而间接实现的,在 CPU 看来,它只是读写了存储器而已。了解了这一点,就可以理解总线的本质,其实就是 CPU 读写设备中存储器的通道。因而才会进一步理解总线中为什么会有 Bufferable 和 Cacheable 这类概念。初识总线的学习者切忌将总线理解复杂化。

表 5-9 AHB-Lite 总线接口信号

信号类型	信号名	位宽	方向（CPU 视角）	方向（设备视角）	定　义
系统信号	HCLK	1	输入	输入	总线时钟
	HRESETn	1	输入	输入	总线复位
控制信号	HTRANS	2	输出	输入	传输方式控制。 0：总线空闲。 1：总线忙。 2：只传一个数据。 3：连续传输数据
	HSIZE	3	输出	输入	数据大小控制。 0：每次传 1 字节。 1：每次传 2 字节。 2：每次传 4 字节。 3：每次传 8 字节。 4：每次传 16 字节。 5：每次传 32 字节。 6：每次传 64 字节。 7：每次传 128 字节
	HWRITE	1	输出	输入	传输方向控制。 0：CPU 读取设备。 1：CPU 向设备中写
	HPROT	4	输出	输入	数据保护策略。 第 0 位（最低位）：表示传输数据类型。为 0 时表示传输指令，为 1 时表示传输数据。 第 1 位：表示数据权限。为 0 时表示使用普通用户权限，为 1 时表示使用特权。 第 2 位：是否允许数据被 Buffer 存储和延迟。为 0 时表示不允许，为 1 时表示允许。 第 3 位（最高位）：是否允许数据被 Cache 缓存。为 0 时表示不允许，为 1 时表示允许
	HBURST	3	输出	输入	Burst 传输策略。 0：只传一个数据。 1：不定长的 Burst 传输。 2：4 次连续传输。 3：8 次连续传输。 4：16 次连续传输。 5：4 次连续 WRAP 传输。 6：8 次连续 WRAP 传输。 7：16 次连续 WRAP 传输
	HADDR	32	输出	输入	操作地址

续表

信号类型	信号名	位宽	方向(CPU视角)	方向(设备视角)	定　义
数据信号	HWDATA	32	输出	输入	待写入的数据
	HRDATA	32	输入	输出	读出的数据
反馈信号	HRESP	1	输入	输出	设备状态反馈。 0：传输成功。 1：传输中出现错误
	HREADY	1	输入	输入	全部设备整体状态反馈。 0：总线上有设备正在忙，不接受下次传输，其他设备都需要等待它空闲。 1：总线上所有设备都是空闲的
	HREADYOUT	1	/	输出	本设备状态反馈。 0：本设备正在忙，不接受下次传输。 1：本设备空闲

AHB接口信号可归类为系统信号、控制信号、数据信号、反馈信号。系统信号即时钟和复位。控制信号是CPU用来对传输进行控制的信号，它会通知设备一些信息，使设备可以按照该信息的要求与CPU进行数据交互。数据信号就是真正交流的数据。反馈信号即设备向CPU反馈的非数据信号。

CPU用HTRANS信号来通知设备会不会传输数据，以及传输多少数据。为0时表示不传数据，总线空闲(IDLE)。为1时，虽然称为总线忙(BUSY)，但实际用途是CPU在Burst传输的中间插入一个BUSY状态，表示这一拍是空闲的，但后面还要继续传输，其他控制信号不会改变。在ARM Cortex-M系列处理器中不会出现Busy状态。为2时表示传输单个数据，传完结束，称为NONSEQ或SINGLE。当设备收到NONSEQ状态后，会知道之前的任何控制状态都无效了，从本拍开始用新的控制信号。为3时表示连续传输，称为SEQ，即Burst传输。在该状态下，其他控制信号会保持，而地址会在每节拍中自增。当发起Burst传输时，第一拍是NONSEQ态，其后才是SEQ态。

注意　在CM0处理器中，只有IDLE和NONSEQ两种状态交替出现。

HSIZE信号表示一次传输的数据宽度。写数据HWDATA和读数据HRDATA都有32位，相当于C语言中的int类型，它包含4字节。在一拍传输中，并非一定要传输4字节，程序员可以根据自己的需要决定传输的字节数。需要注意的是，HSIZE表示的是一拍内传输的字节数，而HTRANS的SEQ态是多个节拍传输。以FIFO作类比，HSIZE规定了FIFO的宽度，HTRANS或HBURST规定了FIFO的深度。

注意　由于AHB数据位宽只有32位，所以HSIZE只会出现0、1、2。

CPU通过HWRITE信号告知设备它的操作目的是读还是写。若为写，则要写的数据将在下一拍出现在HWDATA线上。若为读，则设备需要准备好数据，并在下一拍输出。

HPROT 信号中不同的比特有着不同的含义。第 0 位表示本次传输，传的是指令还是数据，因为 CPU 运行程序时既有读指令的动作，又有读写数据的动作。若传的是指令，则 HWRITE 必定为 0，即读状态，因为 CPU 运行程序时只会读指令，不会写指令，并且 HSIZE 一定为 2，即 4 字节传输。若传的是数据，则读写都有可能，HSIZE 也随程序而变。第 1 位表示用户的读写权限，在 CM0 中，只支持特权模式，因此第 1 位固定是 1。第 2 位和第 3 位分别代表 CPU 是否允许设备对该数据进行 Buffer 或 Cache，因此这两位分别代表 Bufferable 和 Cacheable 状态。Bufferable 是指 CPU 是否允许数据延迟到达，即当 CPU 发起写时，是否允许数据不立即进入指定地址存储，当 CPU 发起读时，是否允许数据不立即反馈，而由设备自行决定延迟。Cacheable 是指 CPU 是否允许数据存储在缓存中，或数据来自缓存而非 CPU 请求的地址。例如 CPU 要求从地址 0x3 中读出数据，但是缓存地址 0x7 中刚好有 0x3 中的数据，则数据从 0x7 中读出并反馈给 CPU，而非直接出自地址 0x3。一般，CPU 是按照设备地址来决定 Bufferable 或 Cacheable 性质的，在 CM0 中，代码地址段 0x0000000～0x1FFFFFF 固定是 Cacheable 的，并且不能 Buffer，SRAM 地址段 0x20000000～0x3FFFFFFF 固定为既 Cacheable 又 Bufferable，设备地址段 0x40000000～0x5FFFFFFF 只能被 Buffer，不能被 Cache。从这一点可以看出，虽然设计者可以通过修改 cmsdk_mcu_addr_decode 文件来任意指定各 AHB 设备的地址，即 SRAM 可以不放在规定的 SRAM 地址段，其他外设也可以不放在外设地址段，但 CPU 在 HPROT 处的限制还是间接要求设计者遵守上述规定。当然，在设计简单的 AHB 设备时可以完全忽略 HPROT 信息。

注意 Bufferable 和 Cacheable 翻译为中文分别是可缓冲的和可缓存的。其含义表达不够清晰，因此一般以英文形式呈现。

HBURST 是 Burst 传输控制信号。若为 0，则只传单个数据，其他模式为连续传输。当连续传输时，相应的 HADDR 会在传输过程中自加，所加的数为 HSIZE 规定的字节数。普通 Burst 为简单自加（自加时需要注意必须控制在本设备范围内，不能通过自加进入其他地址范围），而 WRAP 模式是在 CPU 中有 Cache 时才会用到，因为某些数据存在 Cache 中，则地址在自加过程中可能突然变小，而 WRAP 本身就有非单调的、有转折的意思。当设备收到该信号后，会事先做好准备，例如 CPU 通知即将进行连续传输，设备就会先开辟一段空白内存，等待有连续的数据流进入或读出。实际上，所有控制信号的目的都是事先通知设备，一会儿 CPU 即将发生的动作，以便设备能减少过程发生时的延迟。

注意 在 CM0 中，HBURST 恒为 0，即不能用 Burst 传输。因为 CM0 内部用于暂存指令和数据的寄存器只有一个，无法做到 Burst 传输。

其他控制信号还有 HMASTER、HMASTLOCK，用于多核 SoC 在协同处理中锁定当前工作的 CPU，在 CM0 中不使用，这里不再赘述。

反馈信号 HRESP 传递的是设备处于错误状态的信息，当设备正常时为 0，发生错误时

为1。这里的所谓错误,指设备无法满足CPU的读写要求,或是写入无法执行,或是读出无法执行。有些时候,仅仅是设备速度慢,导致读写有延迟,这种情况不属于错误。

反馈信号HREADYOUT是设备在内部有一定读写延迟时向CPU报告的信号,它通知CPU及总线上的其他器件再多等一会儿。在总线的连接上,一个设备的HREADYOUT信号为0(未就绪)会导致整条总线工作暂停,因此,总线中存在一条总的HREADY线,它是各设备HREADYOUT信号相与后得到的。为了广播到总线上的每个设备,HREADY是总线上全部设备及CPU的输入,所以一个AHB设备,既有输出的HREADYOUT,又有输入的HREADY。反馈信号HRESP也会在每个设备中分别产生,通过或门合并为一个信号,但它并不会被广播到所有设备中,因为它的主要目的是通知CPU做决策,只需接入CPU。上述反馈信号合并处理在cmsdk_ahb_slave_mux模块中。

AHB总线的读写时序分为两拍完成,分别称为地址阶段和数据阶段。在地址阶段,全部控制信号都给出,以便使设备能获取必要的传输信息,在数据阶段,若为写操作,则HWDATA处会出现要写的数据,若为读操作,则设备准备好数据后,在此阶段发出。一个典型的读写操作如图5-19所示。读和写操作在控制侧唯一的区别是HWRITE在写时序中为1,在读时序中为0。传输过程是流水线方式,例如B轮的地址阶段与A轮的数据阶段重合,即A轮正在处理数据时,B轮的地址和控制信息已经发出,以此来提高传输速度。注意,在写时序中,设备真正获得数据是在数据阶段完成之后,例如获得数据A是在图中的*a*点,同理可知,若为读时序,则CPU采样数据A也是在该点。

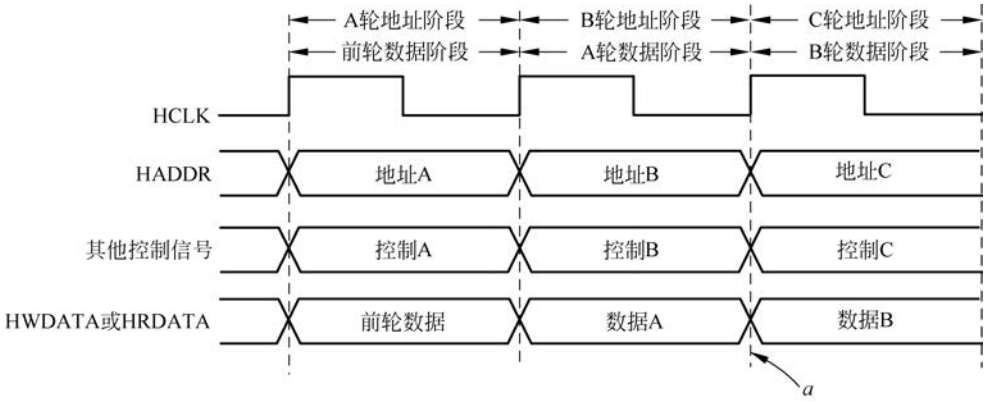


图5-19 典型AHB读写传输时序

上述典型时序出现的前提是设备反应迅速,当CPU有请求时,下一拍就能完成操作。在这种情况下,反馈信号HREADY一直为1。在实际电路中,有些设备反应较慢,无法在下一拍就完成读写,此时,该设备会将自身的HREADYOUT信号拉低,从而使总线整体的HREADY信号也变为0,以此来拖延时间,暂停CPU和其他设备的数据读写活动(各设备的动作并没有停,仍然在运转中,停的只是总线读写操作)。

一个AHB设备之所以反应迟钝,有很多种原因。例如,它是一个基于SPI接口的存储设备,CPU希望读该设备,当它发起AHB协议读后,该操作被转换为SPI读操作。由于

SPI 速度比 AHB 速度低很多，因而造成了延迟。再例如一个设备中有多个驱动时钟，AHB 的配置需要经过异步握手或异步 FIFO 才能进入设备中，此时，设备也会延迟。当然，异步设备可以先开辟一个缓冲寄存器，使 AHB 数据先进入，然后自己在内部慢慢消化，而不影响 AHB 速度，这就是上文中所谓的 Bufferable 的含义，但这种设计问题在于当 AHB 连续配置同一个寄存器时，上一轮异步握手未结束，下一轮异步握手就开始了，会造成握手错误，因此，对于需要异步握手的设备，不推荐使用 Bufferable 的方式，而是推荐将总线的 HREADY 拉低的方式。对于使用异步 FIFO 策略的设备，可以使用 Bufferable 的方式。关于异步 FIFO 和异步握手的选择，详见 4.3 节。

一个反应迟钝的 AHB 设备，通过 HREADY 暂停总线动作的时序如图 5-20 所示。图中，CPU 对地址 A 的操作在数据阶段遇到了拖延，HREADY 被地址 A 所在的设备拉低，因此，A 轮数据阶段又持续了一拍。又由于 B 轮地址阶段与 A 轮数据阶段是重合的，所以地址 B 和对应的控制也都延迟一拍。若该传输为 CPU 写操作，则在 HWDATA 上，数据 A 也延迟一拍，即图中数据 A1 与数据 A2 相同。若该传输为 CPU 读操作，则在 HRDATA 上，数据 A1 不被 CPU 接受，只有数据 A2 会被 CPU 采样，因为 HREADY 本身就表示数据 A1 是无效的。

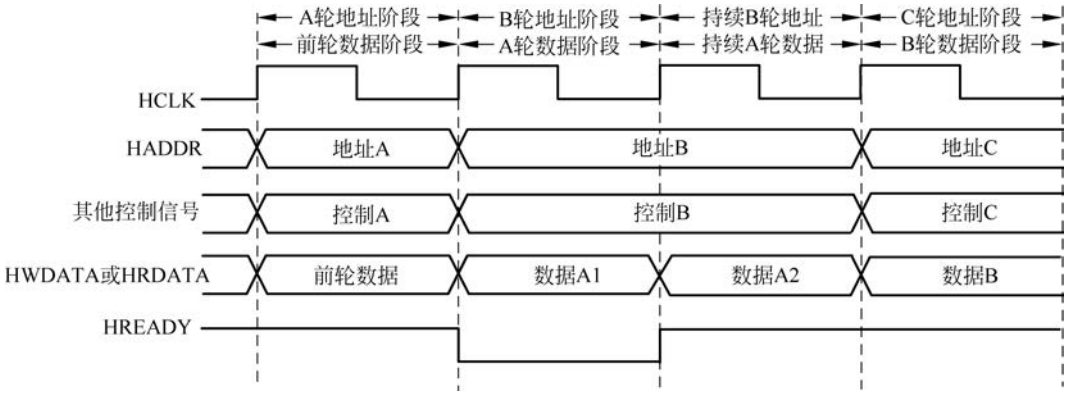


图 5-20 有设备延迟的 AHB 传输

一个设备发生错误，将 HRESP 拉高的示例如图 5-21 所示。CPU 对地址 A 进行读写操作，而此地址无法进行正常读写，地址 A 的设备会回复 HRESP，持续两拍。接下来，CPU 会继续它的操作。

一个真实的 AHB 传输案例如图 5-22 所示。在第 A 拍之前，HTRANS 是 0，说明总线空闲。在第 A 拍，HTRANS 变为 2，是单数据传输，同时 HSIZE 也变为 2，是 4 字节传输，HWRITE 为 0，说明是读操作。HPROT 是 0xA，说明读的是指令，并且 Cacheable。HADDR 是 0x02000BAC，因此 Cacheable 符合预期。第 A 拍是该通信的地址阶段，第 B 拍进入数据阶段后 HREADY 拉低，说明地址 0x02000BAC 读出需要一定延迟，该操作拖延了若干拍，直到第 C 拍 HREADY 上拉，HRDATA 才输出有效数据 0x43017060。从第 B 拍开始，在等待数据的同时，下一次传输的地址阶段也开始了。它也是单数据传输，HSIZE 为

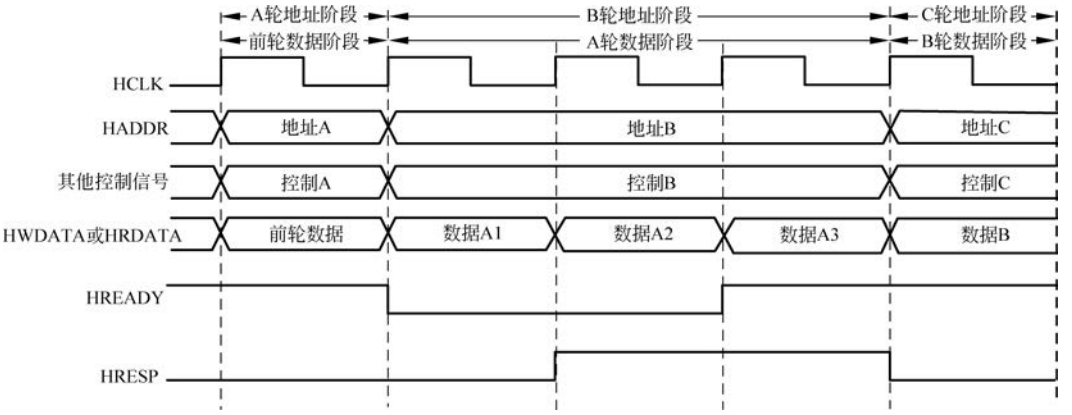


图 5-21　HRESP 的时序

0 说明只操作一字节，HWRITE 为高说明是 CPU 将向地址 0x20000410 写入一字节，写入内容为 0，在 HWDATA 中显示。HPROT 为 0xF，说明写入的是数据，并且兼具 Cacheable 和 Bufferable 两种特性。该传输在第 C 拍才真正进入地址阶段，并且在下一拍成功写入，HREADY 没有拉低。如前文所述，一般以 0x20000000 为基地址的设备是 SRAM，因此，对它的读写基本不会发生延迟，而以 0x02000000 为基地址的设备是指令存储器，包含 ROM 和其他 NVM 器件，ROM 没有延迟，但不同类型、不同接口的 NVM 对读操作的响应速度是不同的。

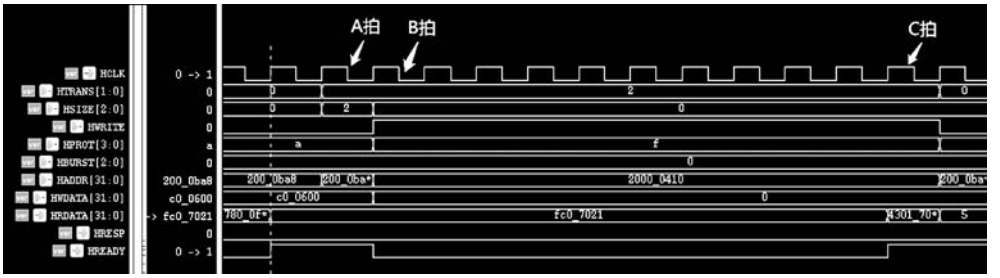


图 5-22　AHB 总线的真实波形

5.17　AHB 设备的设计

28min

总线上挂载的设备有着各自的功能，CPU 仅将其视作一组可供读写的存储器。可以将设备想象为人体器官，例如眼睛负责采集光信号，耳朵负责采集声音信号，它们都是接收设备，发射设备如嘴可以说话，手和脚可以做出动作，语音和动作都可以视为输出信号。CPU 是大脑，负责将采集来的信号进行处理，并指挥输出设备发出程序员想要的信号。如果将眼睛单独流片，那它就是一个光敏器件，放在没有 CPU 的简单场景也可以使用，因此，不能单纯地认为一切设备都是为 CPU 服务的，在很多专用芯片中，CPU 和 SoC 只起到辅助作用，

