1. 绪论
2. 软件仿真器架构
3. 模块间能量信息交互
4. 能量管理模块
5. 外设行为建模
6. 测试与仿真
7. 结论
8. 附录：软件使用

第一章：绪论

1.1 物联网（IoT）发展情况

物联网的是近几天提出的一个新兴概念，其定义如名称所说，是将世界上各种各样的物品通过一定方式接入网络。这些物品可以包含我们日常个人生活中涉及的物品，比如说起搏器、运动鞋、体温计等等，也可以包含社会生活中涉及到的物品，比如说自动贩卖机、售货机、物流车辆等等，物联网甚至可以将一些自然界的事物通过某种方式接入网络，比如说江河中的水流。物联网可以使这些物品分享自身数据，互相协助，以便更好完成某些目标[survey]。

物联网的诞生为人们提供了新的可能性。比如说在医疗健康领域，各式各样的医疗传感器使得人们能够在自己的家中获得完整准确的身体情况信息，网络上的有关医疗健康的服务商可以根据这些身体情况对可能的问题做出预警，对于病人来说，他们能够提早发现自身的病情并尽早就医，对于健康的人，他们能够根据自身的预警信息对疾病做出预防，维持在健康状态。在物流领域，物联网使得追踪任意一辆货运车辆、任意一个包裹成为可能，物流公司能够根据这些信息获知每一个物流节点是否顺畅、哪些方向的请求更加密集等信息，这使得物流公司能够进行合理的统筹规划，提高运送效率。在自然中设置的节点也有很大的意义，比如说，山体的内部压力数据可以通过传感器联网，异常的数据可能意味着即将出现山体滑坡等灾害，通过物联网的手段，灾害预防组织能够更早获得这些信息，进行预防并减少灾害。

物联网还给人们提供新的挑战，这些挑战包含网络地址不足、安全和隐私保护、标准化问题等[survey]，但其中最为重要的是能量采集问题，这也是本文最为关注的问题。物联网中的节点有体积小、数量多、分布广泛的特点，然而常规电池并不能支持这些设备长时间工作，如果使用电池为这些设备供电，定期维护的成本将会变得十分巨大，因此很多物联网中的节点摒弃了传统电池，采用了在环境中收集所需能量的方式。这些节点收集能量的方式多种多样，有的收集太阳能，有的能够收集动能[Energy harvesting vibration sources for microsystems applications]，更多的则是收集环境中的电磁波的能量[platform]。

由于环境能量稳定度不高，随时间变化较大，这些收集环境能量的设备在工作中将频繁面临掉电，因此这些节点中应该使用有别于基于传统供能系统的能量处理技术。

1.2 非易失处理器

1.2.1 非易失处理器简介

如上文所述，收集环境中能量的系统将频繁面临断电，传统处理器在频繁开关机时是无法推进任务的，这是因为在频繁掉电时，处理器寄存器与内存中储存的数据将会完全丢失，导致处理器丢失当前状态，这种传统的处理器是不适合工作在物联网中收集环境能量的节点的。为了能够在频繁断电时还能够进行连续计算，一种新的处理器结构，非易失处理器（Non-Volatile Processors, NVPs）被提出[makaisheng]，这些处理器与传统处理器相比有如下区别：

1. 这些处理器有能量采集系统

首先，NVP是从环境中采集能量的，因此NVP中存在能量采集器，这些能量采集器是随着采集能量类型变化的，一般来说，采集装置的最外侧是将外界能量转换成能够使用的电能的器件，接下来一般接入一些整流、负载转换使得电流、电压参数能够被处理器所使用。此外，为了保证足够的时间来使得在断电时系统能够保存当前状态，NVP中往往有能量收集装置（一般是电容）来收集多余的采集到的能量，当系统面临断电时，处理器将会使用这些临时储存的能量来进行备份操作。

1. 这些处理器有状态机来控制系统状态

和传统处理器不同，NVP需要自行自动控制当前处理器的状态。一个最简单的例子是，当电量充足的时候，NVP处于工作状态，当电量不足时，NVP先进入备份状态，再进入休眠状态，当电量足够开机时，NVP进入恢复状态，接下来进入正常工作状态。当然，这个例子是理想化的，最简单的状态机模型，实际的系统中可能有更加复杂的状态转换模型。

1. 这些处理器有备份-恢复装置与策略

当发生断电时，NVP需要将当前状态备份到非易失储存器中。Flash和磁碟由于速度慢、功耗大等原因不适合作为这种低功耗、快速断电的系统的非易失存储装置。近年来有一些新兴器件，如RRAM[]，nvFF[]，能够以接近SRAM的延时存放数据，且数据不会随断电而丢失。NVP中往往有这样的非易失储存阵列，当发生断电时，系统将易失的寄存器和内存中的全部或部分数据储存在这些非易失储存器中，保证系统状态不丢失。

