第 3 章　软件无线电实验平台 Sora

软件无线电是认知无线电的重要基础，也是实现认知无线电最为关键的一项使能技术(enabling technology)。因此，若要设计认知无线电系统，就必须先学习并掌握软件无线电技术。

本章以微软(亚洲)研究院开发的 Sora 软件无线电平台为例，对软件无线电技术进行简要的回顾，剖析软件无线电的工作原理和系统构成。第 4 章将会具体介绍 Sora 软件无线电平台上使用的模块化编程方法。感兴趣的读者可以结合本章的参考文献对软件无线电技术进行更加深入的学习。微软软件无线电平台(Sora)完全建筑于现代的多核个人计算机(PC)基础上，利用高级软件程序设计语言实现高效的软件无线电信号处理。Sora 将硬件软件无线电平台的高性能和高保真度与通用处理器软件无线电平台的可编程性和灵活性结合起来。同时使用硬件和软件技术，以迎接 PC 上高速软件无线电的挑战。Sora 平台基于通用个人计算机架构，性能优异而且使用方便。目前，Sora 已经得到了国际无线通信领域专家学者的广泛关注，被北美和欧洲等地区的世界著名高校选为无线通信研究平台，目前已有超过 20 个国家的 200 个实验室正在使用 Sora 进行无线通信的研究开发。微软研究院软件无线电项目 Sora 于 2015 年 7 月正式开源[①]。如需下载 Sora 开源代码，可以访问 https://github.com/Microsoft/Sora。

微软(亚洲)研究院开发的高性能软件无线电平台 Sora 是一种基于 PC 的完全可编程的新型无线电平台。Sora 包括软件无线电协议栈和相应的硬件组件。Sora 的硬件组件包括无线控制板(radio control board，RCB)以及一系列可替换射频前端模块(radio frequency front-end module，RF)。Sora 硬件组件完成基本的模数(数模)转换，无线射频前端控制，并提供高速的接口连接 PC 主机，从而在主机内存和射频前端之间交换信号采样。Sora 基于高速 PCIe 总线，可提供最高 16Gbps 的接口速率，用以支持宽频带(例如 50MHz～6GHz)和多天线系统(例如多输入多输出系统，MIMO)。Sora 软件无线电协议栈则利用现代 PC 上的多核 CPU 提供高效的无线信号处理和实时无线网络协议的支持。

---

① 参见 http://www.msra.cn/zh-cn/research/release/Sora.aspx。

3.1 Sora简介

Sora软件无线电平台基于通用个人计算机架构,通过软件来实现尽可能多的通信功能,而硬件部分功能将尽可能地简单、通用,从而提高了系统的灵活性和可编程能力(programmability)[1]。在Sora系统中,模数/数模(AD/DA)转换器构成软件与硬件之间的天然界面。正如图3-1所示,通信系统的模拟部分由硬件完成,而所有AD转换之后(DA转换之前)的处理功能均由软件在通用处理器上实现。这包括整个物理层(基带)处理、媒介接入控制(MAC)、网络层及其他高层协议。

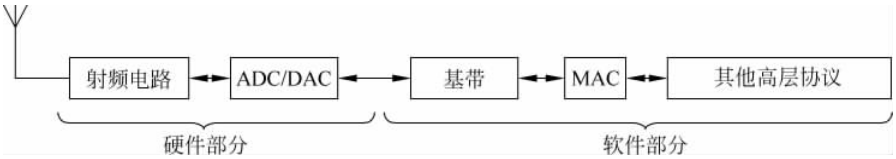

图3-1 Sora软件无线电平台可分为硬件部分(无线射频模块和模数/模数转换)和软件部分

软件无线电平台Sora的硬件部分主要包括一块无线控制板(RCB),以及一系列可替换的射频前端模块。射频前端模块包含射频(RF)电路,例如射频滤波器和在基带和射频之间进行信号转换的上变频器和下变频器,以及将波形在模拟域和数字域之间进行转换的AD/DA转换器。

RCB为射频前端模块和个人计算机之间提供了一个通用、一致的接口用于传输数字采样。这个接口基于高速、低延迟的PCIe总线标准[4],因此可以支持高速的数字信号采样。RCB支持一系列不同的射频前端模块。每一个射频前端模块可以采用不同的射频滤波器和不同的AD/DA采样率。因此,Sora可以选用适当的射频前端模块在不同频段上、以不同的带宽工作。图3-2展示了Sora RCB的实物图。Sora RCB板上包含一个存储器,用以缓存预先计算好的波形数据。对于常用的固定信号,RCB可以直接从存储器中读取波形,而无须再从PC的主存中获得,这一功能可以显著地降低时延。

Sora的系统构成如图3-3所示,主要包括多核商用PC、无线控制板(RCB)、射频板(RF front-end board)等。此外,还有模数转换器(A/D convertor)与数模转换器(D/A convertor)将射频板与无线控制板(RCB)连接起来。

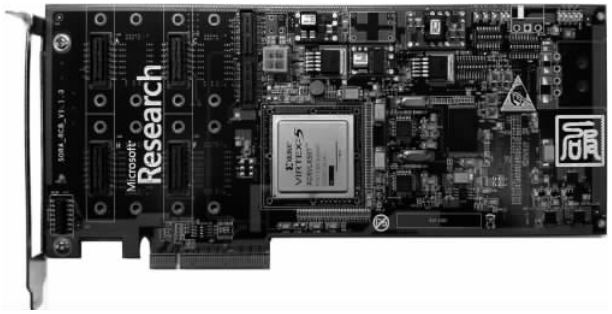

图 3-2　Sora 无线控制板(RCB),可以通过 PCIe 接口与个人计算机(PC)相连

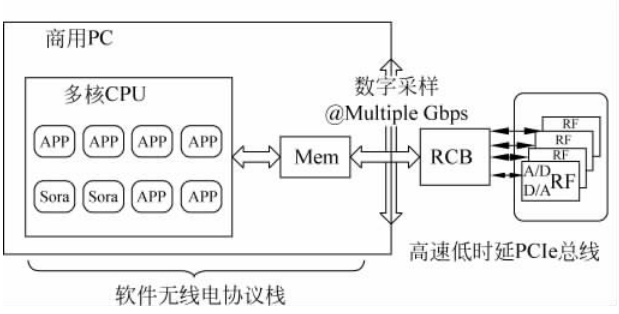

图 3-3　软件无线电平台 Sora 的系统构成

3.2　Sora 软件架构与软件优化技术

Sora 的软件部分提供了实现软件无线电系统所必需的系统抽象,并提供相应的库函数和服务,以简化无线基带和链路层(MAC)协议的实现。图 3-4 给出 Sora 软件无线电的整体架构。Sora 给开发者预留了两套编程所需的接口:内核态的核心应用程序接口(Core API)以及用户态下的用户扩展接口(User-Mode eXtension API,UMX)。