真正发挥作用的是设备本身。

那么,CPU 究竟起到了什么辅助作用呢?主要是它具有按顺序配置存储器的能力。如果没有 CPU,就不能编程。程序即步骤,不能编程意味着无法实现有步骤的操作,例如某设备中有两个寄存器 A 和 B,设备要求先配置 A 再配置 B。有了 CPU,这项要求很容易实现。若没有 CPU,默认情况下,A 和 B 会同时配置,除非在硬件上刻意打拍,拉开配置的时间,也可以通过有限状态机来设定一个流程步骤,但这样的设定和打拍都是由硬件实现的,若该设备又改变了配置顺序,先配置 B 再配置 A,则无 CPU 的情况下就必须重新设计和流片,而有 CPU 的情况下可以轻松调整该顺序。

在芯片架构设计时,存在一个 CPU 选型的问题,即选择一款适合本芯片应用场景的 CPU,主要的衡量指标就是 CPU 的处理速度。处理速度快的 CPU 自然价格高,那么如何在 CPU 的性能和芯片成本之间做到平衡呢?主要考虑设备的要求,特别是关键设备对 CPU 处理速度的要求。例如,一款 ADC(用于将模拟信号转换为数字信号)的采样速率是 100MSa/s,可知它以 100MHz 的频率向 CPU 输入数据流,那么 CPU 就必须有相对应的处理频率,而另一款 UART 设备以波特率 460 800Hz 向 CPU 输入信号,即它以每字节 46kHz 的频率向 CPU 输入数据流,那么对 CPU 的处理要求就很低。除了单个设备的要求,还要考虑到设备综合的要求,一款 SoC 中往往包含多个设备协同工作,需要 CPU 协调它们的工作流程和步骤,因此,CPU 的处理速度选择是以关键设备为主,兼顾整个系统的需求。

由于设备各自有其特殊功能,本节所介绍的仅仅是设备中 AHB 接口的处理方法。AHB 总线接口上的信号,不必全部使用。对于以 CM0 为核心的设备来讲,HMASTER、HMASTERLOCK、HBURST 等 3 个信号基本不用。HPROT 中第 0 位可以在看波形时帮助设计者分辨是数据还是指令,但一般在设备中用不到,第 1 位是固定的 1,也用不到,第 2 位表示的 Bufferable 特性和第 3 位表示的 Cacheable 特性,其设定对于不同的地址段基本是固定的,因此,设备只要知道自己所在的地址段就知道这两个性质,不需要引入这两个信号。如果设备速度快,操作中间不会出现任何读写延迟,则可将 HREADYOUT 直接设为 1,不需要接入任何逻辑。如果设备不会出现读写错误的情况,则将 HRESP 设为恒 0 即可,实际上,除非设备中包含特殊的存储介质,不然一般只会发生延迟读写,不会发生读写错误。HTRANS 在 CM0 中也只有 0 和 2 两种状态。

一个简单的 AHB 设备代码如下例所示,为突出重点,未设计具体功能,仅提供了存储用的寄存器供 CPU 读写。例中,HMASTER、HMASTERLOCK、HBURST、HPROT 等 4 个信号都未接入设备中。由于该设备用寄存器作为存储介质,读写无延迟,输出 HREADYOUT 恒为 1,而且不可能产生错误,HRESP 也恒为 0。提供给 CPU 读写的寄存器为 3 个 32 位寄存器 cfg_dat0、cfg_dat1、cfg_dat2,总的字节数有 12 个,只需 4 位地址表示,因而接口上并未将 32 位 HADDR 全部接入,只接了最低的 4 位。接口中,HSEL 不属于 AHB 接口信号,它是由 HADDR 通过组合逻辑产生的,在第 5.5 节中图 5-7 的地址解析器即为产生各 HSEL 的模块,该节也给出了地址解析器的 Verilog 代码,图 5-16 中的 cmsdk_mcu_addr_decode 模块也是该模块,这里不再赘述。

在模块内部,首先通过控制信号产生自己的选择信号 sel,由于 CM0 中的 HTRANS 只有 0 和 2 两种状态,HTRANS[1]为 1 即表示传输态,因而 sel 成立的条件是 HADDR 选中本设备,CPU 要发起数据传输,并且总线是空闲的。byt0～byt3 选择了寄存器的不同字节范围。byt0 选中最低位字节,byt1 选中次低位字节,byt2 选中次高位字节,byt3 选中最高位字节。选择的方法是 HSIZE 与 HADDR 结合。HADDR[1:0]负责选择一个 int 类型(32 位)中的某字节,而 HADDR[3:2]负责选择某个 int。选择字节,最直接的选择就是使用 HADDR[1:0],适合程序员用 char 指针类型(8 位)进行寻址。另外,当程序员使用 short 类型(16 位)时,也可能选中该字节,因而,HSIZE[0]为 1 就说明程序员使用 short 指针方式寻址。此时,当 HADDR[1]为 0 时,选择低位的两字节,而当 HADDR[1]为 1 时,选择高位的两字节。再者,当程序员使用 int 指针类型寻址时,HSIZE[1]为 1,全部 4 字节都被选中。接下来,之所以需要对 sel、byt0、byt1、byt2、byt3、HWRITE、HADDR 进行打拍,是因为 AHB 的时序是控制信号与数据信号错位一拍,当上述控制信号被设备采样时,HWDATA 尚无法采样,必须再等一拍,因此,为了采样到 HWDATA,并保证该采样过程仍在控制范围内,就必须将控制信号打拍,然后用打过拍的控制信号来控制 HWDATA 的采样过程,从而生成 cfg_dat0 等信号。注意,在生成 cfg_dat0 的 always 块中连续用了多个 if,这些 if 不能改成 else if,因为它们是分别独立判断的,不存在 else 关系,这样写的目的是减少代码行数,也可以分别将 cfg_dat0[7:0]、cfg_dat0[15:8]等信号的产生写为多个 always 块。

```
module ahb_device
(
    //AHB interface
    input                HCLK        ,
    input                HRESETn     ,
    input                HSEL        ,
    input                HREADY      ,
    input       [1:0]    HTRANS      ,
    input       [2:0]    HSIZE       ,
    input                HWRITE      ,
    input       [3:0]    HADDR       ,
    input       [31:0]   HWDATA      ,
    output  reg [31:0]   HRDATA      ,
    output               HREADYOUT   ,
    output               HRESP       ,

    //function
    output  reg [31:0]   cfg_dat0    ,
    output  reg [31:0]   cfg_dat1    ,
    output  reg [31:0]   cfg_dat2
);

//-----------------------------------------------
wire                     sel         ;
wire                     byt0        ;
```

```
wire                    byt1        ;
wire                    byt2        ;
wire                    byt3        ;

reg                     byt0_r      ;
reg                     byt1_r      ;
reg                     byt2_r      ;
reg                     byt3_r      ;

reg         [1:0]       addr_inte   ;

//-----------------------------------------------
assign sel  =  HSEL & HTRANS[1] & HREADY;
assign byt0 =  HSIZE[1] | (HSIZE[0] & (~HADDR[1])) | (HADDR[1:0] == 2'b00);
assign byt1 =  HSIZE[1] | (HSIZE[0] & (~HADDR[1])) | (HADDR[1:0] == 2'b01);
assign byt2 =  HSIZE[1] | (HSIZE[0] & HADDR[1])    | (HADDR[1:0] == 2'b10);
assign byt3 =  HSIZE[1] | (HSIZE[0] & HADDR[1])    | (HADDR[1:0] == 2'b11);

always @(posedge HCLK or negedge HRESETn)
begin
    if (!HRESETn)
    begin
        byt0_r      <= 1'b0;
        byt1_r      <= 1'b0;
        byt2_r      <= 1'b0;
        byt3_r      <= 1'b0;
        addr_inte   <= 2'd0;
    end
    else
    begin
        byt0_r      <= byt0 & HWRITE & sel;
        byt1_r      <= byt1 & HWRITE & sel;
        byt2_r      <= byt2 & HWRITE & sel;
        byt3_r      <= byt3 & HWRITE & sel;
        addr_inte   <= HADDR[3:2]   ;
    end
end

//---------   write   -------------
always @(posedge HCLK or negedge HRESETn)
begin
    if (!HRESETn)
        cfg_dat0 <= 32'd0;
    else
    begin
        if (byt0_r & (addr_inte == 2'd0))
            cfg_dat0[7:0]   <= HWDATA[7:0];

        //注意: 不是 else if,而是 if.将若干个 always 块的逻辑合并为一个
        if (byt1_r & (addr_inte == 2'd0))
            cfg_dat0[15:8]  <= HWDATA[15:8];
```

```
        if (byt2_r & (addr_inte == 2'd0))
            cfg_dat0[23:16]  <= HWDATA[23:16];

        if (byt3_r & (addr_inte == 2'd0))
            cfg_dat0[31:24]  <= HWDATA[31:24];
    end
end

always @(posedge HCLK or negedge HRESETn)
begin
    if (!HRESETn)
        cfg_dat1 <= 32'd0;
    else
    begin
        if (byt0_r & (addr_inte == 2'd1))
            cfg_dat1[7:0]    <= HWDATA[7:0];

        if (byt1_r & (addr_inte == 2'd1))
            cfg_dat1[15:8]   <= HWDATA[15:8];

        if (byt2_r & (addr_inte == 2'd1))
            cfg_dat1[23:16]  <= HWDATA[23:16];

        if (byt3_r & (addr_inte == 2'd1))
            cfg_dat1[31:24]  <= HWDATA[31:24];
    end
end

always @(posedge HCLK or negedge HRESETn)
begin
    if (!HRESETn)
        cfg_dat2 <= 32'd0;
    else
    begin
        if (byt0_r & (addr_inte == 2'd2))
            cfg_dat2[7:0]    <= HWDATA[7:0];

        if (byt1_r & (addr_inte == 2'd2))
            cfg_dat2[15:8]   <= HWDATA[15:8];

        if (byt2_r & (addr_inte == 2'd2))
            cfg_dat2[23:16]  <= HWDATA[23:16];

        if (byt3_r & (addr_inte == 2'd2))
            cfg_dat2[31:24]  <= HWDATA[31:24];
    end
end

//------------- read ----------------
//注意:这里的读操作是组合逻辑,固定输出32位
```

```
always @( * )
begin
    //注意: 这里使用了 addr_inte,而非 HADDR
    case (addr_inte)
        2'd0     : HRDATA = cfg_dat0;
        2'd1     : HRDATA = cfg_dat1;
        2'd2     : HRDATA = cfg_dat2;
        default  : HRDATA = 32'd0;
    endcase
end

assign HREADYOUT = 1'b1;
assign HRESP     = 1'b0;

endmodule
```

最需要关注的是例子中 HRDATA 的产生逻辑使用了组合逻辑,根据 int 地址 addr_inte 分别反馈 3 个 32 位寄存器的值。有一些初学者认为应该按照 HSIZE 发送,即要求发一字节时,HRDATA 就放入一字节,要求发两字节时,HRDATA 就放入两字节,但实际上,不论 HSIZE 是多少,设备传给总线的都是 4 字节,CPU 会在内部选取它想要的字节。一个读写设备的 C 语言程序如下例。0x40010004 是 AHB 设备中寄存器 cfg_dat1 的绝对地址。先在地址 0x40010004 中写入数据 0x00054321,然后读出地址 0x40010005 中存储的字节。虽然设备回复给 CPU 的数据是 0x00054321,但实际 CPU 取到的是 0x43。代码中的 __ISB()调用了 ISB 指令,即等待上一句命令执行完毕,若没有它,由于上下两句读写的都是同一个寄存器,数据 0x00054321 仍然存在于 CPU 中,所以它将不会读取设备,直接将 0x43 赋值给 tmp,达不到实验目的。

```
*((int *)0x40010004) = 0x00054321;
__ISB();
tmp = *((char *)0x40010005)
```

在 RTL 中产生 HRDATA 时,用的地址是 addr_inte,而不是 HADDR,这是因为该逻辑是一个组合逻辑,在时序的第二拍才是数据阶段,因而用的是 HADDR 的打拍信号 addr_inte 作为地址选择。

这里存在一个问题,如下面的 C 代码所示,希望读出从 0x40010005(为了简化表述,下文用地址 0x5 代表该地址)开始的两字节,但最后读出的时序如图 5-23 所示,传输从第 A 拍持续到第 B 拍,HSEL 为高表示本设备已经被选中,HSIZE 为 1 恰为两字节,但地址发出的是 0x4,HTRANS 是 0,并未发出传输。原因是这里的编译过程只允许双字节对齐和 4 字节对齐,即可读从任意地址开始的单字节,从地址 0x4 开始的双字节,或从 0x6 开始的双字节,或从 0x4 开始的 4 字节,但不允许读从 0x5 开始的双字节,或从 0x7 开始的双字节,或从 0x5、0x6、0x7 开始的 4 字节。如果出现了上述不允许的情况,程序编译器则会直接解析为无效传输。对于这种情况,HRESP 也可以拉高两拍,以通知总线发生未对齐错误。

```
*((int *)0x40010004) = 0x00054321;
__ISB();
tmp = *((short *)0x40010005)
```

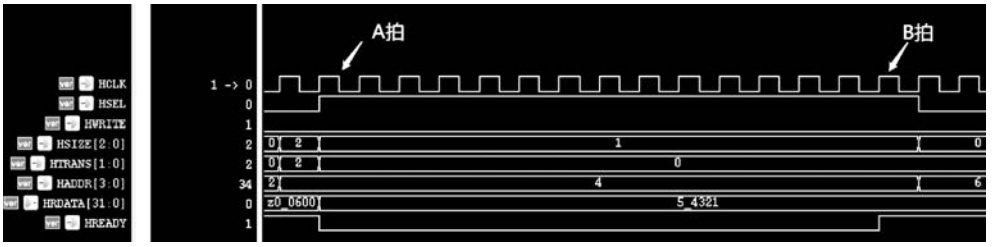


图 5-23 传输被取消的案例

实际上,嵌入式程序员最惯用的读取方法是首先读取 4 字节,然后取出自己需要的字节,如下例就可以实现读取从地址 0x5 开始的两字节,先读出从地址 0x4 开始的 4 字节,然后向右移除一字节,再通过位与 0xFF,掩住最高的一字节,剩下的就是中间两字节。

```
tmp = *((unsigned int *)0x40010004)
aaa = (tmp >> 8) & 0xff;
```

注意 RTL 中的 addr_inte,后缀_inte 表示 Integer,即 4 字节整数。由于_int 后缀一般表示中断,即 Interrupt,所以需要注意两个后缀的细微差别。

下例是一段 SRAM 与 AHB 接口交互的代码,其中,u_sram 的生成已在图 5-2 中做了介绍。本节主要介绍将 AHB 接口协议与生成的 SRAM IP 接口相互转换的逻辑,其中,sel、byt0、byt1、byt2、byt3 等信号逻辑与上例中一致。wr 和 rd 信号是对 SRAM 真正写和读意图的解析。wr_byt0_latch 等 4 个信号、地址信号 wr_addr_inte_latch,以及数据信号 wdat_latch,是对字节写状态的保留。一般的信号,在写操作完成之后就会清零,但它们会一直保持,直到下一次写操作时才更新,因而给它们加了_latch 后缀。上一次写的信息就被原封不动地存储下来,可以视为一个深度为 1 的小型数据 Cache,当某个读地址恰好与 Cache 中数据地址 wr_addr_inte_latch 对应时,该模块就不用读 SRAM,而是直接向 CPU 返回 wdat_latch 内容。这种请求内容与 Cache 中内容一致的现象称为命中,即 buf_hit。本例中 SRAM 控制器的策略是优先读,即若出现了一个写,紧接着出现了连续读,则写操作将一直不发生,它将转换为挂起状态,即 buf_pend 状态。直到读停止,当出现了写或者总线空闲时才会将刚才要写的内容写入,即 ram_wr 状态。由于在 AHB 写和真正的 SRAM 写之间可能存在时间差,所以才需要上述 Cache 的加入。在生成的 SRAM IP 中,写命令 ram_wr_en_n 是低电平有效,每字节都有一个专门的使能开关,因此需要组合拼接。ram_cs 是片选信号,ram_addr_inte 是地址,只支持 32 位一起写入和读出。

```
//与前例相同的 sel 逻辑
assign sel  =  HSEL & HTRANS[1] & HREADY;

//总线读写操作信号
assign wr   =  sel & HWRITE;
assign rd   =  sel & (~HWRITE);

//比特选择逻辑不变
assign byt0 =  HSIZE[1] | (HSIZE[0] & (~HADDR[1])) | (HADDR[1:0] == 2'b00);
assign byt1 =  HSIZE[1] | (HSIZE[0] & (~HADDR[1])) | (HADDR[1:0] == 2'b01);
assign byt2 =  HSIZE[1] | (HSIZE[0] & HADDR[1])    | (HADDR[1:0] == 2'b10);
assign byt3 =  HSIZE[1] | (HSIZE[0] & HADDR[1])    | (HADDR[1:0] == 2'b11);

always @(posedge HCLK or negedge HRESETn)
begin
    if (!HRESETn)
        wr_r <= 1'b0;
    else
        wr_r <= wr;
end

//缓存写位置
always @(posedge HCLK or negedge HRESETn)
begin
    if (!HRESETn)
    begin
        wr_byt0_latch      <= 1'b0;
        wr_byt1_latch      <= 1'b0;
        wr_byt2_latch      <= 1'b0;
        wr_byt3_latch      <= 1'b0;
    end
    else if (wr)
    begin
        wr_byt0_latch      <= byt0 & sel;
        wr_byt1_latch      <= byt1 & sel;
        wr_byt2_latch      <= byt2 & sel;
        wr_byt3_latch      <= byt3 & sel;
    end
end

//缓存写地址
always @(posedge HCLK or negedge HRESETn)
begin
    if (!HRESETn)
        wr_addr_inte_latch <= 11'd0;
    else if (wr)
        wr_addr_inte_latch <= HADDR[12:2];
end
```

```
//与缓存地址进行对比,判断是否命中
always @(posedge HCLK or negedge HRESETn)
begin
    if (!HRESETn)
        buf_hit <= 1'b0;
    else if(rd)
    begin
        if (HADDR[12:2] == wr_addr_inte_latch)
            buf_hit <= 1'b1;
        else
            buf_hit <= 1'b0;
    end
end

//缓存写数据
always @(posedge HCLK or negedge HRESETn)
begin
    if (!HRESETn)
        wdat_latch <= 32'd0;
    else if (wr_r)
    begin
        if (wr_byt0_latch)
            wdat_latch[7:0]     <= HWDATA[7:0];

        if (wr_byt1_latch)
            wdat_latch[15:8]    <= HWDATA[15:8];

        if (wr_byt2_latch)
            wdat_latch[23:16]   <= HWDATA[23:16];

        if (wr_byt3_latch)
            wdat_latch[31:24]   <= HWDATA[31:24];
    end
end

//若命中,则从缓存中调用写数据直接读出,若未命中,则读 SRAM
assign HRDATA =
{ (buf_hit & wr_byt3_latch) ? wdat_latch[31:24] : ram_rdat[31:24],
  (buf_hit & wr_byt2_latch) ? wdat_latch[23:16] : ram_rdat[23:16],
  (buf_hit & wr_byt1_latch) ? wdat_latch[15: 8] : ram_rdat[15: 8],
  (buf_hit & wr_byt0_latch) ? wdat_latch[ 7: 0] : ram_rdat[ 7: 0] };

//-----------------------------------------------------------------
//写请求挂起逻辑,若操作顺序为写->读,则写操作进入 SRAM 的真正时间被推迟,即挂起
always @(posedge HCLK or negedge HRESETn)
begin
    if (!HRESETn)
        buf_pend <= 1'b0;
    else
```

```
    begin
        if (~buf_pend)
        begin
            if (wr_r & rd)
                buf_pend <= 1'b1; //先写后读,挂起
            else
                buf_pend <= 1'b0; //不挂起
        end
        else                        //buf_pend == 1
        begin
            if (rd)
                buf_pend <= 1'b1; //保持挂起状态的方法是连续读,中间不能断
            else
                buf_pend <= 1'b0;
        end
    end
end

//挂起状态中断,或者没有发生挂起,就会发生 SRAM 的实际写操作
assign ram_wr = (buf_pend | wr_r) & (~rd);

//32 位比特使能信号,低有效
always @( * )
begin
    if (ram_wr)
        ram_wr_en_n =   {{8{~wr_byt3_latch}},
                         {8{~wr_byt2_latch}},
                         {8{~wr_byt1_latch}},
                         {8{~wr_byt0_latch}}};
    else
        ram_wr_en_n = 32'hffffffff;
end

//SRAM 片选信号,低有效.只要 SRAM 发生了实际读或写,都需要拉低
assign ram_csn = ~(rd | ram_wr);

//SRAM 地址,只接受 32 位地址,去除末尾两位
assign ram_addr_inte = rd ? HADDR[12:2] : wr_addr_inte_latch;

//SRAM 写数据,若刚才是挂起的,则从缓存中取数据,若未挂起,则从总线上取数据
assign ram_wdat = buf_pend ? wdat_latch : HWDATA;

//例化 SRAM
sram    u_sram
(
    .CLK        (HCLK                ), //i
    .CEN        (ram_csn             ), //i
    .WEN        (ram_wr_en_n         ), //i[31:0]
    .A          (ram_addr_inte       ), //i[10:0]
    .D          (ram_wdat            ), //i[31:0]
    .Q          (ram_rdat            )  //o[31:0]
);
```