图【】中介绍了一个典型的非易失处理器的结构。非易失处理器有着广泛的研究空间，无论是备份方式、系统状态机设计还是能量管理模块都有很多细节值得研究，下文将会分别介绍非易失处理器的这些研究方向。

1.2.2 非易失处理器备份策略研究

尽管非易失处理器是最近几年提出的概念，但可中断计算一直是一个被研究的话题。在没有非易失处理器的时代，人们需要在传统CPU的基础上进行一些软件-硬件上的开发，以便系统能够在断电时不丢失全部工作状态，一个典型的手段是设定记录点（checkpoint）。在记录点中，CPU将会把需要备份的数据从内存和寄存器搬运到硬盘中。由于传统的处理器并不能够像非易失处理器一样能够保证在断电时有足够的时间和能量备份全部需要备份的数据，设定记录点的方式是值得研究的。[momentus]论文提出了一种设定记录点的方式，这种方式是纯软件的，[momentus]修改了LLVM编译器，使得普通程序在被编译时会被插入一些函数来决定是否进行备份，这些函数在运行时会根据现有的能量情况估算短时间内是否会发生断电，如果估计的结果为会发生断电，这些函数会触发记录点备份，这样一来，断电重启时系统将会从记录点而不是程序的最开始被启动。

[momentus]中提出的算法的不足是，运行这种算法需要特殊的编译器，需要在软件端编译时修改程序，这给软件开发带来了复杂度，而且，使用事先插入的记录点触发位置并不能保证程序的运行效率，如果触发点位置不佳可能会导致系统在重启后有较大回退，增加程序运行时间。基于这一点，文章[hibernus]提出了另外一种备份策略，这种备份策略是软件-硬件协同的，这种备份策略为系统引入了能量不足中断，当系统能量低于一定阈值时，会触发中断，这时中断处理函数将会进行备份，将系统状态保存到硬盘中，可以看出这种方式已经极为接近当今非易失处理器的备份方式了。

虽然当前非易失处理器的备份方式都是类似的，但仍然有一些细节值得被研究。在非易失处理器中，备份过程的能量消耗和时间消耗仍然是系统性能的瓶颈，如果备份过程的能量消耗过大，则系统需要有更高的掉电能量阈值，这会导致系统更经常发生开关机，降低在特定能量环境下的运行效率。非易失储存器，无论是nvSRAM，RRAM，还是nvFF备份都需要一定能量和一定延时，一种简单直观的想法就是在发生掉电时不备份全部内存，而是只备份最近被使用过的内存块（Lest Recently Used, LRU），降低备份所需能量和时间，然而，测试表明这种方式与全备份相比带来的性能提升不大[NVPsim]。在一些研究工作中描述了一种新的备份时内存块选择策略，一种死亡块预测算法（SBDP）被提出，测试表明这种算法能够有效提高备份的能量需求和时间需求[backup\_scheme]。

1.2.3 非易失处理器能量管理模块研究

1.2.4 非易失处理器与外设的交互

尽管非易失处理器能够保证在掉电发生时保存当前状态，但一旦涉及到非易失处理器与外设协同工作，情况将变得复杂，这是因为绝大部分外设是易失的，也就是说一旦在外设工作时发生断电，则需要命令外设重复工作。这种复杂性使得非易失处理器与外设的交互过程需要被研究，这方面的研究存在两个方向，第一个就是研究外设硬件的设计，使得外设硬件成为非易失的，当发生断电、掉电时不需要处理器的干涉就能自行恢复并且继续完成任务，另一个方向是研究处理器调用外设的调度方式，使得处理器能够在特定的外界能量情况下尽可能不让外设任务被断电所打断。

[hehe2015dac]是一个典型的处理NVP本身与外设协同的工作，在文章中作者将上电/掉电带来的额外时间计算在内，提出了一种调度外设工作的策略，当同一时间有很多外设任务需要完成时，合适的调度外设的策略使得系统能够在某一特定的外界能量环境下用最短的时间完成所有外设的工作。

[long-term-DAC]同样针对外设的调度问题基于一个太阳能收集节点进行了研究，然而在此文章中调度的优化目标并不是在最短时间内完成所有外设工作，而是使得更大比例的外设工作能够在断电之前完成工作，也就是优化长期的DMR（deadline miss rate）。