Sora Core API 由 RCB 驱动程序实现。为了使用 Sora 的 Core API,开发者需要将自己的信号处理程序写成一个 Windows 核心态驱动程序。编写一个内核驱动程序需要相当的经验,而且调试起来相当的复杂。一个有问题的驱动程序很容易导致整个操作系统崩溃,因此 Core API 对于一般的程序开发人员来说,既复杂又不太安全。

为了降低编程的难度,Sora 同时提供了一个用户态应用程序接口,即用户模式扩展接口(UMX API)。程序员可以通过 UMX 接口在用户态就可以直接访问

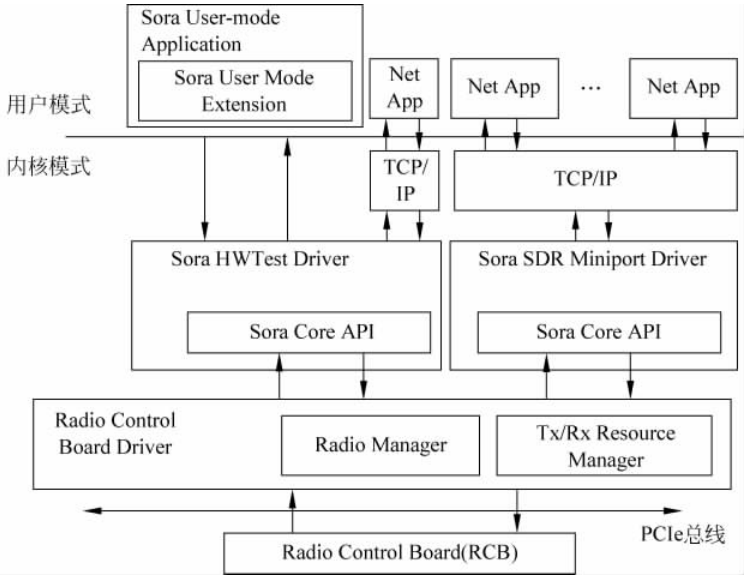

图 3-4 Sora 软件架构

各种软件无线电资源,例如接收数字采样,发送信号波形,以及配置射频模块的各项参数等。UMX API 是通过另一个内核态驱动程序 HWTest 来实现的。优化过的 UMX API 性能已经完全接近 Sora Core API。例如,利用 UMX API 所有的数字采样数据都是直接传输到用户态内存中的(无须从内核空间复制到用户空间),因此与 Core API 具有相同的性能。UMX API 的另一个优点在于它为软件无线电应用程序提供了保护机制,因此不仅可以防止由于程序错误导致的系统崩溃,同时还支持多个应用程序同时共享无线硬件资源。

3.2.1 抽象无线电对象

Sora 平台可以支持多种射频前端模块,不同的射频前端可能采用不同的芯片组,因此工作方式也会有所不同。为了屏蔽掉这些复杂的操作,Sora 将无线电射频硬件抽象成一个对象,称为抽象无线电对象(abstract radio object,ARO)。ARO 为应用软件提供了一个统一的接口用以控制和读写不同的射频硬件。如图 3-5 所示,ARO 具备一个发送信道、一个接收信道和一组统一的控制寄存器。ARO 与无线射频硬件之间有一一对应的关系。一个 Sora 应用程序可以操作一个或者多个 ARO。当应用程序执行写控制寄存器操作的时候,这一命令通过 ARO 传送到 RCB 上,再由 RCB 转发到射频前端模块,最后由射频前端模块上的固件(firmware)将这一命令翻译成为射频芯片特定的操作指令,并写入射频芯片中。

例如，如果应用程序想要改变无线电设备的接收增益，它可以给 ARO 的接收增益控制寄存器（例如，地址 0x0A）写入一个新参数值。对于使用 2.4GHz/5GHz 射频前端的用户，这个写入操作最终被翻译成为相应的指令写入射频控制器（如 MAXIM 2895）。因此，对于任意的一个射频模块硬件，只要编写相应的固件解释这些抽象命令，就可以和已有的 Sora 软件一起工作了。

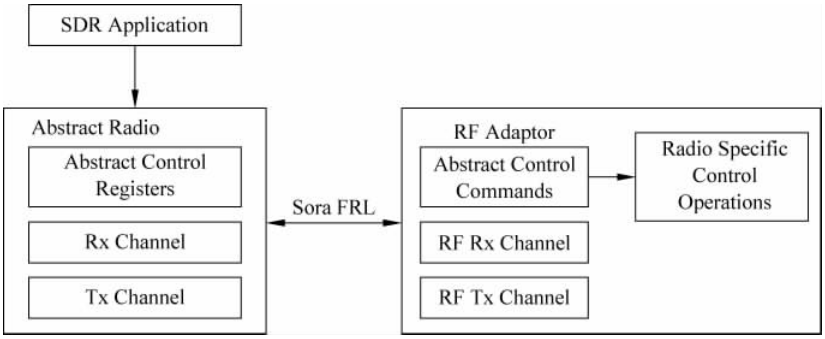

图 3-5　Sora 中的抽象无线电

在抽象无线电对象中，接收信道表示为一个数字信号样本流。接收到的信号在射频前端被数字化，然后传送给 RCB，最后由 RCB 通过直接内存访问（DMA）写入计算机内存中。Sora 分配了一个循环缓冲器来保存接收到的数字样本，如图 3-6 所示。RCB 上面的 DMA 引擎保存一个写指针。通过它，RCB 连续不断地将接收到的数字样本写入内存。与此同时，ARO 接收信道维护一个读指针，用以检索样本。如果读指针与写指针发生重叠，这意味着所有接收到的样本都已经被软件处理完成，读操作将会被阻塞。反之，如果软件的处理能力跟不上 DMA 的操作速度，写指针将会从后面追赶读指针，当它们相遇时，缓存中所有的样本数据就会直接丢失掉。