下例是 ROM 的 AHB 接口逻辑,相对于 SRAM 来讲,ROM 的接口十分简单,只有读操作,而且,由于本例使用的 ROM 是一拍读出,即发出读命令后,下一拍将数据读出来,恰好符合 AHB 的地址阶段与数据阶段的时序,因此对 AHB 控制信号也未做打拍。如果 ROM 读出延迟大,则还需要在 HREADYOUT 上加入几拍低电平,以拖延总线。

```
//与前例相同的 sel 逻辑
assign sel = HSEL & HTRANS[1] & HREADY;

//总线读操作,直接进入 ROM
assign rom_rd = sel & (~HWRITE);

//ROM 地址
assign rom_addr_inte = HADDR[12:2];

//例化 ROM
rom     u_rom
(
    .CLK      (HCLK                ),//i
    .CEN      (~rom_rd             ),//i
    .A        (rom_addr_inte       ),//i[10:0]
    .Q        (HRDATA              ) //o[31:0]
);
```

5.18 APB 总线协议

APB 总线是比 AHB 更为慢速的协议,优点是其接口协议更为简单。APB 设备是比 AHB 设备更次一级的设备,会通过 AHB 转 APB 桥来完成与 AHB 总线的互联。在系统中,CPU 只能看到 AHB 设备,因此,设计者不会在 CPU 上找到 APB 接口。APB 设备在 CPU 看来也是 AHB 设备,CPU 读写这些设备的存储器也是通过 AHB 总线协议完成的,只不过中间经过了桥的翻译。这座桥即为图 5-16 中的 cmsdk_ahb_to_apb 模块。所有 APB 设备信息都先在该模块汇总,并上传给 CPU,CPU 也只与该模块打交道,相当于 CPU 与众 APB 设备之间的管家。APB 协议也分为不同的版本,本书所介绍的是与 AMBA3 AHB-Lite 配套的 APB3 总线协议。

APB 总线接口信号见表 5-10。相比于 AHB 接口,APB 少了很多控制。这是由于它的操作对象仅仅是普通的慢速设备,而不包括 NVM 或 SRAM 等与执行程序有密切联系的存储设备,因此不存在 code 与 data 的区分,以及 Cacheable、Burst 等概念。另一方面,它只支持 32 位寄存器一起写入和读出,不支持单字节或双字节的操作(Byte Strobe),因而也没有 Size 的概念。反馈信号 PREADY 相当于 AHB 中的 HREADY,也起到暂停总线,给设备处

理一段额外时间的作用。PSLVERR 相当于 AHB 中的 HRESP，起到报错的作用。事实上，APB 通过桥上传 PREADY 和 PSLVERR 信息时，也都转换为 HREADY 和 HRESP。在 AHB 中有一个地址解析组合逻辑专门生成各模块的选择信号 HSEL，在 APB 中也存在同样的逻辑，不过它与其他逻辑一起放在 cmsdk_ahb_to_apb 模块中，在接口上有若干个 PSEL 信号，每个 APB 设备对应一个。PENABLE 与 PSEL 配合使用。

表 5-10 APB 总线接口信号

信号类型	信号名	位宽	方向（CPU 视角）	方向（设备视角）	定　义
系统信号	PCLK	1	输入	输入	总线时钟
	PRESETn	1	输入	输入	总线复位
控制信号	PWRITE	1	输出	输入	传输方向控制： 0：CPU 读取设备 1：CPU 向设备中写
	PSEL	1	输出	输入	设备选择： 0：未被选中 1：被选中
	PENABLE	1	输出	输入	设备使能： 0：未被使能 1：被使能
	PADDR	32	输出	输入	操作地址
数据信号	PWDATA	32	输出	输入	待写入的数据
	PRDATA	32	输入	输出	读出的数据
反馈信号	PREADY	1	输入	输出	设备状态反馈： 0：设备忙，总线暂停 1：设备正常，总线继续运行
	PSLVERR	1	输入	输出	设备错误反馈： 0：设备未出故障 1：设备在读写过程中发生错误

APB 的基本写时序如图 5-24 所示。APB 总线读写也分为两拍，但它不是 AHB 那样的流水线式。流水线式，平均到每次读写都只有一拍，非流水线式每次读写都需要两拍，因而，在没有设备暂停总线或发生错误的前提下，APB 速率是 AHB 速率的一半。两拍中，前一拍称为建立阶段（Setup），后一拍称为访问阶段（Access）。设备的 PSEL 在这两拍中都为高，但 PENABLE 只在访问阶段为高，在建立阶段为低。PWRITE、PADDR、PWDATA 等在这两拍内也保持不变。读时序也同理，与写时序的区别在于 PWRITE 为低，设备必须在访问

阶段将读数据放到 PRDATA 上。

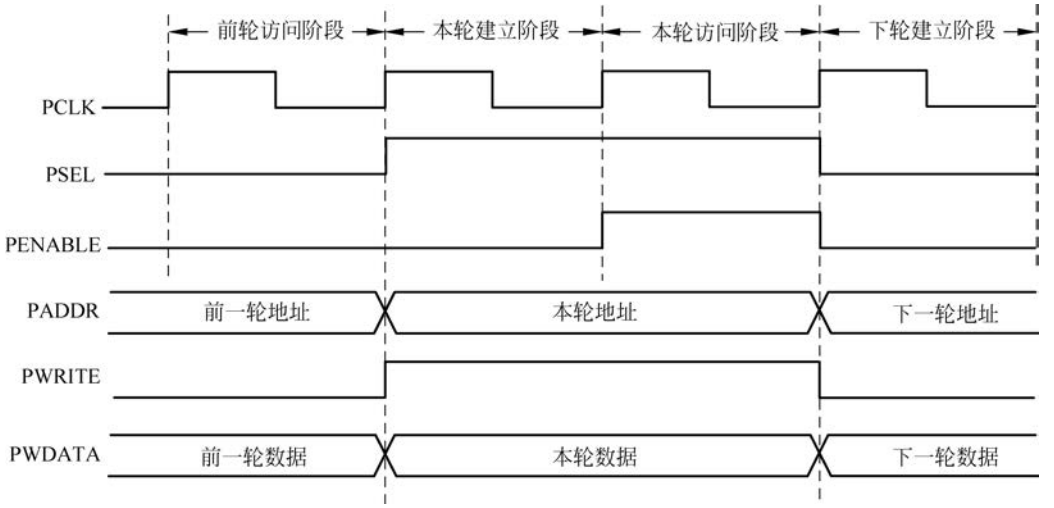


图 5-24　APB 基本写时序

图 5-24 的前提是 PREADY 恒为 1,并且 PSLVERR 恒为 0。若 PREADY 为 0,即设备速度不够,希望暂停总线动作,则该情况将会如图 5-25 所示。此时,访问阶段会拖延,拖延的拍数与 PREADY 为低的拍数相同,在拖延期间,控制信号都不变。

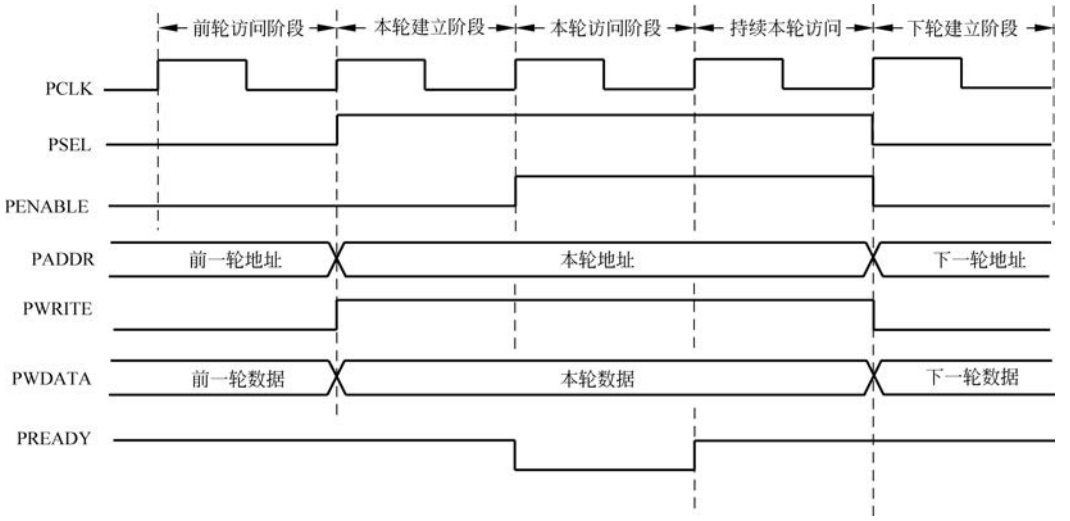


图 5-25　APB 传输中 PREADY 拉低的情况

若设备发生寄存器错误,无法写入或读出正确的数据,就会将 PSLVERR 拉高,其时序如图 5-26 所示。虽然图 5-25 和图 5-26 仅在 PSLVERR 信号上有所区别,但实际上有本质区别,特别是对于 CPU 读操作,当 PSLVERR 拉高后,CPU 会忽略读回的数据,不使用它参与后续的计算。

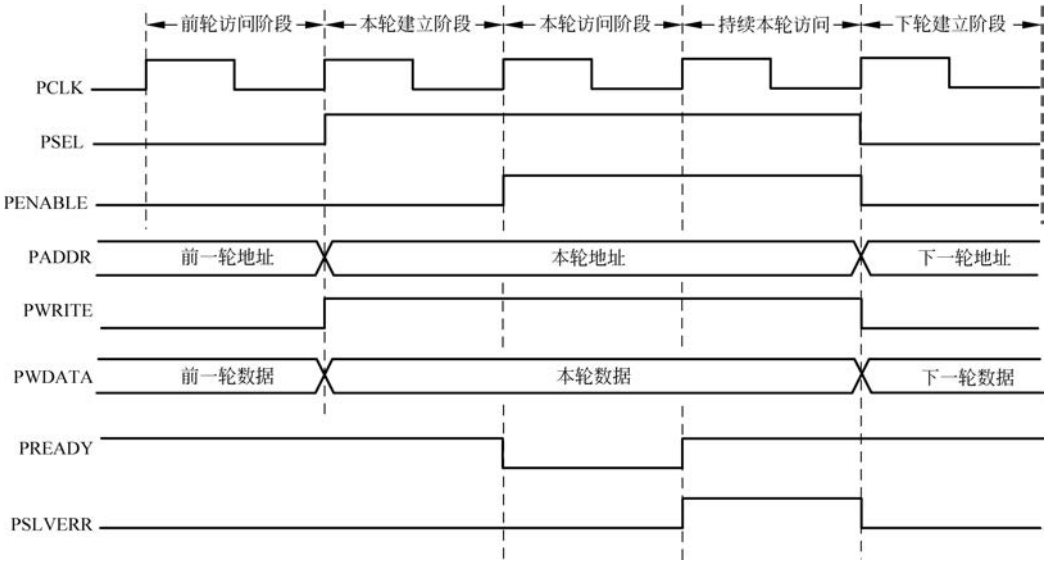


图 5-26 APB 传输中发生错误的情况

5.19 APB 设备的设计

30min

APB 设备接口比 AHB 简单，但要真正做到灵活运用接口来达到设备正常运行的目的，还需要考虑很多细节问题。下面给出了一个较为复杂的 APB 接口 Timer 设计，用以说明设计一个 SoC 设备时所需关注的要点。

该 Timer 内部计数使用 refclk，即参考时钟，它有配套的复位信号 refrstn，而 CPU 配置 Timer 或从中读取信息时，使用 APB 接口，它的时钟是 PCLK。这样的设计可以允许在 CPU 时钟频率变化的同时，Timer 能以固定的节奏计数，但是，由此也引入了跨时钟域问题，即信号在 refclk 和 PCLK 两个时钟域之间来回转换，而且两个时钟的快慢关系难以确定，refclk 可以是固定的时钟频率，但 PCLK 会发生变化，例如在 CPU 正常工作时，该时钟频率比 refclk 高，但当 CPU 休眠时，该时钟频率又变得比 refclk 要低了，因此，在这里可以考虑使用 2.23 节介绍的握手法跨时钟域技术，以及 4.3 节介绍的异步 FIFO 技术。

先关注简单 APB 接口的逻辑实现部分。读数据仍然使用组合逻辑实现，使能信号 rd_en 将持续整个传输过程，不区分建立阶段和访问阶段。只要保证在访问阶段结束后，CPU 能采样到数据即可。写操作是一个维持一拍的脉冲，不能多拍，因为多拍会不停地发起写操作，没有这样做的必要。在这里，代码选择在访问阶段产生写信号，当然，也可以选择在建立阶段产生。这里之所以在访问阶段产生，是为了简化后续 PREADY 的逻辑。在设备中不会发生读写错误，因而 PSLVERR 恒为 0。在接口逻辑中值得注意的地方是时钟采用的是 PCLKGated，而非 PCLK。PCLKGated 是在 PCLK 基础上增加了门控，当 CPU 发起读写操作时，才有 PCLKGated 信号，否则时钟将恒为 0。这样做可以减少 timer_en、int_en 等寄

存器的动态功耗,但这种门控信号只适合于接口读写,内部逻辑运行需要持续的时钟,应注意不要将 PCLKGated 用于内部寄存器的驱动。PCLKGated 的门控来自模块 cmsdk_ahb_to_apb 输出的 APBACTIVE 信号。

注意 APB3 只支持 32 位寻址,因此 PADDR 的每个地址都表示 32 位,即 4 字节。这与按字节寻址的 HADDR 不同。在 cmsdk_ahb_to_apb 协议桥电路中,在输入 HADDR 和输出 PADDR 时会去除 HADDR 的低两位,以保证这一特性。有的桥设计中未做去除,PADDR 与 HADDR 位宽一致,但最低两位是全 0,此时,在将 PADDR 引入 APB 设备中时,可在设备接口上进行截取,例如 PADDR[10:2]。

寄存器的基本类型可分为 3 种,分别是可读可写型(Read and Write)、只写型(Write Only)、只读型(Read Only)。只读和只写型如果映射在一个比特上,可衍生出读写不一致型。可读可写型是最常见的寄存器类型,其特点是写入值与读出值相同,如本例中的 timer_en、int_en、rld_dat 等信号。只写型是指该寄存器可以配置,但从配置地址读取时,无法读出刚才配置的值,例如本例中的 int_clr 和 forc_rld。事实上,只写型往往只是一个脉冲,作为触发信号使用,虽称为寄存器,但在电路中却不对应任何寄存器,仅仅是信号的组合逻辑,工程界仍称为寄存器是为了方便软件人员理解。本例中的只读信号包括 raw_int、int_timer、cur_cnt_sync2apb 等,它们的特征是在模块内部通过逻辑产生,而非人为配置,反映了设备内部的状态,在接口上只表现在 PRDATA,即用户只有读的权力,没有写的权力。可见,只读类型也不对应寄存器,状态信号在内部产生,可以是组合逻辑也可以是时序逻辑,设计者仅仅是将信号线拉到接口上而已,并未进行存储。读写不一致型体现在 PADDR 为 2 时,寄存器的第 0 比特,当用户发起写操作时为配置只写信号 int_clr,当用户发起读操作时读出的是 int_timer,即配置值与读出值不一致,当配置 int_clr 为 1 时,读出的 int_timer 却是 0,因为该地址其实是只读和只写信号的复用,需要注意其与可读可写型的差别。寄存器的地址安排是需要仔细斟酌的,一般会将同一类配置放在一起,例如本例中 timer_en 和 int_en,前者是使能 Timer 整体功能,后者是使能中断,都是使能,因而集中在同一个地址上。rld_dat 本身占用 32 位,无法与其他寄存器分享地址。建议将只读寄存器、只写寄存器、读写不一致寄存器等作为一个整体,与可读可写寄存器地址分开,因为在编程操作中,可读可写寄存器经常会使用读入、修改、写回的方式,例如 C 语言表达式 reg0 |= (1 << 2)和 reg0 &= ~(1 << 2),前者是先读入 reg0,然后将它的第 2 位写 1,最后写回 reg0,后者是将其第 2 位清零后写回,其他位不变。若可读可写型与读写不一致型共同使用 reg0 地址,其中读写不一致型安排在第 3 位,在执行上面的 C 语言程序后,从读写不一致寄存器获取的值可能是 1,再将该 1 写回原位置,就相当于发起了一个只写操作,这是不符合编程意图的,因为本意只是想改变第 2 位数值,而不想发起其他操作。在有条件的情况下,尽量将只读、只写、读写不一致的寄存器分别安排在不同的地址上,以方便理解和管理。当地址紧张时,可以将这三类混在同一个地址中,或将只读、只写与可读可写混在同一个地址中,但切忌将可读可写与读写不一致两种类型的寄存器混在同一地址中。