1.3 非易失处理器验证方式

以上所有有关非易失处理器的研究方向均有很大的研究空间，当有关非易失处理器的新技术被提出时，应该有准确、合理的手段对这些新技术的性能进行验证。比方说，在涉及非易失处理器备份策略的研究时，研究者可能会关心如下问题：当系统的备份方式从内存、寄存器全备份变为部分备份内存、寄存器时，备份所需能量会发生什么变化，备份时间占用程序运行整体时间发生怎样的变化，部分备份造成的数据缺失会产生程序运行多大的时间增加？事实上部分备份减少了备份所需要的时间，但是能量恢复后的数据缺失可能会要求处理器耗费更多时间来恢复这些数据。这是一个极为复杂的问题，涉及处理器运行的很多方面，外界能量收集情况、处理器运行程序种类可能均会对最终的结果产生影响，这些影响很难通过一个特定的公式来定量描述。因此，仿真是验证新技术性能的主要手段。

由于非易失处理器诞生较晚，非易失处理器的仿真是一个崭新的研究方向，非易失处理器的仿真往往比较原始，即设计好电路再在电路的基础上进行验证。此外，近年来诞生了少许非易失处理器的仿真软件，比如说基于gem5实现的非易失处理器仿真软件NVPsim[1]。下文会分别介绍电路级仿真以及NVPsim仿真器。

1.3.1 电路级仿真

顾名思义，电路级仿真就是用一些硬件描述语言设计好电路，并使用一些特定的数字电路仿真软件对设计出的数字电路进行仿真。这是一种通用的办法，适用于一切数字电路，非易失处理器也不例外。目前多数有关于非易失处理器的研究基于这种仿真方法，研究者已有一个非易失处理器原型（多数为实验室开发的示例平台），实现其提出功能的硬件电路设计，集成到已有的实验平台上，并使用硬件仿真工具（如Modelsim[2]）进行仿真。

1.3.2 NVPsim

Gem5提供了一个高自由度、配置方便的处理器体系架构仿真平台[3]，在此基础上，Gu et. al.拓展了Gem5的功能，为Gem5提供了仿真处理器、内存在外界能量变化时行为的功能[1]。NVPsim的作者使用NVPsim对使用不同种类非易失储存器、不同备份策略的非易失处理器进行了仿真，并得出了非易失储存器和备份策略对处理器效率影响的一些结论。

NVPsim基于Gem5的TimingSimpleCPU模型，加入了电压检测模块、系统状态机、备份/恢复模块，并对Gem5事件队列的管理模块进行了一些修改。这些对Gem5的增补使得NVPsim能够仿真非易失处理器在特定外界能量条件下的性能。

NVPsim在系统运行过程中，通过Power Trace文件获取外部能量收集情况，并随时追踪系统的各个模块耗电情况，当能量不足时，系统状态机告知事件队列暂停当前的工作，并通过备份/恢复模块对需要备份的数据进行备份。当关机系统收集够充足能量时，系统状态机告知事件队列继续运行，并通过备份/恢复模块对内存、寄存器中得到备份的数据进行恢复。NVPsim在仿真过程中主要的关注点在于备份、恢复所需的能量和时间，在备份、恢复模块对能量和所需时间进行集中计算，而系统在上电、掉电时的行为主要是由事件队列管理模块完成的。NVPsim相比电路级仿真能够大大降低仿真的难度。

1.4 当前验证方式的不足

无论是电路级仿真还是使用NVPsim仿真软件进行仿真均存在不足之处，本部分会分别分析二者的不足。

1.4.1 电路级仿真的缺陷

虽然电路级仿真有着仿真结果准确、与实际系统误差几乎可以忽略不计等优点，但是这种仿真方式过于通用，并没有对非易失处理器进行专门的设计，因此，这种仿真方式给使用者造成了巨大的不便，极大增加了仿真的工作量。以下为电路级仿真的缺陷：

**高门槛**

首先，电路级仿真需要有完整的非易失处理器电路设计才能够完成。研究者为了研究非易失处理器中某一些特定部分的行为，往往需要拥有整个非易失处理器的电路设计。当前非易失处理器问题的研究者通常需要研究小组前期对电路设计有所准备。为了探索NVP的特定部分，研究过程为：

非易失处理器原始平台设计（prototype）-原始平台仿真、流片-修改原始平台需要研究的部分-测试、仿真

这使得世界上只有少数拥有NVP原始平台（prototype）的研究小组有能力对非易失处理器技术进行研究，大大增大了非易失处理器研究的入门门槛。

我们可以将其与传统处理器体系架构的研究进行对比，传统处理器有一些通用、公开的设计，并有着大量方便的仿真工具，在对体系架构进行研究时，研究者实际上并不需要拥有完整的处理器电路设计。

**配置不方便**

在使用电路级仿真时，任何微小的修改往往都会对电路设计产生巨大的影响，研究者为了探索新技术时往往需要深入硬件描述代码并进行大量修改。仍以掉电时内存备份策略研究为例，为了将系统在掉电时的备份方式从全备份改为部分备份，通常来说硬件描述语言中整个备份模块都需要被重写，备份方式的改变还会对电路的其他部分造成影响，比如说备份/恢复需要的线路宽度可能会发生变化。