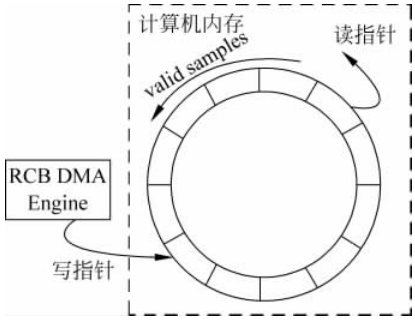

图 3-6　Sora 抽象无线电的接收信道（接收到的数字样本存储在循环缓冲器之中）

Sora ARO 采用两“步”方式发送一个数字信号。首先,应用程序将调制好的数字信号样本转存(transfer)到 RCB 的存储器上;然后,应用程序指示 ARO 将一个转存好的信号从射频前端硬件上发送出去。这两步可以独立、异步进行。例如,应用程序可以事先转存一个信号序列,然后在不同的时刻反复发送这一信号。在 ARO 进行转存时,每一个信号都被分配一个发送标号(Tx ID),用以唯一标识这个转存信号。在第二步时,ARO 将使用这一发送标识来指定将发送的信号,如图 3-7 所示。

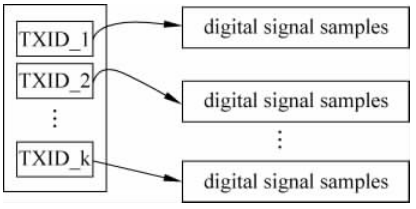

图 3-7 Sora 抽象无线电的发送信道

3.2.2 UMX 编程

Sora UMX 为用户模式的程序访问 Sora 抽象无线电的资源提供了一个非常便利的编程接口。Sora UMX API 进行过高度优化,与 Sora 核心态接口相比,它所产生的性能开销可以忽略不计。同时,Sora UMX 还提供了更好的保护机制,并支持不同应用程序之间共享同一个无线电硬件资源。相比于核心态编程,用户态程序更易编写和调试。因此,用户应该尽可能地通过 UMX 编程接口来使用 Sora 平台。

UMX 应用程序应遵循如下的步骤(例 3.1)。

(1) 初始化 UMX 库;

(2) 初始化一个 ARO;

(3) 为抽象无线电对象分配接收信道;

(4) 从接收信道分配一个接收流;

(5) 配置合适的无线接收机参数;

(6) 为发射信道分配一个转存数据缓存区;

(7) 在转存缓存区里填写调制好的数字信号样本;

(8) 将数字样本块转存到 RCB 存储器上;

(9) 指示 RCB 发送已经转存好信号样本块;

(10) 在程序终止前去初始化(de-initialize)UMX 库。

【例 3.1】　一个 UMX 应用程序的框架示例。

(1) 这个程序的主函数。它首先初始化 UMX 库。然后,它调用主处理函数 umx_main()。

当处理函数返回后,它清除 UMX 库。

```
//
int __cdecl main(int argc,const char *argv[])
{
    // Initialize UMX API
    if (!SoraUInitUserExtension("\\\\.\\HWTest")) {
          printf ( "Error: fail to find the hwtest driver!\n" );
          return -1;
    }

    // Start the main procedure
    umx_main ();

    SoraUCleanUserExtension();
   return 0;
}
```

(2) 在 umx_main()函数中,首先创立一个 ARO。这由函数 SoraURadioStart 来实现。ARO 用一个从 0 开始的数字来标识一个射频硬件(请查阅 Sora 硬件手册来确定这个标识)。一旦 ARO 建立之后,所有的无线电资源都可以通过 ARO 来访问。接着,使用函数 SoraURadioMapRxSampleBuf 将 ARO 的接收通道映射到用户态。SoraURadioMapRxSampleBuf 会返回接收信道的 DMA 缓存地址(RxBuffer)以及环形缓存区的大小(RxBufferSize)。应用程序无须也不应该直接访问该缓存区,而是通过一个 ARO 的接收流来访问接收到的样本数据。接收流是一个 SORA_RADIO_RX_STREAM 对象。通过函数 SoraURadioAllocRxStream 获得之后,再调用 ConfigureRadio 来设置合适的无线电射频参数。在进入处理函数(process)来处理接收到的数字采样之前,分配一个转存数据缓存区(利用 SoraUAllocBuffer 函数)。这个缓存区将在 Sora 信号发送过程中使用,我们在后面的章节里详细讲解。

```
PVOID             RxBuffer = NULL;
ULONG             RxBufferSize = 0;

SORA_RADIO_RX_STREAM  RxStream;
```

```
PVOID                TxSampleBuffer = NULL;
ULONG                TxSampleBufferSize = _M(4);

void umx_main () {
    HRESULT    hr;

     // Create an ARO
     Hr = SoraURadioStart (TARGET_RADIO);
     if (FAILED(hr)) {
          printf ( "Fail to start an ARO ( % d)\n",TARGET_RADIO );
          return;
     }

    // Map Rx channel buffer
    hr = SoraURadioMapRxSampleBuf(TARGET_RADIO,&RxBuffer,&RxBufferSize);
    if (FAILED(hr)) {
        printf ( "Fail to map rx buffer!\n" );
        return;
    }

    // Allocate a virtual Rx stream from the Rx channel
     hr = SoraURadioAllocRxStream ( &RxStream, TARGET _ RADIO, ( PUCHAR ) RxBuffer,
RxBufferSize);
    if (FAILED(hr)) {
        printf ( "Fail to allocate a RX stream!\n" );
        goto error_exit;
    }


    // configure radio parameters properly
    ConfigureRadio ();

    // Allocate a sample buffer for transmission
    TxSampleBuffer = SoraUAllocBuffer(TxSampleBufferSize);
    if (!TxSampleBuffer) {
        printf ( "Fail to allocate Tx buffer!\n" );
        goto error_exit;
    }


    // call processing function
    process ();
```

```
error_exit:

    // Release virtual Rx stream
    SoraURadioReleaseRxStream(&RxStream,TARGET_RADIO);

    // Release Tx sample buffer
    if (TxSampleBuffer) {
        SoraUReleaseBuffer(TxSampleBuffer);
        TxSampleBuffer = NULL;
    }

    // Unmap Rx channel
    if (RxBuffer) {
        hr = SoraURadioUnmapRxSampleBuf(TARGET_RADIO,RxBuffer);
    }

}
```

(3) 无线电射频参数主要包括:①载波频率(中心频率);②频偏矫正(用以微调载波频点);③发送增益(功率);④接收机第一级放大器增益;⑤接收机第二级放大器增益。需要说明的是,现有的 Sora 射频硬件没有自动增益控制(automatic gain control,AGC)功能,因此功率控制需要应用程序对接收增益进行适当的设置以保证接收信号保持在 A/D 转换器的动态范围之内。

```
void ConfigureRadio ()
{
    // Set radio carrier frequency.
    // Here,we set the carrier frequency to 5.2GHz (channel 40)
    SoraURadioSetCentralFreq (TARGET_RADIO,5200 * 1000);

    // Set frequency offset if needed
    // SoraURadioSetFreqOffset ( TARGET_RADIO,0 );

    // Set Tx gain
    SoraURadioSetTxGain ( TARGET_RADIO,0x1500 );     // 21 dB


    // Configure receiving gain
    SoraURadioSetRxPA    (TARGET_RADIO,  0x2000 ); // 接收机第一级放大器增益 16dB
    SoraURadioSetRxGain (TARGET_RADIO,0x1000 );     // 接收机第二级放大器增益 16dB
}
```