产生中断信号是 SoC 设备的基本特征，它可以强迫 CPU 暂停（入栈）正在进行的工作，从而关注设备本身的状态。软件一般用两种方式来处理中断，一种是传统的中断方式，另一种是查询式（也称轮询式）。传统方式已在 5.11 节进行了详述，其特点是 CPU 被强迫转到中断服务程序。若设备中断的紧急程度不高，CPU 不愿意被强迫执行，而是希望按照自己的规划来执行自己的任务和处理中断，就可以使用查询式，即 CPU 只有想查询时才查询。这时，电路中的中断信号就不应该拉起（因为拉起后 CPU 就不得不去处理中断）但设备中的中断状态需要拉起以供 CPU 查询之用。这就是本模块设计中，中断信号被分为 int_timer 和 raw_int 的原因。int_timer 是中断信号线，只有当中断使能打开时，它才可能为 1。raw_int 是内部中断状态，仅反映内部情况，不表现在中断线上。读写不一致寄存器经常在中断管理中使用，一个地址，读的是中断线的状态，即 int_timer，当用户读到后，往往会做出将中断清零的操作（称为清中断），即发出 int_clr 脉冲，将两者安排在同一个地址的同一个比特上，可以方便操作和理解。中断信号 int_timer 往往设计为电平信号，即 Latch 信号，设备内部可使它为 1，而唯有配置 int_clr 才能使它清零。在中断服务程序中，清中断是必须做的步骤，而且往往是在第 1 步就做，因为不清就会不停地反复进入本中断。查询方式是 CPU 想查询时才会读一下中断状态寄存器，看它是否为 1。这里的中断状态有两种，即 raw_int 和 int_timer。前者是设备希望发起的中断，而后者是实际发起的中断，只有当中断使能 int_en 打开时才会将 raw_int 传出来，以 int_timer 形式传给 CPU。在查询方式中，一般不使能 int_en，只查询 raw_int 的状态。

```
module apb_timer
(
    //ref
    input                   refclk      ,
    input                   refrstn     ,

    //apb inf
    input                   PCLK        ,
    input                   PCLKGated   ,
    input                   PRESETn     ,
    input                   PSEL        ,
    input           [1:0]   PADDR       ,
    input                   PENABLE     ,
    input                   PWRITE      ,
    input           [31:0]  PWDATA      ,
    output  reg     [31:0]  PRDATA      ,
    output                  PREADY      ,
    output                  PSLVERR     ,

    //function
    output                  int_timer
);

//-----------------------------------------------------------------
```

```
wire            rd_en               ;
wire            wr_access_en        ;
wire            wr_en0              ;
wire            wr_en1              ;
wire            wr_en2              ;
wire            int_clr             ;
wire            forc_rld            ;
reg             timer_en            ;
reg             int_en              ;
reg     [31:0]  rld_dat             ;
wire    [31:0]  rld_dat_sync2ref    ;
wire            forc_rld_sync2ref   ;
wire            rld_en_busy         ;
wire            timer_en_sync2ref   ;
wire            auto_rld            ;
wire            auto_rld_sync2apb   ;
wire            auto_rld_busy       ;
reg             cur_cnt_vld         ;
wire            cur_cnt_busy        ;
reg     [31:0]  cur_cnt             ;
wire    [31:0]  cur_cnt_sync2apb    ;
reg             raw_int             ;

//APB 接口信号解析
assign rd_en         = PSEL & (~PWRITE);
assign wr_access_en  = PSEL & (~PENABLE) & PWRITE;
assign wr_en0        = wr_access_en & (PADDR == 2'd0);
assign wr_en1        = wr_access_en & (PADDR == 2'd1);
assign wr_en2        = wr_access_en & (PADDR == 2'd2);
assign PSLVERR       = 1'b0;

//产生只写信号
assign int_clr       = wr_en2 & PWDATA[0];
assign forc_rld      = wr_en2 & PWDATA[2];

//APB 配置信号
always @(posedge PCLKGated or negedge PRESETn)
begin
    if (!PRESETn)
    begin
        timer_en    <= 1'b0;
        int_en      <= 1'b0;
    end
    else if (wr_en0)
    begin
        timer_en    <= PWDATA[0];
        int_en      <= PWDATA[1];
    end
end

always @(posedge PCLKGated or negedge PRESETn)
```

```
begin
    if (!PRESETn)
        rld_dat <= 32'hffffffff;
    else if (wr_en1)
        rld_dat <= PWDATA;
end

//APB 读信号
always @( * )
begin
    if (rd_en)
    begin
        case (PADDR)
            2'd0:   PRDATA = {30'd0, int_en, timer_en};
            2'd1:   PRDATA = rld_dat;
            2'd2:   PRDATA = {30'd0, raw_int, int_timer};
            2'd3:   PRDATA = cur_cnt_sync2apb;
            default:PRDATA = 32'd0;
        endcase
    end
    else
        PRDATA = 32'd0;
end

//当 forc_rld 引发 rld_dat 同步时,PREADY 拉低
assign PREADY = ~(forc_rld | rld_en_busy);

//将 rld_dat 从 APB 域同步到工作域
sync_bus
#(
    .BUS_WIDTH (32),
    .INIT      (32'hffffffff)
)  u_rld_sync2ref
(
    .clk1      (PCLK               ),//i
    .rstn1     (PRESETn            ),//i
    .clk2      (refclk             ),//i
    .rstn2     (refrstn            ),//i
    .bus1      (rld_dat            ),//i[31:0]
    .bus2      (rld_dat_sync2ref   ),//o[31:0]
    .sig1      (forc_rld           ),//i
    .sig2      (forc_rld_sync2ref  ),//o
    .busy1     (rld_en_busy        ) //o
);

//将 timer_en 从 APB 域同步到工作域
sync_direct #(.BUS_WIDTH    (1))  u_en_sync2ref
(
    .clk1      (PCLK               ),//i
    .rstn1     (PRESETn            ),//i
    .clk2      (refclk             ),//i
```

```
    .rstn2      (refrstn            ),//i
    .bus1       (timer_en           ),//i
    .bus2       (timer_en_sync2ref  ) //o
);

//计时器的核心逻辑,计数器本身,从 rld_dat 递减到 0 后会引发中断
//当 forc_rld 命令发起后,会打断其计数进程,重新载入初始数据
always @(posedge refclk or negedge refrstn)
begin
    if (!refrstn)
        cur_cnt <= 32'hffffffff;
    else
    begin
        if (forc_rld_sync2ref | auto_rld)
            cur_cnt <= rld_dat_sync2ref;
        else if (timer_en_sync2ref)
            cur_cnt <= cur_cnt - 24'd1;
    end
end


//为反馈 cur_cnt 数值而产生的 toggle 同步驱动信号
always @(posedge refclk or negedge refrstn)
begin
    if (!refrstn)
        cur_cnt_vld <= 1'b0;
    else if (timer_en_sync2ref)
        cur_cnt_vld <= ~cur_cnt_vld;
end

//用 cur_cnt_vld 驱动 cur_cnt,从工作域同步到 APB 域
sync_bus
#(
    .BUS_WIDTH  (32),
    .INIT       (32'hffffffff)
)   u_cur_cnt_sync2apb
(
    .clk1       (refclk                         ),//i
    .rstn1      (refrstn                        ),//i
    .clk2       (PCLK                           ),//i
    .rstn2      (PRESETn                        ),//i
    .bus1       (cur_cnt                        ),//i[31:0]
    .bus2       (cur_cnt_sync2apb               ),//o[31:0]
    .sig1       (cur_cnt_vld & (~cur_cnt_busy)  ),//i
    .sig2       (                               ),//o
    .busy1      (cur_cnt_busy                   ) //o
);

//自动载入脉冲,能迫使 cur_cnt 重新载入初始值,同时也是中断产生的条件
assign auto_rld = (cur_cnt == 32'd0);
```

```
//将 auto_rld 从工作域同步到 APB 域
sync_sig     u_auto_rld
(
    .clk1        (refclk                        ),//i
    .rstn1       (refrstn                       ),//i
    .clk2        (PCLK                          ),//i
    .rstn2       (PRESETn                       ),//i
    .sig1        (auto_rld & (~auto_rld_busy)   ),//i
    .sig2        (auto_rld_sync2apb             ),//o
    .busy1       (auto_rld_busy                 ) //o
);

//中断状态标志,是一个 Latch 信号,因此必须用时序逻辑
//当 auto_rld 时,状态为 1,用户手动清零后才恢复 0
always @(posedge PCLK or negedge PRESETn)
begin
    if (!PRESETn)
        raw_int <= 1'b0;
    else if (int_clr)
        raw_int <= 1'b0;
    else if (timer_en)
    begin
        if (auto_rld_sync2apb)
            raw_int <= 1'b1;
    end
end

//实际传送给 CPU 的中断信号,受到 int_en 门控影响
assign int_timer   = raw_int & int_en;

endmodule
```

上述接口、中断等知识均为 SoC 设备设计所通用,具有普遍性和一般性。此外,本例中的 Timer 还体现了在设计跨时钟域设备时的一些要点。它的主要寄存器都准备了两套,一套工作在 PCLK 上,负责与 APB 总线进行信息交互,另一套工作在 refclk 上,负责在内部运行时发挥作用。换句话说,CPU 给设备的配置,不能立即作用到设备中,同样,CPU 欲读设备的状态,它所得到的亦非设备内部实时运行的状态。这样的寄存器如 rld_dat,它是重载数据(Reload Data)的简称,是 APB 上的寄存器,对应的内部寄存器为 rld_dat_sync2ref。它的主要功能是给 Timer 一个计数的起点,并且当每次计数到 0 时,重新装载的仍然是该数。当 CPU 将一个 rld_dat 值配置给 Timer 时,不能立即生效,因为此时 Timer 很可能正在计数,新的重载数据直接进入可能会给 cur_cnt 赋一个亚稳态值。在设计时,需要把控将 rld_dat 传送到内部的时机和条件,例如,当 cur_cnt 每次数到 0 时将 rld_dat 信号传来,而跨时钟需要握手和打拍,需要在 cur_cnt 数到 0 之前数拍就发起跨时钟使能,也可以用手动方

式,即用户配置 rld_dat 后,再配置一个强制同步的脉冲信号 forc_rld,使能数据跨时钟,还可以在配置 rld_dat 后,自动产生一个同步脉冲,将该信号同步到内部时钟。本例中使用的是手动同步方式。sync_bus 是一个总线跨时钟同步器模块,其内部构造已在 2.23 节进行了详述。这里之所以使用握手同步方式而非异步 FIFO 方式,是因为 rld_dat 不是数据流,用户只是偶尔改变其配置,并不以传输数据为目的,在这种情况下,FIFO 的深度为 1～2,整体逻辑开销与握手方式没有明显区别。在设计时应根据需求灵活决定某信号是否需要做跨时钟,如果需求规定 Timer 在运行时不允许重载数据,仅在 Timer 不计数时重载,即所谓静态重载,则 rld_dat 可以不做跨时钟处理,直接拉到内部使用。对于 Timer 内部而言,rld_dat 可视作一种准静态信号,其定义为虽然不是纯静态信号,但不经常变化,可视其为静态信号。静态信号是指在设备运行的全程值都不变的信号。同理,Timer 的使能 timer_en 也有其对应的内部信号 timer_en_sync2ref。该信号的同步使用的是 sync_direct 同步器,即电平信号直接同步器,因为 timer_en 是一个电平信号,该同步器结构和代码也可参阅 2.23 节。很多设计会将此类配置信号作为准静态信号而不予跨时钟处理,此时,需要分析清楚亚稳态的传播路径是否与其他设备相互隔离,以及为本设备设计一个专用的软复位,以备出现亚稳态时通过手动软复位使其恢复正常初始态。

注意 在硬件设计中,所谓手动,指的是通过软件编程的方式实现某种功能。所谓自动,指的是通过 FSM 状态机等硬件自触发机制实现功能。自动方式可以减少软件工作量,也有速度快的优势,但一旦设计完成就无法修改,因而只适用于步骤非常明确、对响应速度要求非常高的情况,而普通情况下,推荐将大部分功能和步骤的实现交由手动方式完成,以提供最佳的灵活性和容错性。

由 CPU 向设备配置的方向上,若存在跨时钟处理,则在设计时应防备程序员连续多次配置同一个寄存器。例如以下 C 程序,会连续配置同一个地址,假设该地址是 rld_dat 的地址,在极端情况下,上一次跨时钟握手尚未完成,下一个配置已到来,此时就会发生逻辑混乱,造成中间状态无法恢复。如果 refclk 非常慢,会导致握手时间非常长,这种现象就容易发生。解决办法有两种,一种是将 rld_dat 的握手改为异步 FIFO,其深度取决于预估程序员连续配置的长度。另一种是将 PREADY 拉低,使总线暂停,直到握手结束后再重新置高,这样,在第一句程序执行完之前,第二句程序不会执行,从而保证了握手同步有序进行。在设计时,需要考虑应急预案与面积成本的平衡。应急预案越多,设计的可靠性就越高,越不容易出现意外情况,但同时,面积也会增加,有时还会拖慢程序的执行速度。因而在基于需求的设计中,需要详细理解用户需求,必要时应该在模块设计说明文档中注明禁止用户操作的行为,从而节省设计本身的成本,降低复杂度。在本例中,使用了拉低 PREADY 的方法。当用户配置了 rld_dat 后,手动发起 forc_rld 进行同步。同步过程产生忙信号 rld_en_busy。在发起 forc_rld 和同步忙期间,PREADY 为低,直到同步握手完成后拉高。

```
*((unsigned int *)0x40000001) = 100;
__ISB();
*((unsigned int *)0x40000001) = 200;
__ISB();
*((unsigned int *)0x40000001) = 300;
```

注意 上述C语言示例，若中间没有ISB间隔，则编译器一般会只执行最后一句，前面两句会被编译器视为无效语句。编译器的优化在其选项中，分为00～03共4个级别，优化力度随级别上升，但若语句中有ISB指令间隔或将地址声明为volatile unsigned int *类型时，表示程序员希望这3句都被依次执行。

在从设备向CPU反馈状态的方向上，处理方式与上文方法有所不同。在本例中，内部计时cur_cnt需要反馈，以使用户随时了解设备内部的计时情况。另外，auto_rld是本次计时周期结束的标志，也需要跨时钟同步到APB时钟域，以驱动同为APB时钟域的中断信号。cur_cnt是数据状态信号，auto_rld是脉冲信号，需要用不同的同步方法，因此本例中使用总线同步模块sync_bus同步cur_cnt，使用脉冲同步模块sync_sig同步auto_rld，具体设计详见2.23节。值得研究的是反馈状态时的策略。auto_rld脉冲是间隔出现的，不可能出现连续的脉冲，但也要防备由于rld_dat设置数值太小，使出现auto_rld的频率加快，导致握手混乱，因此，在驱动同步时，产生了忙信号auto_rld_busy，并反馈到输入端，当上一次同步未完成时，即使auto_rld出现，也不能发起下一次同步。对于脉冲信号的同步，不适合使用异步FIFO。从auto_rld同步到产生中断状态的整个过程，并没有CPU参与，属于设备内部行为，因此不应该影响到APB总线上的任何信号，特别是PREADY。对于cur_cnt的反馈，可以选择两种方式。一种是仅当CPU读它时，才发起反馈和同步，如果同步时间不够，就拖延PREADY。另一种是真正将其作为一种状态，凡是状态，就是不论CPU读不读，都会反映到寄存器上。本例中用的是第2种策略。为了让状态不停地反馈，特别产生了一个周期性脉冲信号cur_cnt_vld，其波形类似时钟，工程上常称为toggle信号，它每隔一拍就驱动一次cur_cnt同步，若同步频率太高，则用忙信号cur_cnt_busy予以限制，以保证同步完成。

注意 同步模块产生的忙信号又反馈回同步模块的驱动，容易造成组合环。需要在设计时合理地插入触发器，以打破可能出现的环路。

像本例中Timer设备这样，CPU的配置字先是被存储在APB域的寄存器，然后在合适的时机同步到内部寄存器，就是所谓Bufferable的体现。一般的设备，CPU在HPROT中都会给予Bufferable权限，此类操作不会影响设备自身工作及CPU对设备的控制。一个Bufferable设备的结构如图5-27所示。

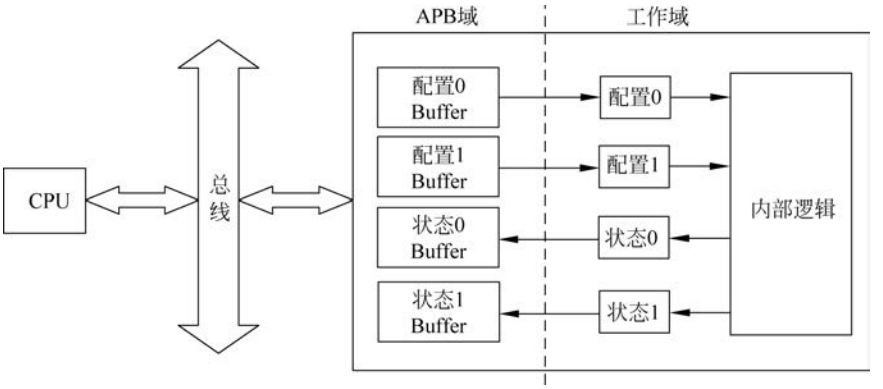


图 5-27 Bufferable 设备的结构

33min

5.20 SoC 芯片时钟与复位信号的设计

设计 SoC 芯片,必须注重片内时钟与复位策略。从单个模块设计来看,只要不是跨时钟模块(握手或异步 FIFO),一般只需一个时钟输入和一个复位输入,但从芯片总体来看,需要多种时钟共同输入来驱动芯片在不同的要求下工作。可选的时钟源包括 RCO、晶振、PLL 等。RCO 或晶振一般配备两种频率,一种是兆赫兹级,另一种是千赫兹级的。兆赫兹级时钟用于驱动芯片正常工作,千赫兹级时钟用于在芯片休眠时有一个能维持其内部缓慢运行的时钟。当芯片内部需要用到比 RCO 或晶振更高的频率时,就需要 PLL 上场,因此,SoC 芯片中时钟的来源可以归纳为 5 个,分别为兆赫兹级 RCO、千赫兹级 RCO、兆赫兹级晶振、千赫兹级晶振、PLL。PLL 需要输入参考钟,而 RCO 和晶振的兆级时钟都可以作为 PLL 的参考钟使用。这 5 路时钟的起源如图 5-28 所示。RCO 的特点是芯片内部可以产生,但其频率不够稳定,抖动比较大,其精度是±1e4ppm,即频率在±1%范围内抖动。当对芯片工作频率精度要求不高时,可以使用 RCO 或以 RCO 作为源头的 PLL 作为主要时钟源,这样,片外的晶振就可以省掉,从而可以降低方案成本。例如该产品单纯控制 GPIO 电平的高低电平,或者仅操作一些慢速设备如 UART、I^2C 等,可以只使用 RCO。当产品对频率精度要求高时,必须使用外部晶振作为时钟源,其精度是±20ppm,即频率在±0.002%范围内抖动。同时,RCO 容易随着周围温度的变化而产生频率改变,称为温漂,这种频率偏移不是抖动,而是平均频率的偏移。在必要时需要产品暂时停止工作,校准后再重新工作,这是一般产品所不允许的,而晶振的温漂特性则稳定很多,基本不用考虑频率偏移的情况。千赫兹级 RCO 或晶振一般选择 32.768kHz 作为频率,因为该频率恰好是 2 的 15 次方,即对它进行 15 次 2 分频就可以得到 1Hz 频率,可方便地翻译为人们日常惯用的秒、分、时等计时单位。该频率作为给普通用户提供定时的频率非常合适,比它频率高的,功耗大,休眠时不希望有如此大的功耗,比它频率小的又太慢,影响了计时精度和唤醒反应时间。PLL 在以兆赫兹或几十兆赫兹为参考的情况下,可以很方便地输出百兆赫兹频率,给芯片的快速处

理提供时钟。

注意 目前有一种基于MEMS微机电的振荡器可以取代晶振。晶振一般是固定频率的，如果要改变频率，则必须更换晶振。这种振荡器不需要更换，只需更改配置便可以更改频率，因而可以实现一个电路板上的多种频率动态输出。其精度甚至达到了0.01ppm，而且具有良好的抗外部振动性能。

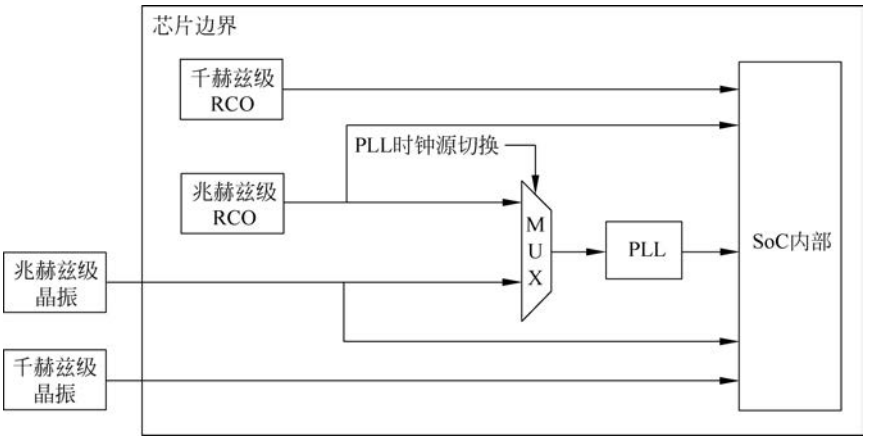


图 5-28 SoC 芯片时钟源

CPU和总线只能有一个时钟源，上述5个时钟源该如何选择呢？这里需要分阶段来讨论。在上电阶段，一般使用兆赫兹级RCO作为启动时钟供给CPU，这样，即便电路板上没有焊接晶振，芯片也可以正常启动，并正常执行Boot程序。当Boot执行完毕后，进入App程序执行阶段，可以根据用户需要，将切换到兆赫兹级晶振或PLL上。当CPU休眠后，会切换到千赫兹级RCO或千赫兹级晶振，一般休眠时钟用RCO提供，因为晶振驱动电路也有一定功耗。有些应用在休眠时也需要准确的时钟，例如用户希望每隔一分钟芯片解除休眠一次，进行某种操作再回去休眠，这分钟的时间要求较为严格，若按照RCO的精度，每100分钟就偏差一分钟，用户无法接受，此时需要用晶振提供32.768kHz时钟。切换时钟所使用的配置也可以放在一个AHB设备或APB设备中，CPU通过配置该设备从而设置了自己的时钟源。若用户使用了2.26节中介绍的无毛刺时钟切换电路，则不必担心在配置过程中发生CPU死机的可能。

一种时钟选择的架构方案如图5-29所示。5个时钟被放在统一的时钟复位模块中进行选择，该模块的时钟处理部分实际为一个5选1 MUX，该MUX可以由2.26节介绍的2选1 MUX级联而来。将选择后的信号作为CPU和整个SoC系统的主要工作时钟。图中PLL是经过一个时钟分频器后才进入MUX中的，原因是PLL输出的频率为数百兆赫兹，一些CPU由于本身架构和流片工艺的限制，在上百兆赫兹频率下无法满足建立和保持时间，因此必须将PLL降频使用。那么，既然CPU不需要PLL那么高的频率，为什么有些时候不是直接使用RCO或晶振的兆赫兹级时钟，而是一定要用分频后的PLL时钟呢？原因

在于总线上挂的设备。图中给出的设备是一个像 5.19 节介绍的 Timer 一样,同时使用两个时钟的设备,在总线接口处使用 CPU 时钟,在内部工作中,使用 PLL 直接输出的百兆赫兹时钟。若该设备内部也用了跨时钟域同步技术,则 CPU 的时钟与工作时钟 PLL 无关,CPU 时钟源头可以任意选择,但若设备内部未做跨时钟域同步处理,则两个时钟的交互就容易产生亚稳态。只有保证这两个时钟来自同一个时钟域,才能避免亚稳态的发生。因而,当 SoC 芯片中启用图中的双时钟设备时,连 CPU 的时钟也要改为 PLL 分频的时钟,以保证该设备的两路时钟同源。某些设备之所以不做跨时钟处理,是因为速度,即高速设备需要与 CPU 快速交换信息,而握手或异步 FIFO 都会拖延信息交互,此时,只能取消跨时钟机制,并把 CPU 时钟改为 PLL 同源时钟。由于 PLL 的参考源来自 RCO 或晶振,所以广义上,PLL 天然与 RCO 或晶振同源,但在实际电路中,往往将通入数字电路之前的时钟均视为异步时钟,因为模拟电路中的时序不容易分析,在数字 SignOff 时,这段模拟电路部分不参与时序验收,无法检验其同步性,将其视为异步比较稳妥。既然是异步的,在需要同步时,要么抛弃异步信号改用数字中同源生成的信号,要么做跨时钟处理。本图使用了抛弃异步信号的策略。

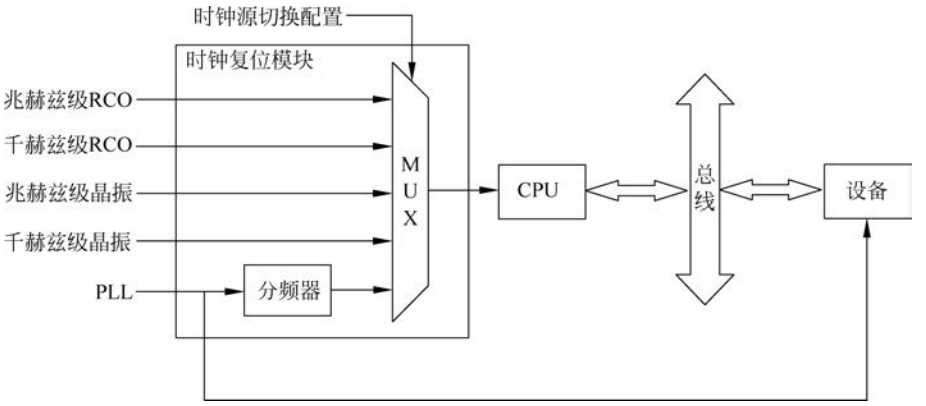


图 5-29 芯片内部时钟选择方案

图 5-29 中 CPU 只有一个时钟输入,而一般的 CPU IP,在接口上会有多个时钟输入,这些时钟是什么关系,用一个时钟表示多个时钟是否贴切?要获得答案,需要理解 CPU 的每个时钟分别驱动了 CPU 内部的哪些组件。以 CM0 为例,它需要的时钟有 5 个,分别命名为 FCLK、SCLK、HCLK、DCLK、SWCLKTCK,它们所控制的内部模块如图 5-30 所示。FCLK 即自由时钟(Free Clock),一般不会给它加门控限制,因而它的波形完全取决于外界给 CPU 提供的时钟。FCLK 直接控制的模块只有 WIC,但间接控制着整个 CPU,因为除 SWCLKTCK 以外,其他 3 个时钟都来自 FCLK。之所以驱动 WIC 的时钟不加门控,是因为 WIC 是 CPU 从深睡到唤醒的关键组件,只要 CPU 有时钟输入,就应该不被阻挡地使 WIC 得到驱动。SCLK 即系统时钟(System Clock),它来自 FCLK,中间加有门控,它直接驱动 NVIC 模块,间接驱动 CM0 核。HCLK 是 AHB 时钟,它来自 FCLK,中间加有门控,它驱动着 CM0 的核心和 AHB 总线设备(包括下属的 APB 总线设备)。DCLK 也来自

FCLK，中间加有门控，它驱动着 Debug 模块。SWCLKTCK 是与另外 4 个时钟异步的时钟，它来自仿真器，即 JLink、ULink 等外接设备，通过 JTAG 或 SWD 接口进入 CPU 内部，因此，SCLK、HCLK、DCLK 都是 FCLK 附加门控后衍生出的时钟，而 SWCLKTCK 来自芯片以外，在正常使用中不输入，仅当连接仿真器时才有输入。当表示 CPU 的时钟时，可以使用图 5-29 所示的方式，只以一根时钟线表示 CPU 的全部时钟，此时钟线即为 FCLK，上文所述 RCO、晶振、PLL 的切换，切换的对象就是 FCLK。

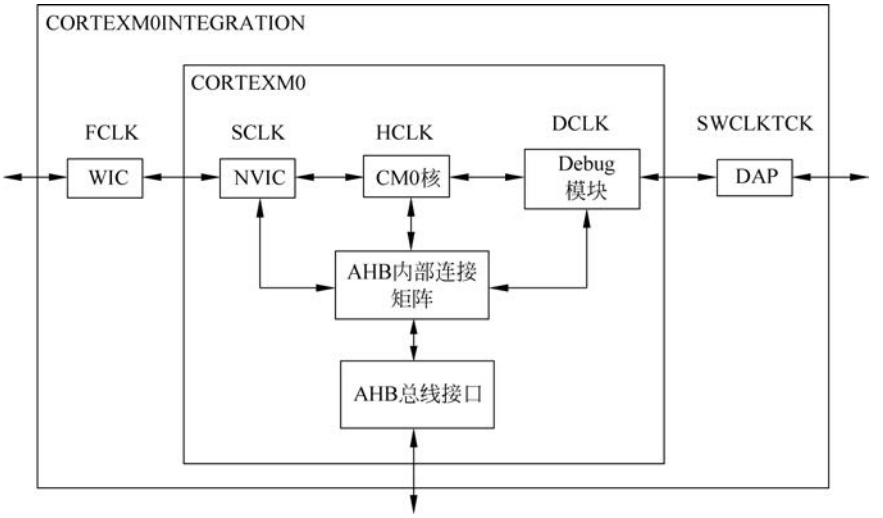


图 5-30 CM0 内部的时钟域

SCLK 与 FCLK 的区别建立在系统中存在 WIC 模块的前提下，若 WIC 不存在(集成时将 WIC 宏配置为 0)或 WIC 不使能(将 WICENREQ 拉为恒 0)，则 SCLK 和 FCLK 合并，此时 SCLK 也像 FCLK 一样不加门控，同时，该系统也放弃了进入深睡这一选项。SCLK 的门控条件非常宽泛，即大多数时间 SCLK 有时钟流过，仅在特殊省电场景下 SCLK 没有时钟。这一特殊的省电场景由 4 个条件构成，其一是 CPU 已经进入了深度休眠，其二是 WIC 模块被使能，其三是 CPU 的 WAKEUP 信号输出为 0(并无唤醒动作)，其四是仿真器并未连接本芯片(CPU 接口信号 CDBGPWRUPREQ 用来显示仿真器是否连接，当连接时，该信号输出 1)。由此看来，SCLK 仅在 CPU 进入深睡到被唤醒的中间时段被关闭，牵连 NVIC、CM0 内核、总线时钟均被关闭。

HCLK 在 CPU 进入休眠时关闭(普通休眠和深度休眠)，当 CPU 正常工作时打开，因此，相对于 SCLK 只在深睡时才关闭的特性，HCLK 的不同在于它在普通休眠下也会关闭。CPU 接口输出的状态信号 GATEHCLK 就体现了 CPU 是否已进入休眠的特性，因此 HCLK 的门控可以直接使用 GATEHCLK 控制，当它为 1 时，说明 CPU 进入了休眠，所以关闭 HCLK，反之则开启。

DCLK 在仿真器连接本芯片后开启，其他时间关闭，因此，也可以用 CPU 的输出信号 CDBGPWRUPREQ 作为指示，当它为 1 时，说明有仿真器连接，可以开启 DCLK。

除 CPU 接口需要的 5 个时钟外,还有 APB 时钟 PCLK,它还衍生出了自己的门控时钟 PCLKGated。PCLKGated 的门控来自模块 cmsdk_ahb_to_apb 输出的 APBACTIVE 信号,当 AHB 对 APB 桥有操作时,PCLKGated 才有时钟,所以 PCLKGated 的使用范围仅限于可读可写的配置寄存器,而 APB 器件内部的正常工作需要时钟持续驱动,不能使用 PCLKGated,而对于 PCLK,也并非简单地使用 HCLK。有些设计考虑到 APB 设备速度慢的特点,将 HCLK 进行分频后作为 PCLK 使用。以 HCLK 作为 PCLK 的源头,意味着当 CPU 进入休眠后,APB 设备也将没有时钟驱动,某些设备还是希望在休眠中继续运行,最起码要保留发出中断的能力,只有中断才能激活 CPU,在这种情况下,可选择 SCLK 或 FCLK 作为 PCLK 的源头时钟。

一个设备可能既需要总线接口时钟,又需要其他驱动时钟,例如 ADC、RTC 等都有自己独立的工作时钟,这些时钟也需要在时钟复位模块中为它们单独设置门控,以便在不用到它们时将时钟关闭以减少功耗。

综上,时钟复位电路中的时钟部分,产生了 SoC 系统中需要的全部时钟。相比于分散的时钟产生方式,这种集中产生方式更方便管理、修改、时序约束。门控器件一般选择带 Latch 功能的 ICG,其时序如图 2-51 所示,它可以有效地避免开关时钟时产生毛刺。ICG 在芯片工艺 PDK 的标准单元库中可以找到。综合以上因素设计的 SoC 时钟控制部分如图 5-31 所示。常用的内部使用千赫兹级 RCO 或千赫兹级晶振的设备如看门狗,内部使用 PLL 等高速时钟的设备如高速 Flash、高分辨率的 PWM 等。除图中所画时钟外,还可能存在其他外来时钟,例如 ADC 一般也会有自己的时钟,但不参与产生 FCLK 的 MUX 选择,仅单独为 ADC 驱动设备内部使用。时钟复位模块中的时钟 MUX,不一定只有一个。图中的这个

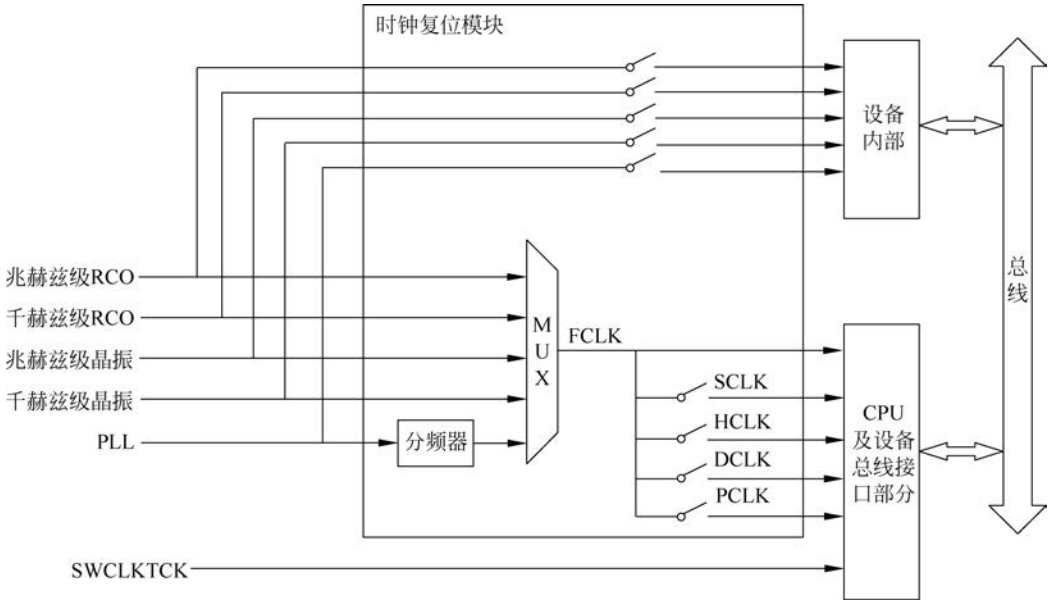


图 5-31 SoC 芯片的时钟策略

MUX 是该模块的重点，即为 CPU 提供的时钟 MUX，但在设计中，外围设备也可能需要时钟选择，例如某设备会在兆赫兹级 RCO 和兆赫兹级晶振之间做 MUX 选择，类似的排列组合很多，图中已省略，但设计者需要意识到可能出现的复杂情况。

关于复位信号的起源已在 2.24 节进行了详细论述，可以简单概括为在模拟电路中，电源电压经过比较器产生了一个 POR 信号进入数字电路中，作为数字电路的整体复位信号。另外，为了在芯片出现运行故障时可以免去拔插电源的麻烦（有些应用甚至不允许拔插电源），一般还会设置一个芯片引脚作为复位引脚，正常情况下该引脚恒为 1，当需要重启芯片时，将该引脚下拉为 0，并保持一段时间，所以芯片的复位是双源制的，POR 负责芯片上电时的复位，复位引脚负责热重启时的复位。

在芯片内部，根据 2.25 节介绍的异步复位同步释放知识可知，凡是异步时钟源，都需要产生单独的复位，不可直接使用 POR 复位。图 5-31 中的异步时钟共有 7 个，分别是 5 个 MUX 源头时钟、SWCLKTCK 和 FCLK。为什么 FCLK 也当作异步时钟呢？是因为 MUX 的原因导致无法确定 FCLK 的具体时钟源，因此也要为它单独设置一个复位信号。SWCLKTCK 一般不必产生复位，因为 CPU 顶层没有此时钟的复位信号接口。已知一个时钟，将一个异步复位信号同步到该时钟下的 RTL 代码可参看 2.25 节，总的原则和目标是当 POR 或复位引脚发生复位时，芯片内部不同时钟域都会同时发生复位，而当 POR 和复位引脚都不复位时，芯片内部的不同复位信号会根据各自所属的时钟建立情况逐一解除复位。