(4) 应用程序调用 SoraRadioReadRxStream 从接收流中提取信号样本块 SignalBlock,SignalBlock 是一个存储 28 个 16 位 I/Q 采样点的数组。SoraRadioReadRxStream 同时返回一个布尔标识(fReachEnd)。如果 fReachEnd 被置位,则表明当前的样本块是接收流中最后一个样本块,软件的处理速度已经超过了硬件的采样速率。因此,程序从接收处理函数中退出,让出一些 CPU 时间处理别的任务(如果有的话)。

```
void receive ()
{
SignalBlock sigblk;

    while (1) {
      hr = SoraRadioReadRxStream(
                    &SampleStream,
                    & fReachEnd,// indicate if end of stream reached (you must
                                // wait for hardware)sigblk);

        process_signals ( sigblk );

        if ( fReachEnd ) break;
    }
}
```

(5) Sora 发送过程包含信号转存和信号发送两步。首先,应用程序需要调用 SoraUAllocBuffer 分配一个转存缓存区(已在 umx_main 中完成),然后将调制好的数字信号样本填写到这个转存缓存区中。当这些都完成之后,程序可以执行信号转存操作——调用 SoraURadioTransferEx——将数字样本转存到 RCB 存储器中。在第二步中,应用程序调用 SoraURadioTx 将一个转存好的信号从射频硬件上发送出去。一个转存好的信号可以被发送任意多次。当这个信号不再被使用时,可以通过调用 SoraURadioFree 将它从 RCB 存储器中移除。

```
void send ( PVOID TxSampleBuf,ULONG TxSampleBufSize )
{
HRESULT hr;
   ULONG TxID = 0;

    // Transfer the Tx digital samples to the Tx channel
  hr = SoraURadioTransferEx(TARGET_RADIO,TxSampleBuf,TxSampleBufSize,&TxID);

  if (SUCCEEDED(hr))
```

```
    {
        // Instruct the radio hardware to send the signal out
        hr = SoraURadioTx(TARGET_RADIO,TxID);

     // Release the Tx channel resource
        hr = SoraURadioTxFree (TARGET_RADIO,TxID);
  }
 }
```

3.2.3　数据包反射

Sora 支持一种称为数据包反射(reflection)的机制将 UMX 应用程序无缝地整合到 Windows 的网络协议栈中。图 3-8 给出了数据包反射架构。HwTest 驱动程序将一个虚拟以太网接口开放给操作系统。因此,网络程序可以通过通常的网络套接口(socket)从这个虚拟网络接口收发报文。HwTest 驱动程序会捕获所有的发送报文,并临时存放在一个发送队列(sendQueue)中。UMX 应用程序可以通过 Sora 数据包反射 API 来获取这些临时存储的报文,因此这些数据包又被反射到用户态。这样,UMX 程序可以把这些数据包进行调制,产生相应的数字信号,然后从射频硬件上发送出去。在接收时,UMX 程序将数据包从接收信号里解调出来,然后再通过 Sora 数据包反射 API 将解调的数据报文插入到 HwTest 驱动的接收队列(recvQueue)之中。最后,这些数据包通过虚拟网络接口传递到上层的 TCP/IP 层,直至网络应用程序的套接口上。

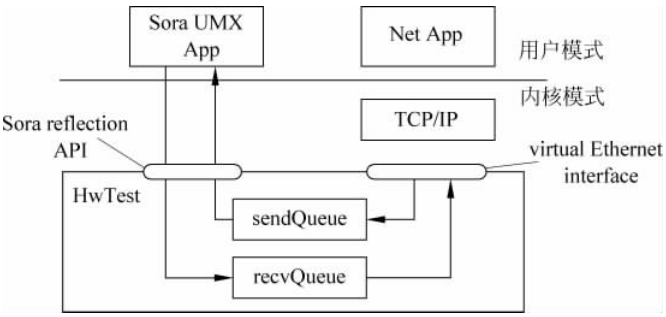

图 3-8　Sora UMX 反射架构

下面简要介绍一下如何使用 Sora 数据包反射 API:

(1) 使用 SoraUEnableGetTxPacket 初始化 Sora 数据包反射 API。

(2) 通过 SoraUGetTxPacket 从 sendQueue 中获取数据包。

(3) 使用此数据包之后,UMX 应用程序需要调用 SoraUCompleteTxPacket 将与数据包相关联的资源退还给 HwTest 驱动。

(4) 使用 SoraUIndicateRxPacket 将(通常是从接收信号中解码出来的)数据包插入到 HwTest 驱动的接收队列 recvQueue 之中。

3.2.4 Sora 独占线程

软件无线电所面临的一个关键挑战就是如何满足许多无线通信协议对实时性的要求。例如,IEEE 802.11 协议需要在正确接收一个数据包之后的限定时间内(例如 10～16μs)发出一个确认信号 ACK。否则,发送端将认为发送失败,从而重发数据报文。而基于 TDMA 的系统,可能会要求在一个非常高的精度(例如微秒量级的精度)来控制数据包传输。这些精确的时序要求对于一般的通用软件系统是相当困难的。

通常的软件系统(例如 Windows 操作系统)的处理时间往往是不确定的。造成这种不确定的原因是因为处理器(CPU)是被多种不同任务所共享的,而传统的非实时操作系统不支持绝对优先级调度,因此非实时的任务可能打断实时任务的执行。采用专用的实时操作系统(例如 RTLinux[11])可以解决这一问题,但是这些实时操作系统一般功能简单,编程和维护也相对复杂。

Sora 采用了另一种思路,即充分利用现代 CPU 的多核功能来解决这个问题。Sora 采用了一种称为内核专用(core dedication)的简单而有效的方法来保障实时性。根据需要,Sora 从多核系统中分配足够数量的内核给高实时性任务,从而实现对它们实时要求的保障。

内核专用是通过独占线程(exclusive threading)来实现的。独占线程具有最高的优先级,即 Windows 操作系统的实时优先级,所以它不会被其他较低优先级的线程所抢占。同时,独占线程固定到一个特定的 CPU 内核上,因此它只能使用这一指定的 CPU 内核。独占线程可以被硬件中断所干扰,严格地说,这些硬件中断也会造成一些不确定性,但硬件中断的处理一般较快。同时,在 Windows 系统中,硬件中断也可以指定关联 CPU 内核进行处理。因此,这些硬件中断可以关联到其他的内核上,从而使得它们不会干扰独占线程的工作。