由于芯片内部会产生多个复位，而它们的源头只有两个，于是就形成了一棵双根复位树，其示例如图 5-32 所示。从图中可以清晰地看出复位的源头来自 POR 和复位引脚。来自引脚的复位信号称为引脚复位信号。由于引脚复位信号受片外电平控制，所以必须考虑在它上面可能存在的抖动，例如当手指按动复位开关时会产生抖动，因此，需要为它专门进行去毛刺处理，然后才能放心使用。去毛刺处理本质上也是个时序电路，因此也需要复位，

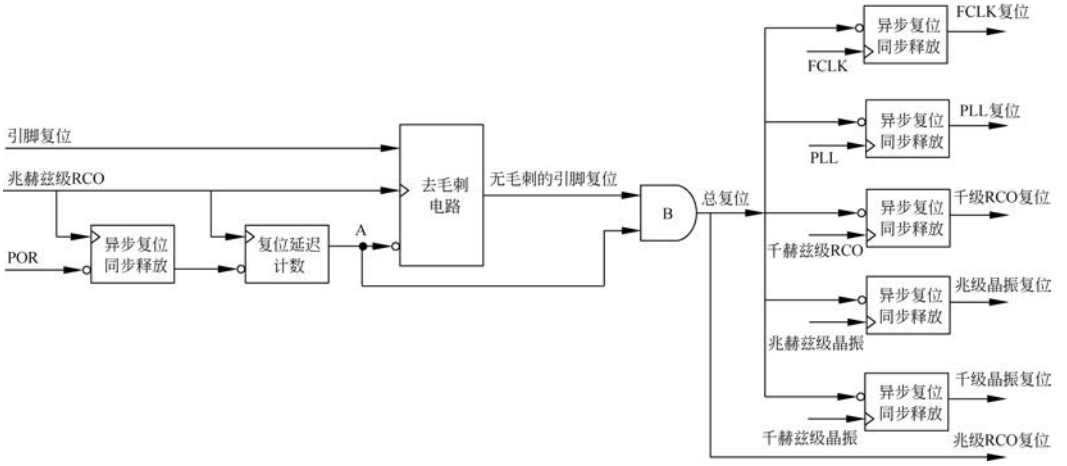


图 5-32 SoC 芯片的复位树

它的复位源也由 POR 提供。最后,两路源头在 B 点的与门合并,形成芯片的总复位信号,再由该信号向外衍生出不同时钟对应的复位信号,所以对于 CPU 和各设备而言,它们不关心复位来自 POR 还是引脚,而是只关心总复位信号。B 点之所以要用与门,是因为复位是低电平有效,不论是 POR 还是引脚复位,只要其中一个为 0 就会发生复位动作。

对 POR 进行异步复位同步释放操作,以及之后进行复位延迟计数操作,最终得到 A 点的信号是整个芯片复位的基础。异步复位同步释放及复位延迟计数都已在第 2.25 节的示例中进行了描述。计数的目的是延迟复位一段时间,以免刚上电的时钟信号尚未完全稳定就进入运行阶段,继而出现错误。使用计数器也存在一定风险,在时钟不稳定阶段可能会出现计数上的混乱,或虽然计数是对的,但采样计数值时错误,本质上都是进入亚稳态后又恢复,导致计数并不是设计所期望的逐拍加一,而是可能出现跳变。为了避免这种情况,一方面,要求模拟电路给出 POR 时,已经做了一定的延迟,另一方面,数字计数器的位宽要宽一些,因为只有一种条件可以跳出计数、解除复位,因而位宽越宽,在计数值随机变化后满足跳出条件的可能性就越小,复位越安全,但同时,位宽太宽会拉长复位时间,设计者需要根据具体的电路特征决定计数器的宽度。

对 POR 进行同步释放的驱动时钟,在图 5-32 中选择了兆赫兹级 RCO,原因是该时钟为上电时就已经存在的时钟,而晶振则未必焊接,其频率比千赫兹级 RCO 高,因而可以进行较精细计数和去毛刺,但是,若设计者希望在芯片进入普通休眠时引脚复位仍然能够去毛刺,则此处时钟应改为千赫兹级 RCO,因为只有它在普通休眠时仍在工作,其他时钟都已经关闭了。在芯片进入深度休眠时,全部时钟都会停止,此时,如果用户按下复位键,由于去毛刺电路停止工作,总复位信号不会出现复位动作,所以时钟也不会被激活,因而此时的引脚复位还应与模拟电路的时钟使能开关相连,使复位动作触发时钟工作,并激发 WIC 唤醒 CPU,此时,引脚复位是没有做去毛刺的。

一个去毛刺电路的 RTL 设计如下例所示,其对应的时序如图 5-33 所示,其中,key 是带毛刺的信号,key_no_glitch 是无毛刺的信号。毛刺定义为 key 信号上短于一定宽度的脉冲,包括高脉冲和低脉冲。毛刺的宽度由 cfg_cnt 来确定,计时单位是时钟周期。例如,将 cfg_cnt 定义为 10,时钟频率为 1MHz,则毛刺最大宽度为 10μs。由于作为通用模块设置,cfg_cnt 的具体数值未定,因而需要设置一个 CNT_WID 参数,在例化时根据具体需要设定 cfg_cnt 的数值范围。key 的初始值在不同的应用场景中也不同,因此又设置了 INIT_VAL 参数来自定义初始值。模块输入的 key 如果来自异步时钟,则可以先进行电平跨时钟,得到 key0,内部只以 key0 作为输入。key0 的变化被检测出来,作为 key_change。内部计数器 pulse_cnt 作为毛刺计数,当 key0 变化时,pulse_cnt 从 1 开始重新计数。当计数到最大值时不从头计数,而是保持最大值 max 不变。图 5-33 中以 m 作为 cfg_cnt 设置,当计数器计到 m 后,deglitch_done 信号置 1,说明输入信号已经维持了规定长度的恒电平,该阶段的去毛刺结束,输出信号将等于输入信号。当 key0 发生变化时,last_value 会保留 key_no_glitch 的上一种状态,这使 key_no_glitch 可以在 key0 变化时暂时保持其上一种状态,直到 deglitch_done 再次置 1 后才能等同于输入信号。若输入电平持续时间比较短,即判断为毛

刺，则不会引起 deglitch_done 置 1，key_no_glitch 就一直保持不变。图中 A～E 表示了输入的不同电平，其中 A～C 的电平足够长，pulse_cnt 都数到了 m，因此输出时也体现了这些电平。D 阶段的脉冲较短，计数器未数到 m，因此 D 阶段的电平被模块过滤掉了。去毛刺的代价是输出延迟，本例中，输出相对于输入延迟了 m＋1 拍，这也体现了信号与系统课程中因果系统的含义，即有因才会有果，一个模块只能处理历史信息而无法处理未来信息，因此必须等待毛刺发生才能处理，模块不可能在毛刺未出现时就判断一个电平是毛刺。从这个意义上说，所有系统，包括纯组合逻辑系统在内，都属于带有延迟的因果系统。设计只能通过减少打拍或减少组合逻辑的层级来减少延迟，但无法消灭延迟。两个数据如果分时间先后到达模块，则只能将早到的数据延迟到与晚到数据相同的时间后再进行处理，不可能将晚到的数据提前。对于带反馈的系统，例如 PLL、PID 等电路，理想模型下它们是实时反馈的，但真实电路是含延迟的，这会造成理想仿真与实际电路在性能上的差异，需要设计者注意。

```
module sig_deglitch
#(
    parameter   INIT_VAL = 1'b1 ,
    parameter   CNT_WID  = 5
)
(
    input                       clk             ,
    input                       rst_n           ,
    input       [CNT_WID-1:0]   cfg_cnt         ,
    input                       key             ,
    output  reg                 key_no_glitch
);


//-----------------------------------------------------
wire                            key_change      ;
reg             [CNT_WID-1:0]   pulse_cnt       ;
reg                             last_value      ;
reg     [1:0]                   key_syn         ;
wire                            key0            ;
wire                            deglitch_done   ;
reg                             key0_r          ;

//-------------------   sync   -------------------------------------
always @(posedge clk or negedge rst_n)
begin
    if (!rst_n)
        key_syn <= {2{INIT_VAL}};
    else
        key_syn <= {key_syn[0], key};
end
```

```
assign key0 = key_syn[1];

always @(posedge clk or negedge rst_n)
begin
    if (!rst_n)
        key0_r  <= INIT_VAL;
    else
        key0_r  <= key0;
end

assign key_change = key0 ^ key0_r;

//---------------------  sync  ----------------------------------------
always @(posedge clk or negedge rst_n)
begin
    if (!rst_n)
        pulse_cnt   <= {CNT_WID{1'b0}};
    else
    begin
        if (key_change)
            pulse_cnt <= 1;
        else if (~(&pulse_cnt))
            pulse_cnt <= pulse_cnt + 1;
        //未写的 else 是溢出保护,等同于 pulse_cnt 等于全 1 时,计数器保持住
    end
end

assign deglitch_done   = (pulse_cnt >= cfg_cnt);

always @(posedge clk or negedge rst_n)
begin
    if (!rst_n)
        last_value <= INIT_VAL;
    else if (key_change & deglitch_done)
        last_value <= key0_r;
end

always @(posedge clk or negedge rst_n)
begin
    if (!rst_n)
        key_no_glitch <= INIT_VAL;
    else
    begin
        if (deglitch_done)
            key_no_glitch <= key0_r;
        else
            key_no_glitch <= last_value;
    end
end

endmodule
```

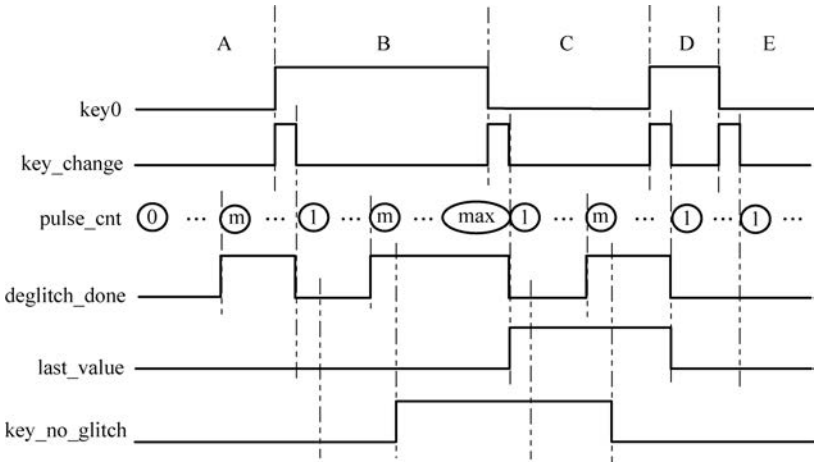


图 5-33 去毛刺电路时序

5.21 SoC 芯片的休眠策略设计

45min

随着便携式、可穿戴式设备的普及，依靠电池供电的芯片越来越多。这些芯片都要求有超长的待机时间，即在非工作状态下，尽量减少功耗，因此，SoC 芯片的休眠、待机、省电策略变得越来越重要，不论设计何种应用场景的 SoC，都必须考虑到该问题。

休眠是在芯片处于非工作状态下采取的降功耗措施，因此，在讨论休眠问题之前，必须先定义非工作状态，但可惜的是，非工作状态的概念比较模糊，无法给出确切的定义。例如，手机屏幕关闭的状态下，操作系统会控制多核 CPU 中的一些核进入非工作状态，而另一些核仍处于工作状态；屏幕驱动芯片和摄像头驱动芯片更有可能处于非工作状态；由于仍然需要接听电话，5G 等通信芯片保持工作状态，而暂时无通信需求的蓝牙芯片可能处于非工作状态。这是从系统宏观层面看到的工作状态。微观角度，例如研发的目标是手机中的显示屏驱动芯片，它的非工作状态定义比较清晰，即屏幕处于关闭状态，但即使是它也需要定义多种非工作状态的模式，例如用户希望尽量快速点亮屏幕，这就需要使芯片仅仅进入普通休眠模式，若允许延长一些点亮时间，就可以定义深度休眠模式，从而进一步节省功耗。因此，非工作状态在不同的芯片应用中有不同的定义，一旦芯片进入非工作状态，还必须确认它需要进入何种程度的休眠。休眠程度大体可分为普通休眠和深度休眠。普通休眠也称为浅睡，它不会停止 CPU 时钟，但会将时钟频率降低。保持时钟的目的是使一些依赖时钟才能驱动的外设也能正常发起中断将 CPU 唤醒，这样，唤醒 CPU 的手段就有多种选择。例如 UART、I^2C、GPIO、Timer 等都可以用来唤醒 CPU。在唤醒方便的同时，由于保留了时钟，功耗无法进一步降低。深度休眠也称深睡，是完全将 CPU 时钟切断的策略，从而彻底节省了动态功耗，一般在切断时钟的同时，芯片的供电电压也会降低，例如从 1.8V 降到 1.6V，或者用驱动能力降低等措施以进一步减少静态功耗，有些芯片会彻底切断某些模块

的 VDD 电源通路,以便节省这些模块的静态功耗。这些措施虽然可以大幅降低功耗,但也减弱了 CPU 的唤醒能力,延长了唤醒时间。减弱唤醒能力是指一些依靠时钟才能驱动中断的设备无法发出中断,因此不能利用它们来唤醒 CPU,一般在深睡后只能用外部输入激励的方式进行唤醒。延长唤醒时间是指在切断了某些模块的电源后,使其再次启动并稳定到工作状态需要额外付出一定时间。值得注意的是,从浅睡中唤醒未必比从深睡中唤醒时间短,因为浅睡中时钟频率特别慢,唤醒动作通过该频率进行,操作较慢,而深睡可以从无时钟直接切换到快速时钟,所以速度可能会快于浅睡。决定深睡唤醒速度的因素是时钟和电源的建立时间,因为深睡中时钟产生模块和部分电源关闭,它们从开启到稳定的时间在不同的设计中存在差异,这些差异,可能是由设计水平导致的,也可能是由指标要求导致的,即有着较高稳定性要求的设计,其启动和确认稳定的时间会较长。将芯片的非工作状态分为浅睡和深睡只是粗略的区分方法,由上文的介绍可知,节省芯片功耗的方式和手段多种多样,设计者可以将这些手段进行排列组合,然后定义成多种层次的省电模式。另外,非工作状态也不一定要休眠,一些芯片中未考虑休眠因素,不工作时,时钟和电源都保持不变,这样的芯片在市场上大量存在。

注意 芯片的关机状态,即切断总电源的状态,也可认为是一种非工作状态,但一般不将其涵盖在讨论范围之内,因为此时芯片的功耗是零,但芯片内部的运行信息均已丢失。这里讨论的非工作状态,即休眠状态,是指芯片仍有供电,其在工作状态下的运行数据仍然可以在 RAM 中找到,芯片唤醒后,能从上一个工作状态继续工作,而不是从初始态工作。

上文的休眠指的是 SoC 系统层面的休眠,而在 CPU 内部也定义了休眠机制。在图 5-30 中给出了 CM0 内部各模块时钟域的划分,并在 5.20 节详细介绍了 FCLK、SCLK、HCLK、DCLK 的时钟门控方法。从这些背景知识可知,WIC 模块在深睡情况下仍然工作,NVIC 和 WIC 在浅睡情况下仍然工作,CM0 核只在不休眠时工作。这里的深睡、浅睡都是 CPU 内部的定义,不同于 SoC 系统层面的休眠。CPU 定义休眠,不以时钟有无、电源是否切换为判断依据,而是以 CPU 的 SLEEPING 和 SLEEPDEEP 两个状态信号为依据。当 SLEEPING 为 1,并且 SLEEPDEEP 为 0 时,说明 CPU 处于浅睡状态。此时,CPU 中除了 NVIC 和 WIC 在工作外,其他模块均停止运行。当 SLEEPING 为 1,并且 SLEEPDEEP 也为 1 时,说明 CPU 处于深睡状态。此时,CPU 中只有 WIC 工作。要想让 CPU 进入休眠状态,需要输入指令 WFI 或 WFE,其中 WFI 可以用中断唤醒,WFE 除了用中断还可以通过 CPU 上的 RXEV 口接受脉冲来唤醒(用于多核连接的 CPU 中)。至于进入的休眠状态是深睡还是浅睡,不由 WFI 或 WFE 决定,而是由 SCS 寄存器的 SCB 中的 SCR 第 2 位 SLEEPDEEP 来决定,设为 0 表示一旦进入休眠就进入浅睡状态,设为 1 表示一旦进入休眠就进入深睡状态,详情见表 5-4。需要注意的是,CPU 定义的休眠对系统没有任何要求,换而言之,CPU 进入休眠后,系统设计可以选择各种策略,甚至选择时钟和电源保持不变,对 CPU 的休眠和工作无动于衷,此时,也会发现芯片功耗有所降低,因为 CPU 确实已经停止工作,内部时钟被切断,但降低后的功耗仍然无法忽略。一般,衡量芯片的低功耗水平都是

微安级(用测量电流来替代计算功耗)的,如果达不到该程度,则功耗不能忽略。

正是由于系统休眠和 CPU 休眠是两个独立的概念,所以才需要在系统设计时将两者关联起来。简而言之,就是当 CPU 进入休眠后,它会输出 SLEEPING 和 SLEEPDEEP 状态信号。设计者需要将这两个状态信号与时钟开关、电源开关和控制进行逻辑上的关联,以便使 CPU 的休眠与系统的休眠形成联动。当 CPU 休眠后,引发时钟切换动作或关停动作,以及电源控制动作。当有唤醒信号时,首先激发时钟的切换或开启动作,以及电源动作,然后唤醒 CPU。一般会将这类控制逻辑放到一个专门的模块中,这就是常说的电源管理单元(Power Management Unit,PMU)。

PMU 实际上并不复杂,其逻辑量仅相当于一个小型模块,但在设计时必须谨慎对待,因为它不是一个纯粹的数字模块,还包括对模拟的控制,例如上文所提到的供电电压降低、电源驱动能力降低、切断某些模块的电源、时钟关闭等功能,这些都不属于数字设计的范畴,而是属于模拟设计的范畴,因此,PMU 的控制手段、控制流程,需要与模拟设计工程师进行讨论后才能制定。它要求在模拟上构建相应功能,并在数字上设计配置接口以配置这些功能。例如,在休眠时希望使用降低电源驱动能力的方式来获得省电效果,就需要在模拟上设计两种不同的 LDO(LDO 和 DCDC 是模拟电路中用于生成指定电压的模块),其中一种是大驱动能力的,另一种是小驱动能力的。当进入休眠时,由数字先开启小驱动能力的 LDO,然后关闭大驱动的,以完成一次切换。驱动小意味着可带动的晶体管数量少,若晶体管数量多,则每个管子的供电可能会不足,工作就会不正常,极易导致死机,因此,小驱动能力的 LDO 必须确保 CPU 已经进入休眠状态后才能启用。再例如,想在休眠时切断某个模块的电源,就需要在该模块的电源线 VDD 处插入一个开关,并且数字逻辑可以控制该开关。这种开关一般需要手动插入,因为在自动布局布线流程中,VDD 和 GND 是并行排列的金属线,不会在该供电线路上插入逻辑门,而且,开关的插入可能会影响芯片中各门电路的供电平衡性,即 IR Drop,导致芯片上电后不同门电路的电压不均匀,因此需要考虑更多的因素以保证芯片正常工作。

与模拟的配合方案确定后,PMU 的数字设计部分需要关注的就是开关模拟电路时的延迟情况。如在大小 LDO 的例子中,开启小驱动能力的 LDO 也需要时间,这段时间内大驱动能力的 LDO 不能立即关闭,否则芯片就会没有电压。再例如时钟产生电路,在休眠状态下,会关闭一部分快速时钟,或全部时钟均关闭,但如果电路中有一部分电路在 CPU 休眠时还会有一段时间的电平变换动作,而不是立即停止,则 PMU 不能在检测到 CPU 休眠态的同时关闭时钟,那样将把这部分电路的动作定格在工作的中间状态,而不是空闲状态,等 CPU 苏醒后,该电路会从中间状态继续工作,这是危险的,可能导致苏醒后的工作不正常。时钟的苏醒也需要等待时间,在这段时间内不允许唤醒 CPU,否则 CPU 将在一个不正常的时钟上工作。综上,PMU 设计的原则就是,当检测到 CPU 休眠后,等待一段时间或一个触发条件,分批次地关闭数字和模拟的功能模块、时钟和电源,当检测到中断后,先开启电源、时钟,等到这些设备均稳定后,再唤醒 CPU。

一个 SoC 芯片从工作态到休眠态再被唤醒回到工作态的例子如图 5-34 所示。图中用并行的 3 条状态线分别表示 CPU、数字和模拟的状态,横向为时间轴。先开始 CPU 处在工作态,数字和模拟也都处在正常工作的状态。当用户需要让芯片进入休眠时,不应直接输入 WFI 或 WFE 命令,而是要先在 CPU 控制下,通过总线配置,将一部分非关键性电路关闭。所谓关键性电路指时钟、电源、取指令相关的电路,然后输入休眠命令,此时,仅 CPU 进入休眠态,而其他电路的功耗,除已关闭的那部分电路外,均未降低。PMU 模块通过检测 CPU 的输出信号 SLEEPING 和 SLEEPDEEP 来确认它已休眠,然后它会等待尚在运行状态的电路停止,最后将这部分电路关闭,包括不用的数字内部时钟,在设计时均有门控,也都可以关闭。待数字模块全都关闭后,触发模拟模块的关闭过程,主要是时钟和电源。至此,芯片才进入真正的休眠状态,功耗降至最低,并且运行状态仍保留在 RAM 中。在图中标注的实际休眠阶段即指本状态。当用户需要解除休眠时,需要给芯片一个中断或事件,该触发信号不能像传统的设计一样直接进入 CPU,而是要先触发开启电源和时钟,等待稳定后,由电源和时钟输出一个完成信号作为唤醒 CPU 的中断信号。唤醒后的 CPU 会在开始工作之前,使用总线将它需要的数字和模拟模块全部开启。

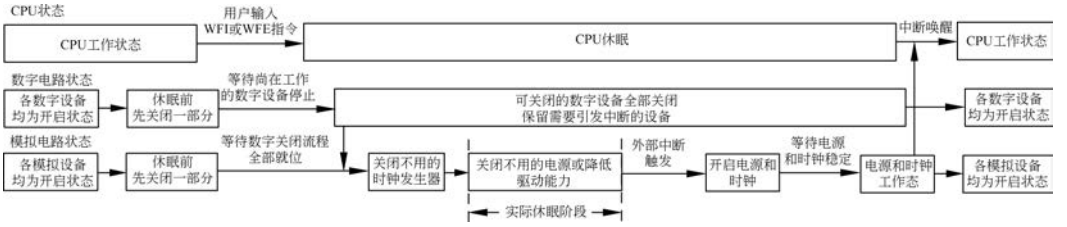


图 5-34 SoC 芯片从工作到休眠再到被唤醒的操作流程

PMU 并不会包办整个休眠唤醒流程,它只承担一部分工作。SoC 芯片总是以 CPU 为核心,所以通过总线配置寄存器开关模块的方式比较稳健,不易出现设计错误,因此,大部分的模块关闭、开启工作是在 CPU 醒着时由它自己完成的。如果 CPU 休眠,指令和程序将无法执行,这才需要 PMU 上场,而类似时钟、电源,以及休眠后还在运行的数字模块,是无法在 CPU 醒着时关停和开启的,因此只能在 PMU 中实现。总之,PMU 是一种通过逻辑门电路代替 CPU 操作寄存器开关的设备,仅在 CPU 失能的情况下工作。

图 5-34 中的中断唤醒过程是值得商榷的。按照正常的 CPU 设计集成方式,每条中断线都是可以溯源的,即某个特殊编号的中断线会触发对应的中断服务程序。在图中,不论什么中断,都会首先触发电源和时钟动作,而不是直接通入 CPU 中,CPU 的唤醒是由电源和时钟完成的标志实现的,这一过程掩盖了唤醒芯片的中断源。在实际设计中,对于中断唤醒方面,需要进行分类讨论。

当芯片处于浅睡时,由于仍然有慢速时钟的存在,所以内部设备或外部激励都可以作为中断源,更简单的方式是用 Timer 让芯片每隔相同的时间唤醒一次。为了尽量保证中断可以溯源,中断线仍然正常连接到 CPU 上,发起中断就可以快速唤醒 CPU,但同时,由于电源也需要切换,要求新切换的电源能够快速建立,能在 CPU 苏醒的同时增强供电能力。如果

电源切换太慢，则必须将中断进行打拍延迟才能输入 CPU，如果是普通中断可以延迟，但是像 SysTick 及 DAP 引起的 Debug 唤醒都发生在 CPU 内部，则无法通过外部打拍延迟，这就限制了芯片唤醒的手段，特别是 Debug 唤醒，是非常重要的唤醒方式，否则芯片在休眠时接上 JLINK 也无法使用，这是很多使用者都接受不了的。因此，对于浅睡来讲，关键的问题在于加快电源切换的速度，甚至不切换电源，保留足够的供电能力。工作时钟的开启也要同步进行，以保证时钟切换握手顺利进行。

当芯片处于深睡时，由于时钟已经全部关闭，此时限制中断来源实属无奈之举。一般以外部电平激活电源和时钟，再由电源和时钟发起中断激活 CPU，如图 5-34 所示。外部电平一般指从芯片的 GPIO 口输入的低电平或高电平。有时，为了简化数字设计，直接将该外部电平作为中断信号连接到 CPU 上，这就要求一方面电源要快速建立，另一方面输出的时钟在开始时就不能有毛刺，否则 CPU 将在时钟毛刺环境下苏醒。这就要求在模拟时钟设计中必须自己做延迟，等时钟稳定后再输出，这样的模拟设计也允许电源和时钟模块同时开启，因为在时钟稳定的同时，电源一定也稳定了，可以用一个时钟稳定的标志同时代表时钟电源双稳定。

深睡的唤醒时间未必比浅睡长，因为在深睡时，可以不进行时钟切换，直接从无时钟状态开启工作时钟，而在浅睡情况下，必须从慢速时钟切换到快速时钟，而根据第 2.26 节的介绍，时钟切换首先要在慢速时钟上进行打拍，这就限制了切换时间，因此，前文提到的电源切换速度的问题，对于千赫兹级慢速时钟的浅睡场景，实际要求是不高的，因为模拟设计的速度一般为兆赫兹级。

一个 PMU 的设计结构如图 5-35 所示，其主要结构由 CPU 休眠状态信号控制的 3 个 MUX 组成，分别是时钟选择 MUX、数字使能 MUX，以及模拟使能 MUX。数字休眠策略切换电平决定了时钟选择 MUX 和数字使能 MUX 的切换，而模拟休眠策略切换电平决定了模拟使能 MUX 的切换，即时钟和数字使能是一起切换的，而模拟使能是单独切换的。PMU 挂在 AHB 或 APB 总线上。当芯片尚在正常工作时，用户可以将这些休眠策略都配置到 PMU 的寄存器中，例如在正常工作时使用 PLL 提供时钟，数字使能开启，而在浅睡时使用慢速休眠时钟，数字使能关闭，在模拟使能中保留休眠时钟和电源，其他使能关闭，在深睡时仍然选择 PLL 时钟，数字使能关闭，模拟使能全部关闭。这些配置完成后，当 CPU 进入浅睡或深睡时，数字休眠策略切换和模拟休眠策略切换就会在 MUX 中选择相应的策略，而最终的效果与 CPU 配置寄存器的效果无异。数字和模拟休眠策略信号均为电平信号，用于维持选择状态。切换工作状态控制逻辑会根据 CPU 输出的指示信号 SLEEPING 和 SLEEPDEEP 来完成控制，当休眠信号上升时，说明 CPU 已休眠，先切换数字时钟并关闭那些不与 CPU 同步关停的设备，然后关停模拟的一些时钟和电源，此顺序与图 5-34 所描述的顺序一致。为了完成上述关停顺序，需要在逻辑中使用计数器，但当 CPU 苏醒时，即休眠信号下降时，就不能按顺序开启了，而必须一起开启。设计中需要注意两点。第一点是图中所称的数字设备仅指不与 CPU 同步关停的设备，需要 PMU 帮 CPU 关闭，图中模拟设备也仅指电源和时钟，其他的数字和模拟设备都可以在 CPU 工作时由指令程序关停或开启。

第二点是深睡时的时钟选择一般与正常工作时相同,这并不意味着深睡时有着与正常工作同样的时钟,而是说深睡时的时钟策略是不切换而直接关停,这样可以保证唤醒速度最快,免去了唤醒后再切换的时间开销。

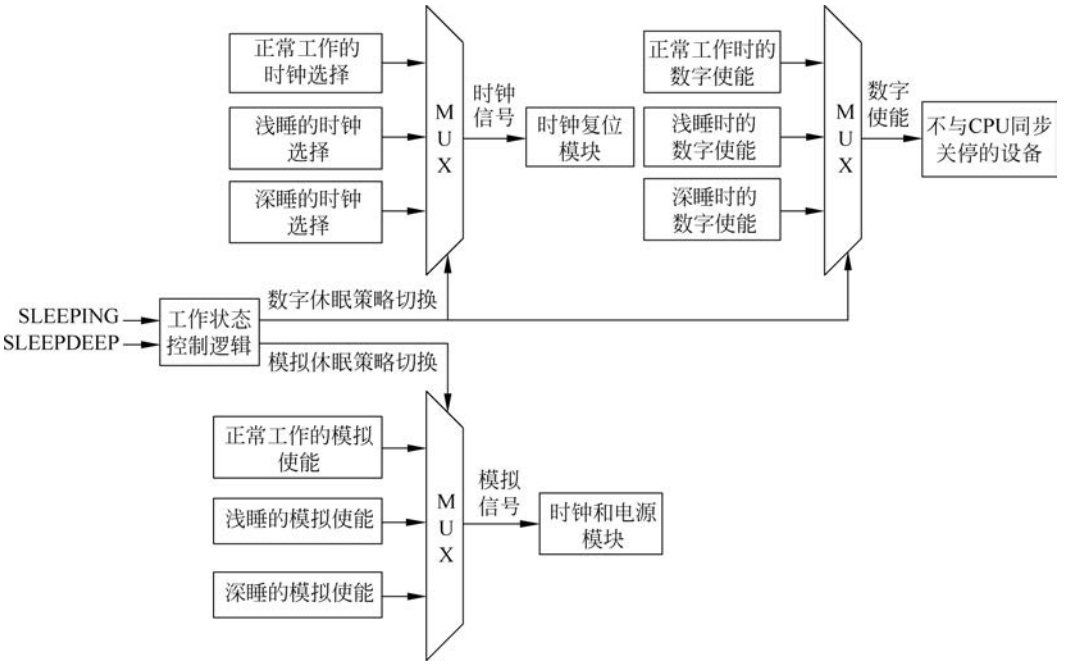


图 5-35 PMU 设计框图

深睡策略的设计需要用外部电平激励唤醒电源和时钟,即一旦外部激励电平开启,时钟和电源模块的使能将忽略寄存器配置,直接变为使能状态,这只需使用一个或门便可以做到。如果时钟没有输出,CPU 的休眠也不会被唤醒,因此不必担心 CPU 先被唤醒而后出时钟的问题,这是一种天然的保序机制。这里需要注意的是,因为没有时钟,所以去毛刺也变成不可能,因此,外部激励信号在未经去毛刺的情况下使能时钟和电源,存在一定的风险,需要在施加外部激励时做一定的防抖保护。外部激励电平可以做成 GPIO 多选一方式,即在休眠前,由用户选择某个 GPIO 引脚作为激活引脚,其他引脚不能激活,并且激活的电平也可以选择高电平或低电平。这些功能,使用组合逻辑即可实现。

在进入深睡休眠时,会彻底切断时钟,而从图 5-30 可知,当 CPU 在深睡时仍然会开启 WIC,它在 FCLK 的驱动下工作。在切断时钟时,可以选择用 ICG 门控方式先切断 FCLK,然后停止时钟发生器的输出,也可以选择直接切断时钟发生器的输出。后者在切断过程中可能会引入时钟毛刺,因为模拟电路时钟发生器往往不会做完整周期保护,如果将使能位配置为 0,则立即关停。那么,如果使用直接关闭方式,时钟毛刺会不会使 WIC 内部出现亚稳态?答案是不会。在第 2.23 节讲亚稳态产生原因时,强调过对于一个稳定不变的信号,无论以什么时钟采样,采样结果均是稳定的,即常数信号可以不经处理直接跨时钟。WIC 中 FCLK 控制的时序逻辑主要有两部分,一部分是使能,另一部分是接收到外部中断后转发给

CPU 内核。使能状态是一直保持的，中断掩码在用户不配置的情况下保持不变，中断状态在休眠前会用 CPU 将 NVIC 中全部的中断都清空，因此在 CPU 进入休眠时，其内部中断线也都是常数。在内部所有信号在休眠时均为常数的情况下，关闭时钟出现的毛刺不会造成影响。

CM0 的 WIC 使能机制如图 5-36 所示。先由 PMU 向 WIC 配置一个使能信号 WICENREQ(高电平信号，非脉冲)，WIC 会将输入的高电平转换为低电平 WICDSREQn 传给 CPU 核，CPU 核同意开启 WIC，就回复低电平 WICDSACKn 给 WIC，再过一拍，高电平 WICENACK 会从 WIC 反馈给 PMU，表示 WIC 已经开启。当 PMU 要关闭 WIC 时，可将 WICENREQ 拉低，若干拍后，WICENACK 也会反馈低电平，表示 WIC 已关闭。一般在设计中不会频繁开关 WIC，若计划使用 WIC，就在 CORTEXM0INTEGRATION 的选项中将 WIC 选项设为 1，并将 WICENREQ 固定接高电平即可。

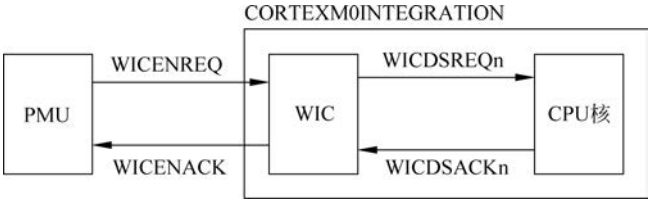


图 5-36 WIC 使能机制

WIC 的最主要功能是当外部中断到来后发出 WAKEUP 脉冲，其触发时序如图 5-37 所示。图中，WAKEUP 脉冲从中断到来后发起，持续 4 个时钟周期。中断也同时触发了 CPU 的苏醒，苏醒时间比 WAKEUP 慢两拍。WAKEUP 可以被拉到时钟复位模块中，和 CPU 的苏醒信号一起用于开启 SCLK 时钟门控，由于它比苏醒信号早发起，所以可以保证时钟在苏醒前的两拍就已开启，为苏醒做好准备。概括地说，WIC 并不会唤醒 CPU，而仅仅在接收中断后开启 SCLK，CPU 本身仍然是中断唤醒的。

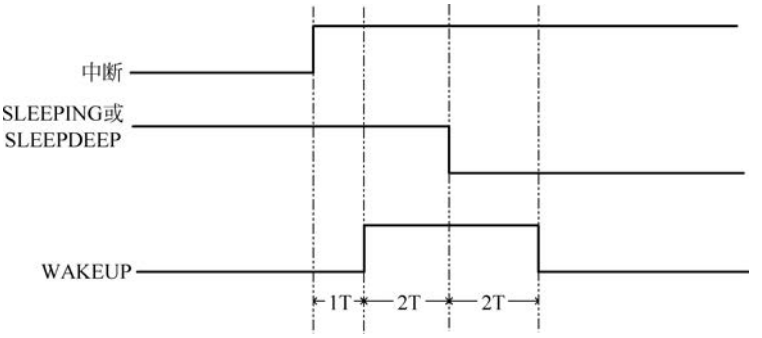


图 5-37 WIC 中 WAKEUP 信号的触发时序

除了 WIC，连接仿真器也可以将 CPU 从深睡中唤醒。连接仿真器后，DAP 会向 CPU 核发起上电请求。CPU 无法自己决定是否可上电，因为此时电源是否适合上电 CPU 并不知情，它需要请求 PMU 的意见。该请求通过 CDBGPWRUPREQ 高电平传给 PMU。

PMU 接到通知后会开启电源、时钟模块,并将 CDBGPWRUPACK 回复给 CPU。同时,CDBGPWRUPREQ 也应连到时钟复位模块中,用于开启 SCLK 和 DCLK 的门控。此时,CPU 才能被唤醒。如果设计中没有 PMU,则可以将 CDBGPWRUPACK 用 FCLK 打一拍再输给 CPU。CDBGPWRUPREQ 和 CDBGPWRUPACK 都是电平信号,仅当仿真器与 CPU 断联后,CDBGPWRUPREQ 才会由高电平恢复为低电平。对比 WIC 的使能和仿真器请求上电的过程,可以发现,WIC 使能是由 PMU 发起并由 CPU 批准的过程,而仿真器请求上电的过程是由仿真器发起并由 PMU 批准的过程,两者方向是相反的,如图 5-38 所示。在具体应用中,WIC 使能可以不经过 PMU 直接使用固定使能,而仿真器对 PMU 的请求则需要切实作用到 PMU 中,因为在休眠时连接仿真器的应用场景十分常见,若绕过 PMU 直接唤醒 CPU,则在浅睡时可以唤醒,但深睡时,CDBGPWRUPACK 没有打拍时钟,所以无法唤醒。