Sora 线程的体系架构如图 3-9 所示。每个独占线程被绑定给一个专用内核。每一个独占线程包含有一个实时任务的程序指针列表和一个调度程序。该调度程序以轮询调度(round-robin)方式循环调用每一个实时任务。实时任务的调度是以协作方式完成的,这就意味着每一个实时任务程序需要在完成工作之后立即返回,这样就可以将控制交回调度程序,从而允许其他的实时任务有机会执行。在实际使用中,一个实时任务占用专用内核的时间最多不能超过几毫秒。

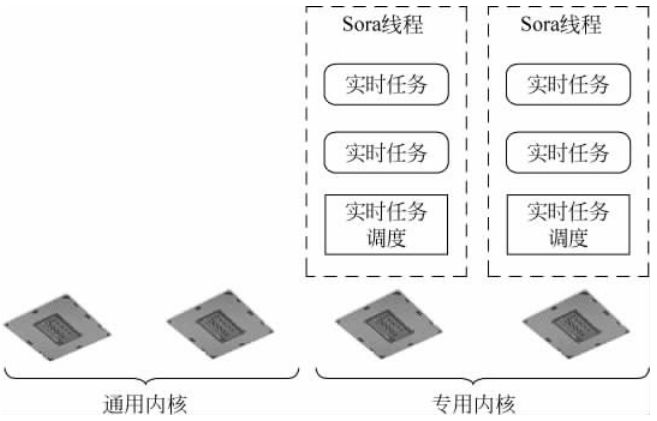

图 3-9　Sora 独占线程示意图

下面简要介绍使用 Sora 独占线程的流程：

(1) 调用 SoraUThreadAlloc 分配一个独占线程对象。

(2) 调用 SoraUThreadStart 给独占线程分配一个内核并开始执行。

SoraUThreadStart 将一个实时任务的指针当作一个参数，一旦独占线程开始工作，调度程序将不断地调用这个实时任务。一个实时任务过程具有下面的格式。

```
BOOLEAN rt_routine ( LPVOID lpParam );
```

如果程序返回 FALSE 意味着该实时任务已经完成，然后调度程序将该程序从激活任务列表中删除，此后不再调用它。否则，程序返回 TRUE，会被再次调用。

(3) 程序可以调用 SoraUThreadStop 终止一个独占线程。注意：SoraUThreadStop 不可从实时任务程序内部调用。实时任务程序应该返回 FALSE 值来终止。

【例 3.2】 利用 Sora 线程 API 实现实时任务。

(1) 分配一个 Sora 线程对象，该对象通过句柄数据类型来引用。在这个线程对象成功创建后，可以调用 SoraUThreadStart 启动该线程。SoraUThreadStart 获取一个指向实时任务程序的指针。

```
HANDLE thrd_handle = NULL;

bool start_thread () {
```

```
    // Allocate a Sora thread object
    thrd_handle = SoraUThreadAlloc();
    if (thrd_handle == NULL )
        return false;

    // Start the thread
    if ( !SoraUThreadStart ( thrd_handle,thread_proc,NULL )) {
        stop_thread ();
        return false;
    }

    return true;

}
```

(2) 在实时任务程序中处理数据。当对实时性要求比较高的操作完成后,程序需要立即把控制权返回给进程调度程序。当所有处理工作完成后,程序会返回FALSE。

```
BOOLEAN thread_proc ( PVOID pParam ) {
    // Do processing
    bool bRet = process_data ();

    if ( bRunning ) {
        return true;
    } else {
        // exit
        return false;
    }
}
```

(3) 退出前释放线程对象。

```
void stop_thread () {
    if ( thrd_handle ) {
        SoraUThreadStop(thrd_handle);
        SoraUThreadFree(thrd_handle);

       thrd_handle = NULL;
    }
}
```

3.2.5　优化

本节介绍多核 CPU 上实现高效数字信号处理程序的软件优化技术。我们将主要讨论以下三个常用的软件优化技术：

(1) 使用查找表(LUT)来存储计算(结果)。

(2) 使用单指令多数据(SIMD)指令实现进行数据的并行处理。

(3) 利用多核并行处理。

1. 查找表

现代的 CPU 都具有大容量、高速缓存(cache)。因此可以事先将算法的执行结果保存在查询表(lookup table)中。这样在进行在线处理时，计算结果就可以直接从查询表中获取，而无须重新计算，极大地降低了计算量。

【例 3.3】　通过查找表实现卷积编码器。

卷积码是一种广泛使用的信道编码方案，它输出的(经过编码的)比特是基于一小段原始输入比特。图 3-10 所示的是 IEEE 802.11a 协议的卷积编码器结构。该编码器具有一个 6 位的移位寄存器，每个输入位有选择地与在寄存器中的几个比特进行异或运算，产生两个输出比特。算法的直接实现将需要执行八个异或操作，即每个输出比特都是输入位与移位寄存器中四个比特位依次进行异或的结果。因此，对每一个比特进行编码都会花费几十个 CPU 周期！相反，如果采用查询表 LUT 方式实现，预先计算好的结果——每个输入字节和各种寄存器状态组合所对应编码输出——都存储在查询表中。这样对于任意一个输入字节，仅通过两个查找操作就可得到编码结果(八个输入比特)，所需时间还不到 20 个 CPU 周期(平均一个比特仅需 2～3 个周期)。存储这样一个查询表总共需要 32KB 的高速缓存，这对于现代的 CPU 来说是微不足道的。与直接实现的方案对比，查询表的方式将运算速度提高了 8 倍以上。

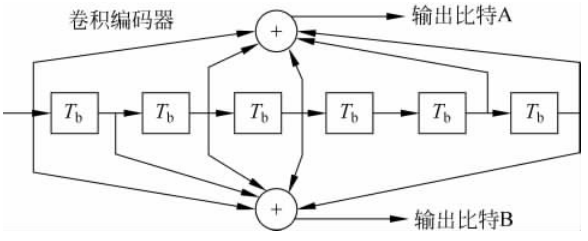

图 3-10　IEEE 802.11a 卷积编码器结构

【例3.4】 利用LUT实现软解映射器。

软解映射器算法用于解调信号,它需要计算输入符号中所含每个比特的置信度。通常,软解映射器的计算复杂度与调制密度(调制星座点的数目)成正比。具体来说,它要遍历星座图中所有调制点,计算出接收符号到所有代表"1"的调制点的最小欧式距离,与它到所有代表"0"的调制点的最小欧式距离之间的比值。同样地,可以利用查询表的方法来实现软解映射器。预先计算好所有可能的输入符号(由符号的I/Q值决定)的置信度,并保存在查找表中。软解映射器的查找表一般也无须很大,例如在802.11a/g协议下的54Mbps调制(64-QAM),这个查找表仅仅有1.5KB。

2. 使用SIMD(单指多数)指令

几乎所有现代CPU都包含有单指多数(SIMD)指令,如SSE[10]、3DNow! 和AltiVec。SIMD指令允许CPU的执行单元对多个数据同时进行运算,从而提高处理速度。在数字信号处理中,有许多算法可以通过SIMD指令加速。为了方便使用SIMD指令,Sora SDK提供了一个称为Vector1的C++模板库。Vector1封装了矢量数据类型,并为它们提供了一整套便捷的算术运算。这些矢量算术运算都通过SIMD指令来实现,因此可以对其每一个元素数据同时运算。Vector1中的矢量数据可以使用多种元素类型(例如浮点数,整数以及复数整数等)。它也提供了一整套便捷的算术运算支持所有矢量数据类型。所有这些操作都已经使用SSE指令得以实现和优化。

【例3.5】 通过Vector1对一组整数求和。

在本例中,使用整数向量(vi)操作对整数数组求和。程序首先将整型指针转换成一个整数矢量(vi)指针,然后使用矢量加法对所有矢量元素求和,最后使用水平相加(hadd)将所有元素添加到矢量中。

```
#include <vector128.h>
// Sum an array of integers using Vector operations
// We assume the input pointer is 16-byte aligned and the size is a multiple of vi::
size.

int vec_sum ( int * data,int nsize ) {
    vi * pv = (vi*) data;
    vi sum;
    set_zero (sum);

    for (uint i=0; i<nsize / vi::size; i++) {
```

```
        // sum vi::size = 4 of integers using single instruction
        sum = sum + pv[i];
    }

    // Sum the row of four elements in the vector.
    sum = hadd (sum);

    // the final result is in the first element of the vector
    return sum[0];
}
```

【例 3.6】　使用 Vector1 实现相关算法。

相关是数字信号处理中应用最普遍的算法，相关运算定义如下：$y_i = \sum_{j=0}^{L} x_{i-j} \times c_j$，$c_j$ 是相关的模板，x_i 是输入的数字样本，x_i 和 c_j 都是复数。由于每个输出是独立的，因此有可能并行地计算多个相关值。对于每一个输入样值，可以计算部分项的相关值，换句话说，可以为所有的相关值保存 L 项部分和。一旦得到新的输入，会更新所有已存储的部分和。当完整的相关值计算完成后，再输出结果。为了方便并行计算，需要多次复制相关参数，并将它们存储到一个矢量数组。图 3-11 所示的是相关计算的并行实现。程序每次操作四个输入样本（构成一个复整数矢量类型），并计算四个部分和。每个相关值的部分和也通过一个复整数矢量类型保存。

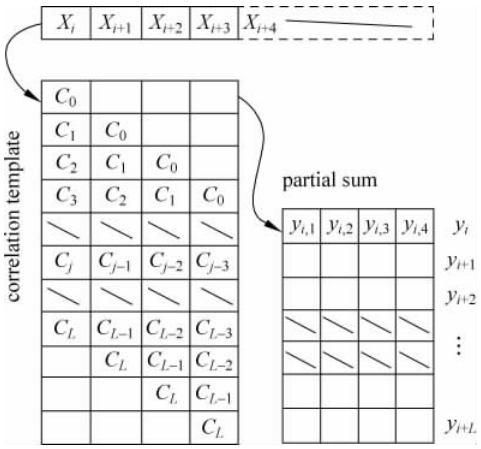

图 3-11　相关运算的并行执行

```
// The input samples are arranged in complex vectors. Each element is a COMPLEX16.
// The procedure assumes the partial_sum storage and output buffer are provided by
the caller
// and are properly initialized.
// vcs::size = 4
FINL void correlate ( vcs * input, int size, vcs * taps, int tap_len, vcs * partial_
sum, vcs * output )
{
    vcs  ans[vcs::size];
    vcs * ps = input;
    vcs * po = output;

    // compute through all samples
    for ( int i = 0; i < size; i++ ) {
        vcs * ppsum = partial_sum;
        vcs * ptap = taps;

        // compute the four output correlation value y_i, y_(i + vcs::size)
        for (int j = 0; j < vcs::size; j++ ) {
            ans[j] = mul ( * ps, * ptap) + * ppsum;
            ppsum ++; ptap ++;
        }

        * po = hadd4 (ans[0], ans[1], ans[2], ans[3] );
        po ++;

        vcs * pstore = partial_sum;

        // update all partial sums
        for (int j = vcs::size; j < tap_len; j++)
        {
            * pstore = ( * ppsum) + mul ( * ps, * ptap);
            pstore++; ppsum ++; ptap ++;
        }

        ps ++;
    }
}
```

3. 多核并行处理

当信号处理的计算要求超过单个 CPU 内核的计算能力时,就需要将工作分

发到多个内核上以增加计算速度。如图 3-12 所示，通常情况下有两种方法可以将处理工作分发到并行工作的多个 CPU 内核。图 3-12(a)所示的是任务级并行性。在这种情况下，多个相同的处理线程同时在不同的内核中运行。调度程序将给不同的数据块分配到不同的线程上来处理。如果各个数据块之间是彼此独立的，这种任务级并行非常有效，而且实现起来很简单(例如，对应于无线链路不同帧的数据块)。然而，这种方法的缺点是处理时延较长，所有的处理都需要在单个内核上完成，因此就形成了一个瓶颈。另一种并行方案则如图 3-12(b)所示，可以将一条处理流水线上不同的功能模块交给不同的内核来处理，就像一条管道(流水线)一样。前一个内核处理的输出结果作为输入交给第二个内核处理。这种管道并行计算方式具有更低的处理时延，但在实际系统中实现起来可能会比较复杂。如何选择最佳的并行方案依赖于具体应用的要求，例如对于数字信号处理的延迟容限等。