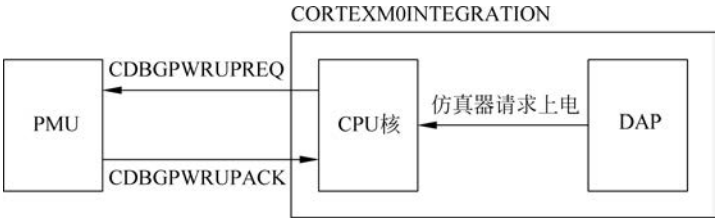


图 5-38 仿真器唤醒 CPU 的机制

从上述 SoC 芯片休眠的硬件集成策略可以看出,休眠机制是数字和模拟控制较为集中的部分,它不同于简单地配置寄存器,而是强调步骤和流程,需要数字设计者对模拟模块的开关时间、驱动特性有一定了解。基于这些了解的设计,集中于 PMU 模块,它仅在 CPU 失去运行指令能力时才发挥作用。与 PMU 相关的模块是 CPU 和时钟复位,三者协同配合才能完成休眠和唤醒的过程。浅睡的休眠机制较容易设计,并且不易出错,深睡的控制较为复杂,并且容易出现因考虑不周而引发的唤醒失败,设计时需要格外注意。

除了硬件设计,休眠策略也需要相应的软件流程。一个示例流程如图 5-39 所示。先在 SCB 中选择进入浅睡和深睡,然后选择触发休眠的条件。休眠有两种触发方式,一种是用户输入 WFI 或 WFE 指令,另一种是芯片平时就处于休眠状态,等有中断时进行处理,处理完毕后不需要用户输入任何指令也能自动进入休眠。后一种方式适合业务量少但对待机功耗和待机时间特别敏感的应用,如有源 RFID 标签这类长期不换电池、偶尔有运行需求的芯片。该选项在 SCS 寄存器的 SCB 中的 SCR 第 1 位 SLEEPONEXIT 字段。在 CPU 尚未休眠时,就应该将休眠时的策略配置好。如果即将进入深睡,则以外部 GPIO 引脚中断为主要的唤醒源,需要事先将该唤醒源配置妥当,例如,PIN MUX 切换到 GPIO 模式,引脚的输入使能打开,进行上下拉配置,配置中断唤醒的极性。还应对 NVIC 进行配置,关闭与唤醒无关的中断使能,防止其打扰芯片的休眠。浅睡的唤醒源选择比深睡多,一般不需要靠外部 GPIO 引脚唤醒。对于一些特殊的运行方式,例如 XIP,除了考虑芯片 Die 本身的低功耗外,外部悬挂的 Flash 的功耗也应考虑在内。进入休眠前,需要先退出 XIP 运行,进入 SRAM 运行,并且向 Flash 发命令使其进入低功耗模式。在唤醒时,先进入 SRAM 运行程序,然后

唤醒 Flash，继而进入 XIP 状态运行。要做到这一点，就需要用到第 5.6 节程序分散加载的知识，将中断程序编译到 SRAM 中，并将中断向量表也放在 SRAM 中。

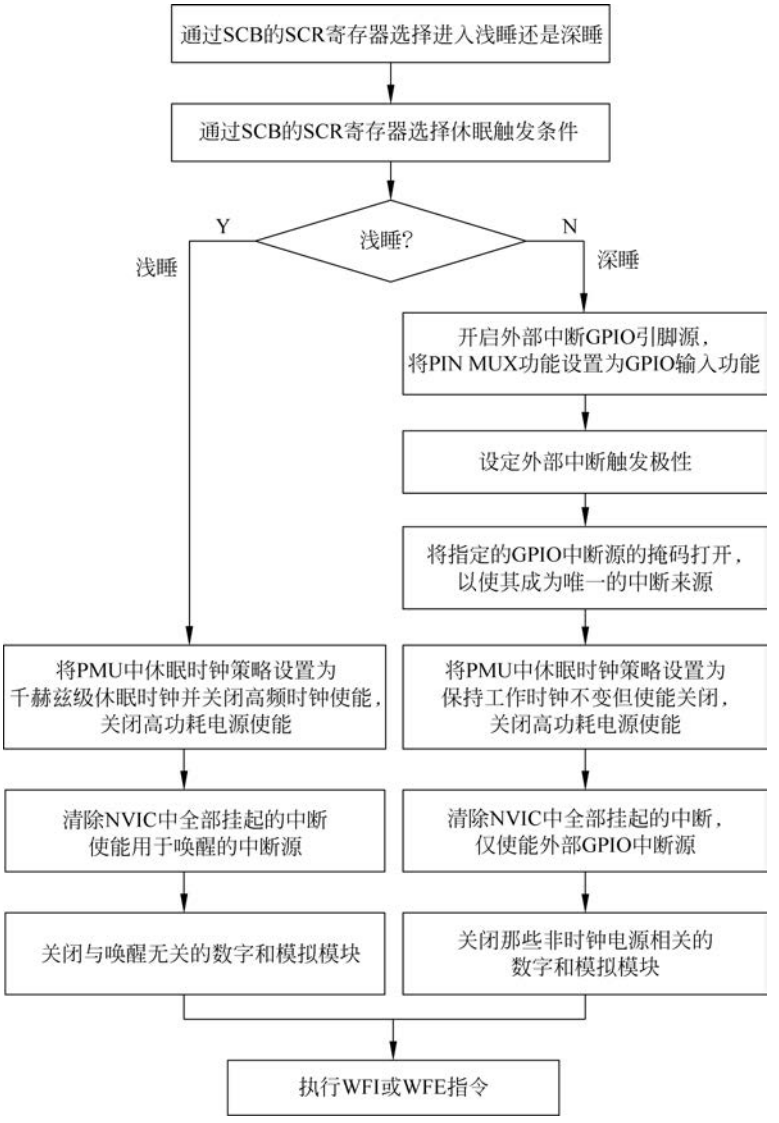


图 5-39 进入休眠的软件流程

5.22 SysTick 的集成和使用

SysTick 不论有无操作系统都会经常使用，它可以作为一个有着固定时钟源的 Timer 使用，还提供了时钟校准功能。SysTick 处于 CM0 核中，但它的配置却在 SCS 中，其基地址见表 5-3。SysTick 寄存器的具体定义见表 5-11。

表 5-11 SysTick 寄存器的具体定义

AHB 地址	寄存器名	段 名	比特位置	说 明
0xE000E010	CTRL	ENABLE	0	SysTick 计数使能
		TICKINT	1	SysTick 中断使能
		CLKSOURCE	2	时钟源选择： 0：外部时钟 1：SCLK 时钟
		COUNTFLAG	16	内部中断标志。 通过发起写该寄存器可以清除，也称为写清
0xE000E014	LOAD	LOAD	23:0	重载计数值
0xE000E018	VAL	VAL	23:0	当前计数值
0xE000E01C	CALIB	TENMS	23:0	时钟校准字(只读)
		SKEW	30	驱动时钟频率是否为 100Hz 的整数倍（只读）： 0：是 1：否
		NOREF	31	时钟源选择的状态(只读)： 0：外部时钟 1：SCLK 时钟

要将 SysTick 集成到系统内，连接方式具有一定的特殊性。在 CORTEXM0INTEGRATION 接口上有一个 26 位 STCALIB 信号，需要将该信号放到某个 AHB 或 APB 设备中进行配置，无法放到 SysTick 本身进行配置，因为它在 CPU 内部，一般在集成时不会改动 CPU 内部的代码，因此，配置 SysTick 的功能需要两部分寄存器，其中一个是表 5-11 所示的 SCS 寄存器，另一个是由集成设计者自行设计的设备地址，用于配置 STCALIB 信号。STCALIB 的定义见表 5-12。可见，STCALIB 与地址 0xE000E01C(CALIB 字段)是基本相同的，只不过 CALIB 都是只读的，而 STCALIB 可以配置。实际上，CALIB 读到的值正是 STCALIB 所配置的。STCALIB 中的 NOREF 与 CTRL 中的 CLKSOURCE 具有相同的定义，因此，时钟选择信号是被这两个配置双重驱动的。在 CPU 内部，对此多驱问题的处理方式是其中任何一个配置为 1，最终的结果才为 1，也就是说，只有当两者都配置成 0，才能使用外部时钟计数。

表 5-12 STCALIB 的具体定义

AHB 地址	寄存器名	段名	比特位置	说 明
设计者自定义	STCALIB	TENMS	23:0	时钟校准字
		SKEW	24	驱动时钟频率是否为 100Hz 的整数倍： 0：是 1：否
		NOREF	25	时钟源选择的状态： 0：外部时钟 1：SCLK 时钟

除 STCALIB 以外，CPU 还有一个与 SysTick 相关的时钟使能接口 STCLKEN，它便是外部时钟的使能信号。外来时钟与 SCLK 的关系是异步的，不能直接驱动计数器，而是通过 STCLKEN 间接完成。STCLKEN 由外部时钟产生，但需要同步到 SCLK 时钟域，其产生逻辑如下例所示，其中，外部时钟名为 ext_clk，对应复位为 ext_rstn。toggle 信号由 ext_clk 驱动翻转产生，将其同步到 SCLK 时钟域，并探测变化沿，探测结果为 STCLKEN，将该信号输入 CPU 中。

```
always @(posedge ext_clk or negedge ext_rstn)
begin
    if (!ext_rstn)
        toggle <= 1'b0;
    else
        toggle <= ~toggle;
end

always @(posedge SCLK or negedge HRESETn)
begin
    if (!HRESETn)
        toggle_sync <= 3'd0;
    else
        toggle_sync <= {toggle_sync[1:0],toggle};
end

assign STCLKEN = toggle_sync[1] ^ toggle_sync[0];
```

外部时钟源一般会选择休眠时钟等相对较慢的时钟，特别是 SysTick 常用于秒、毫秒等公制时间单位的计时，在产品中也常用 32.768kHz 这样的频率，因为将它进行 15 次二分频就可以得到准确的一秒。在软件中，需要考虑如何设定 STCALIB 中的校准字 TENMS 和它的 SKEW。校准字是以 10ms 为衡量标准的，其对应的频率是 100Hz。当外部时钟频率是 100Hz 的整数倍时，将 SKEW 设为 0，并将 TENMS 设为该整数倍减 1。例如，不论是外部时钟还是 SCLK，最终的驱动时钟为 15MHz，它是 100kHz 的 150 000 倍，因此，SKEW 填 0，TENMS 填 149 999。

SysTick 是 24 位递减计数的，当计到 0 时会发起中断，并在 COUNTFLAG 位上置 1，因此，软件使用中断法或轮询法都可以检测到计数为 0，清除中断只需对 COUNTFLAG 发起写操作。

5.23 非 SoC 架构的芯片

市场上除了主流的 SoC 芯片，还有一些非 SoC 类型的芯片，即不包含 CPU，靠纯逻辑运行的芯片。SoC 芯片的优势在于可编程，灵活性、稳健性很强，与之相比，非 SoC 芯片在流片后，其功能和用法就定型了，仅有寄存器控制，其更改余地不大，要实现类似的流程控制，需要在设计时就加入有限状态机。那么，为什么还会存在非 SoC 芯片的需求呢？原因

是它在某些简单应用场景下比 SoC 芯片面积小、成本低。SoC 芯片必备的一些组件,如 CPU、RAM、ROM,其面积占整个芯片的面积比例较大,而在非 SoC 芯片中可以省去。

失去 CPU 后,非 SoC 芯片就失去了自动执行预定动作的能力,只能执行两种动作,一种是上电时就预定好的动作,另一种是人机接口实时输入的控制命令。

上电预定动作当然不是指流片前在 RTL 中写的寄存器的默认值。芯片在流片前规定的上电默认值只是一个参考值,而这些值中可能会有一部分不符合芯片的应用情况和工艺状况。特别是对于模拟电路,电路原理图、版图、工艺类型、Foundry 等因素都可能影响它的参数,使流片前预计的参数与流片后实际测量的参数产生较大的区别。为了在缺失 CPU 的情况下使流片后各寄存器的配置也能修改,必须建立一套 FSM 机制,从外部存储器中载入修正数据配入寄存器。外部存储器不需要特别大的容量,一方面是出于节省成本的考虑,另一方面,非 SoC 芯片往往是逻辑量较少的小型芯片,并没有很多寄存器。相对于大容量的 Flash,小容量的 EEPROM 更适合此类场景。EEPROM 一般使用 I^2C 接口进行读写,因此,芯片内部需要集成一个 I^2C 主设备(Master),以便上电后自动读取 EEPROM 的值,并按顺序填写到芯片的各寄存器中。为了避免芯片在外部不挂载 EEPROM 的情况下,或 EEPROM 尚未被烧写为所需要的值的情况下就被芯片读取,引起配置错误,继而导致芯片功能不正常的问题,芯片一般会对读取的数据进行合法性检查。检查项可以有两个,一个是写在 EEPROM 首地址的 ID,另一个是写在 EEPROM 末地址的 CheckSum 或 CRC 校验字。当 EEPROM 已配置了一套合理的控制字后,也会顺便在其首地址配置一个固定的 ID,即识别字。芯片读取 EEPROM 时,首先读取 ID,若其为规定值,则继续读取其他地址,否则就停止读取,芯片使用默认值工作。ID 的作用主要是体现 EEPROM 的存在,并且已经被人为烧写过。写在 EEPROM 末尾的 CheckSum 或 CRC 校验字的作用是证明 EEPROM 数据的完整性和传输的可靠性。当芯片将 EEPROM 中的字节逐一读入后,其内部会自动计算一个 CheckSum 或 CRC 值,而 EEPROM 末尾包含烧写者计算的检查值。将芯片内部计算值与 EEPROM 末尾的检查值进行对照,若一致,则说明 EEPROM 烧写正确且传输正确,数据可以使用,否则说明 EEPROM 数据损坏或在传输过程中损坏,不能使用,芯片仍然以默认值上电。由于需要最后检查 CheckSum 或 CRC 值才能证明数据的可用性,芯片在读取 EEPROM 后不能直接将数据写入功能寄存器,而是要先存入临时寄存器,待最终的检查通过后再载入,因此,配置用的寄存器的面积增大了一倍,若出于成本因素,不愿增加这部分面积,则可以在检查不通过后停止芯片继续运行,而不是以默认状态运行,因为此时默认状态已经被篡改。CheckSum 是一种简单的检查数据完整性的方式,将全部寄存器字节或字段按比特异或,就可以得到一字节或字段长度的 CheckSum 码,CRC 校验的检错能力更强,但电路更为复杂,面积开销更大。不论是 CheckSum 还是 CRC,在位宽限定的情况下,都有一定的漏检概率,即可能会有数据本身是错的,但未检查出来的情况,一般选择 32 位检查,可将该问题出现的概率降低到可忽略的程度。

人机接口实时操作是指在芯片外面挂接一个主控芯片,如 MCU,将该 MCU 与芯片通过 SPI、I^2C 等接口相连。MCU 可按顺序和步骤改变非 SoC 芯片内部的配置值。有时,这

个 MCU 可被计算机代替，用人工方式按一定时间顺序对芯片中的同一个寄存器进行不同的配置。在这种应用中，MCU 或人工可看作 CPU，而非 SoC 芯片可视为挂接在 CPU 上的一个设备，两者组成了一个 SoC 芯片系统，总线的功能由 SPI 或 I^2C 接口来承担。此方案的缺点是成本高，因为非 SoC 芯片设计的前提就是为了降成本，而它的应用如果需要在外面单独配备一个 MCU，则成本并未降低，反而可能提高了，因此，此方案并非此类芯片的推荐方案，但可以用于芯片调试，避免每个配置步骤都必须重新上电走存储器读取流程。在正常应用中，仍使用上电载入 EEPROM 数据的方式。既然 EEPROM 常使用 I^2C 接口，那么人机交互方式的接口一般也选同样的 I^2C 接口，这样，两种方式可以通过同一个 I^2C 接口进行操作，不会浪费引脚，但外部 MCU 或计算机作为主控，非 SoC 芯片就只能作 I^2C 从设备(Slave)，因而在芯片中除了要设计 I^2C 主设备外，还要再设计一个 I^2C 从设备。芯片运行过程可分为两部分，第一部分为上电状态，由 I^2C 主设备接管 I^2C 接口，从 EEPROM 中载入配置数据，第二部分为工作状态，由 I^2C 从设备接管 I^2C 接口，接受芯片外部 I^2C 主设备的读写。芯片内外 I^2C 接口和设备的连接方式如图 5-40 所示，包括芯片内部的主设备和从设备，芯片外部 EEPROM 和 MCU，它们都通过同一套接口相连，因而 I^2C 接口也被称为 I^2C 总线。值得注意的是，在 I^2C 通信中，只有从设备有地址，在图中，芯片内部的从设备和芯片外部的 EEPROM 都应有自己的地址，当用户通过 I^2C 烧写 EEPROM 时，不需要将 EEPROM 从电路板上取下，而是直接从 MCU 或计算机控制的 I^2C 主设备对 EEPROM 的地址发起写操作，芯片中的从设备也会侦听到该通信，但由于地址不同于烧写地址而会被自动屏蔽。

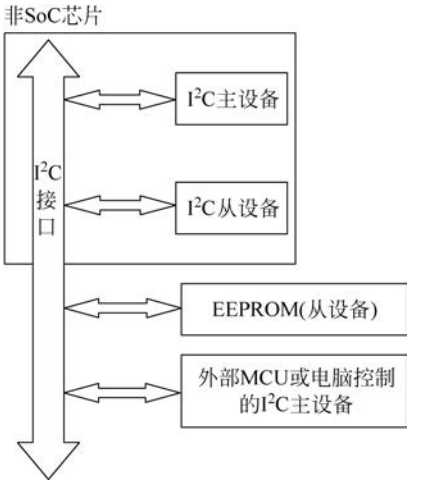


图 5-40 非 SoC 芯片及其外部环境的 I^2C 连接

非 SoC 芯片仅在功能简单、寄存器少的情况下才会比 SoC 芯片更省面积。它虽然节省了 CPU、RAM 和 ROM 的面积，但它需要更多的寄存器。例如，要设计一颗芯片来控制室内温度，对于 SoC 芯片，可以只用一个寄存器，通过编程在不同的温度场景下配置不同的值即可，而如果做成非 SoC 芯片，通过有限状态机来达到相同效果，首先必须确定状态的个数，然后确定每种状态下对应的配置字，再为这些配置字单独配备一个寄存器，在运行时，状态机在不同状态中将不同的配置字输入实际起作用的寄存器中，状态机相当于一个 MUX，如图 5-41 所示。如此，一个寄存器就变成了多个寄存器，从而增加了面积。

设计者也可以将 SoC 芯片中的每个设备视为一个单独的非 SoC 芯片，因为任何芯片都不是完全靠 CPU 控制才能运行的，即使最简单的 Timer 也有自动计数的功能，芯片设计的魅力主要体现在 CPU 配置完后的自动运行上，仅将设备视为组合逻辑的集合，而将 CPU 视为全部时序的体现，这种想法是错误的，因此，这个世界上可能非 SoC 芯片会越来越少，

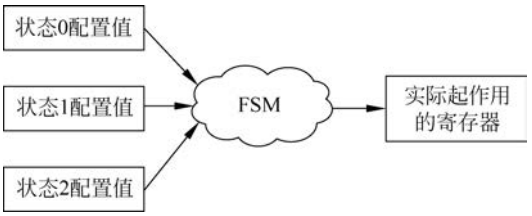


图 5-41　有限状态机运行所需要的配置

但其体现的设计思想,包括状态机、自动复位、自动加载数据等方法和思路,将会以越来越复杂的形式体现在 SoC 芯片的设计中。