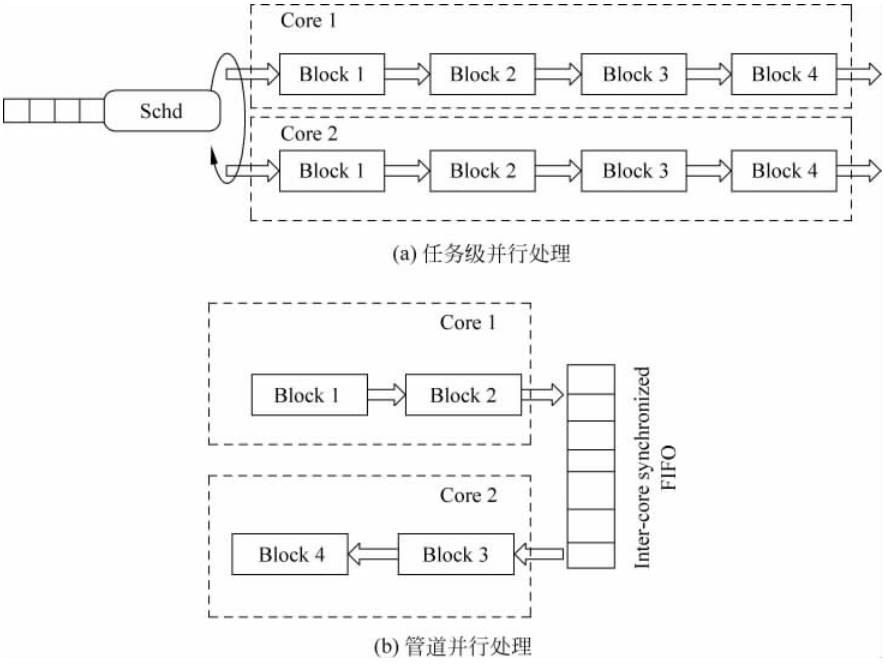

图 3-12　并行处理方法

为了实现管道并行，在不同内核上的处理函数需要通过共享存储器 FIFO 队列进行相互通信。这些共享 FIFO 队列必须保证同步。传统的同步的方法是通过一个计数器来实现的，称为基于计数器的 FIFO (counter-based(CB)FIFO，例 3.7)。但在多核系统中，这个计数器被两个处理器内核所共享。因此，每一次当一个内核

给变量写入一个新值时,都会导致另一个内核出现高速缓存缺失(cache miss),从而该内核必须从较慢的内存中(而不是从高速缓存)获取更新值。由于数据产生端和消费端都要修改这个变量,所以对于每个数据的读写都不可避免地产生二次高速缓存缺失。在数字信号处理中,信号采样的频率是非常高的,因此要非常频繁地进行同步(例如,每微秒进行一次),这样的高速缓存缺失产生的开销将会非常显著。为了避免这种开销,可以使用另外一种同步FIFO的方法。这时,不再使用唯一的共享同步变量,而是为FIFO中的每个数据槽(slot)增加一个标志头(header)。该标志头用来标记相应的数据槽是否为空。数据槽的大小总是设定为缓存线(cache line)的整数倍数。如例3.8所示,FIFO仍然以循环缓冲区的方式组织。数据消费程序(consumer)通过读指针检索数据,而数据产生者(producer)通过写指针输入数据。读指针总是在追逐着写指针,因此这样的同步FIFO也称为追逐指针FIFO(chase-pointer (CP) FIFO)。追逐指针FIFO(CPFIFO)可以在很大程度上减少上述的高速缓存缺失开销。如果数据的产生和处理的速度是相同的,这两个指针则会分开一个固定偏移量(例如两个高速缓存线)。这时,读写同步队列将不会出现高速缓存缺失,因为对于数据消费程序而言,在它读取当前数据槽时,存储器管理硬件会自动预先读取下一个数据槽的内容。因此,当数据消费程序需要这一数据槽时,该槽内的数据已经在高速缓存里了。而如果数据的产生和消费具有不同的处理速度,例如读数据的速度快于写数据的速度,那么数据消费程序最终需要等待数据产生者产生新的一个数据槽。在这种情况下,每次写入一个新的数据槽,数据消费程序也会产生一次高速缓存缺失。但数据产生者将不会受高速缓存缺失的影响,因为下一空闲数据槽已经被提前提取到它的本地缓存中。这时,由于数据消费程序不是流水线的瓶颈,因此即使产生了高速缓存缺失也不会对系统整体性能造成严重影响。

【例3.7】 CBFIFO同步的伪代码。

```
// producer:
void write_fifo ( DATA_TYPE data ) {
      while (cnt >= q_size);       // spin wait
      q[w_tail] = data;
      w_tail = (w_tail + 1) % q_size;
      InterlockedIncrement (cnt);  // increase cnt by 1
}

// consumer:
void read_fifo ( DATA_TYPE * pdata ) {
```

```
    while (cnt == 0);                // spin wait
    * pdata = q[r_head];
    r_head = (r_head + 1) % q_size;
    InterlockedDecrement(cnt);       // decrease cnt by 1
}
```

【例 3.8】 CPFIFO 同步的伪代码。

```
// producer:
void write_fifo ( DATA_TYPE data ) {
    while (q[w_tail].flag > 0);      // spin wait
    q[w_tail].data = data;
    q[w_tail].flag = 1;              // occupied
    w_tail = (w_tail + 1) % q_size;
}

// consumer:
void read_fifo ( DATA_TYPE * pdata ) {
    while (q[r_head].flag == 0);     // spin
    * data = q[r_head].data;
    q[r_head].flag = 0;              // release
    r_head = (r_head + 1) % q_size;
}
```

3.2.6 小结

在通用个人计算机(PC)上实现支持宽带软件无线电,不仅需要将高速率的数字信号样本传输到个人计算机中,还需要对这些数字信号进行高速的计算处理(例如调制、解调等)。因此要求整个系统能支持很高的吞吐速率(包括射频前端和物理层之间,以及物理层内部各个处理模块之间)。例如,Wi-Fi 协议中要求在 20MHz 带宽上能够支持 54Mbps 的物理层速率,而为满足此要求,高速的软件无线电平台至少要能够实时处理 1.2Gbps 的数字信号采样(假设单天线系统,20MHz 信道带宽,16 比特模数转换,4 倍采样速率即 80MSPS 采样速率)。此外,无线通信协议、物理层和链路层还需要很低的处理延迟。例如,802.11 的 MAC 层需要精确的时钟控制以及很低的 ACK 响应时延(16μs)。这些对于通常的 PC 和操作系统而言是难以实现的。

总的说来,Sora 软件无线电系统是从硬件和软件两个方面同时采取措施,并通过联合优化技术来应对上述挑战的。

1. 硬件上的措施

无线控制板(RCB),使用高速低延时的 PCI Express (PCIe)总线接口标准,PCIe 总线不仅吞吐量大,而且响应时间也优于 USB、GbE 等接口,非常适合软件无线电的应用场合。可以支持 16.7Gbps(×8 模式)吞吐量,处理的滞后时间达到为纳秒级,完全满足无线协议的吞吐量和时间需求。

2. 软件上的措施

(1) 查表法(LUTs):大大降低了物理层处理部分的计算复杂度。

(2) 单指令多数据流(SIMD)技术:充分利用 CPU 并行加速指令,特别适合 FFT 和 IFFT 这一类运算。

(3) 多核处理:多核 CPU 之间的流水线式操作。在不同内核之间通过 FIFO 同步,充分发挥各个内核的计算能力。

(4) 独占内核技术:为了确保 CPU 可以实时响应,Sora 采用了独占线程技术,可以让某个或者某几个内核专用于软件无线电的任务,不被其他系统调用影响。实现这种技术不需要修改操作系统的内核。

Sora 的硬件构成包括以下四部分:无线控制板(RCB)、射频适配板(RAB)、射频板(RFB)和商用 PC。Sora 的软件部分为开发者在通用操作系统实现规范的 PHY 和 MAC 协议提供了必要的系统服务和编程支持。Sora 核心库(Core Library)用于获得硬件资源,主要包括了以下三个子库:

- 线程库(ethread lib),为支持实时控制的专用线程提供 API。
- 有限状态机库(FSM lib),为状态机编程提供有限状态机框架。
- 数字信号处理库(DSP lib)提供一些普遍用到的数字信号处理优化算法。

Sora 的基本工作过程简述如下。

Sora 的接收过程:

如图 3-13 所示,在接收数据时,射频前端从天线处获得模拟信号,经过下变频,A/D 转换将模拟信号数字化成离散样本,最后传递给 RCB 板。由 RCB 写入 DMA,再从 DMA 中读取信号采样数据,这样 Sora 就可以对接收的数据进行处理了。

Sora 的发射过程:

如图 3-14 所示,在发送数据时,首先是将待发送的 I、Q 信号样本下载到 RCB 内存中,然后调用指令指示 RCB 发送存入的波形样本。射频前端模块产生数字样本同步流,并通过 D/A 转化、上变频形成符合要求的模拟射频信号,并最终通过天线发射出去。

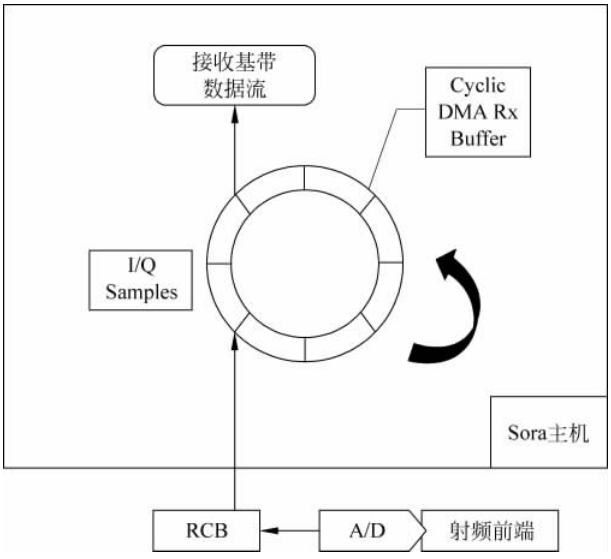

图 3-13　Sora 的接收过程

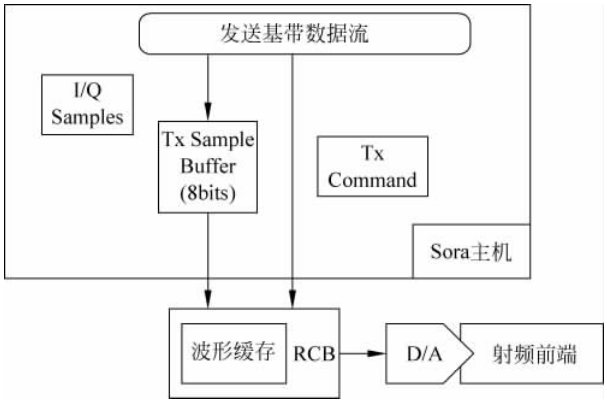

图 3-14　Sora 的发射过程

思考题

1. 请画出软件无线电平台 Sora 的系统架构，并讲述各软、硬件模块的主要功能。

2. 试比较以下不同软件无线电的实现方式的优缺点：

（1）基于 FPGA；

(2) 基于专用DSP处理器;

(3) 基于通用处理器(CPU)。

3. 试比较如下接口技术的优缺点:①PCIe;②USB;③Ethernet。

为什么Sora平台选取了PCIe总线作为无线控制板(RCB)和个人计算机(PC)内存之间的接口?

4. 简述Sora系统是如何在通用处理平台上实现宽带无线通信系统,主要采用了哪些关键技术?简述其接收与发射过程。

5. 查阅加拿大通信研究中心(Communication Research Centre)所研发的基于Wi-Fi的认知无线电通信实验系统(CORAL)[7],并与Sora系统进行全面比较。

6. 请利用SIMD实现8×8复整数矩阵的乘法,并与标量实现进行比较,处理速度有多大的提高?为什么?

7. 请在Sora平台上编写程序实现:

(1) 在2425MHz上发射一正弦波;

(2) 接收以2422MHz为中心的20MHz频带。做傅里叶频谱分析,检测出发送的正弦波。在屏幕上显示频谱分析的结果。请问检测到的正弦波频率是2425MHz吗?为什么?

8. 查阅基于GNU Radio与universal software radio peripheral (USRP)的软件无线电系统[2,12],并与Sora进行对比研究,总结归纳其异同。

参考文献

[1] Tan K, et al.. Sora: high-performance software radio using general-purpose multi-core processors[J]. Communications of the ACM. ACM, 2011.

[2] GNU Radio[OL]. http://www.gnu.org/software/gnuradio/.

[3] Wireless open access research platform (WARP)[OL]. http://warp.rice.edu/trac.

[4] PCI Express Base 2.0 Specification[S]. PCI-SIG, 2007.

[5] Cummings M, Haruyama S. FPGA in the software radio[J]. IEEE Communications Magazine, 1999.

[6] Lin Y, et al.. SODA: A low-power architecture for software radio[C]. Proc. 33rd International Symposium on Computer Architecture, 2006.

[7] CORAL[OL]. http://www.crc-coral.com/.

[8] Tennenhouse D L, Bose V G. Spectrumware a software-oriented approach to wireless signal processing[C]. MobiCom, 1995.

[9] Sora软件无线电平台微软官方网站[OL]. http://research.microsoft.com/en-us/projects/sora/.

[10] Intel® 64 and IA-32 Architectures Software Developer Manuals[OL]. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.
[11] RTLinux: A Real-Time Linux[OL]. http://www.oschina.net/p/rt-linux.
[12] 维基百科. USRP[OL]. http://en.wikipedia.org/wiki/Universal_Software_Radio_Peripheral.