在探索非易失系统中的新技术时，研究者并不希望微小的修改牵涉出复杂、混乱的系统变化，而是希望仅仅修改一些参数就能够对新技术的性能进行仿真，显然电路级仿真并不能做到这一点。

**仿真时间过长**

在使用电路级仿真时，仿真软件需要模拟出系统的每一个引脚在每一时刻的信号值，这会使仿真的时间开销变得巨大。而非易失处理器是非常复杂的电路，有着大规模集成芯片的等级，此外，为了能够获得准确的仿真结果，仿真使用的测试程序（benchmark）所需周期数往往不低。

非易失处理器仿真的这些特点会进一步增加仿真所需的时间，使得每一次电路级仿真的时间变得难以忍受。而电路级仿真的优点——对电路细节描述的全面与准确往往并不是被关注的对象，研究者关注的可能仅仅是运行测试程序（benchmark）所需要的时间和能量效率。

**软硬件接口复杂**

使用电路级仿真时，被仿真的对象是硬件本身，缺少一个简单、易用的软硬件接口，因此，使电路级仿真运行研究者关心的测试程序（benchmark）往往会消耗额外的时间。当前已有的非易失处理器使用的并不是大规模商业化的架构（如x86、ARM），比如说THU1010n非易失处理器芯片使用的是8051架构[4]，这些架构可能存在一些编译器，但并没有针对非易失平台特殊编写的编译器，因此研究者在电路仿真中运行特定程序往往需要手动或者用一些工具将程序翻译成机器语言并以硬件的形式写入硬件描述语言中，这显然是十分复杂的。

在理想情况下，使用高级语言（如C）编写的程序就能够运行在仿真平台上，这会大大降低仿真非易失处理器行为的难度，然而电路级仿真并不能够支持这一点。

1.4.2 NVPsim的缺陷

NVPsim继承了gem5仿真平台的一些特点，因此避免了上述电路级仿真带来的不足。比如说NVPsim像gem5一样支持运行ARM、x86等流行的程序，这使得用户能够使用高级语言编写测试程序（benchmark）并使其运行在仿真平台上。NVPsim也忽略了电路中过于细节的部分，只描述系统模块的行为，这大大增加了NVPsim的仿真速度。NVPsim还使得用户能够通过修改少量仿真参数直接改变目标非易失处理器的行为，降低了仿真的配置难度。

然而，NVPsim仍然存在着一些不足，使得其并不能成为一个广泛使用的非易失处理器仿真平台。NVPsim存在着如下不足：

**对硬件的能量行为描述不自由**

NVPsim对系统的行为采用了“集中式”管理办法，并没有对每一个模块分别编写在系统状态发生变化（上电、掉电）时的行为。NVPsim在系统状态发生改变时改变模块行为方式的唯一渠道是事件队列管理模块，当系统发生掉电时，事件队列管理模块读取gem5的事件队列，将所有事件暂停，这样一来系统就停止了运行，而当系统发生上电时事件队列管理模块再将所有事件重新调度使得系统恢复运行。

这种对掉电、上电的建模方式虽然有效，但是缺少对硬件在系统状态改变时行为描述的自由度。NVPsim将系统中所有的掉电、上电时的行为描述都集中在了一起，在上/掉电时只对系统整体的事件队列进行操作，而不对每个模块分别进行操作，这样一来所有的模块在系统状态变化时都表现出了相同特性：暂停运行。

事实上，每一个模块在系统状态改变时的行为是不同的，易失的模块应该丢弃当前内部的所有状态，非易失模块应该保持内部状态不变，而另一些模块比如说电压检测模块、系统状态机模块应该继续工作，不受掉电、上电的影响，显然NVPsim对这样行为的描述是不足的。

**对正确性验证的先天不足**

上文描述了NVPsim的工作流程和对掉电、上电的建模方式，这种建模方式仅仅暂停了硬件触发的所有事件，实际上默认了所有模块都是“非易失”的，也就是说，掉电时所有模块的内部状态都没有发生改变，仅仅是暂停运行而已。NVPsim并没有提供对硬件易失性的描述，也没有为编写者提供描述硬件易失性的接口。

实际上非易失系统中还是存在某些易失模块的，比如说CPU内的寄存器或者传统的内存模块再掉电时应该丢失储存的全部数据。这一点不足使得NVPsim只能描述非易失系统的运行时间、消耗能量等信息，而并不能描述非易失系统在掉电-上电过程中运行结果是否正确。

**对外设缺少描述**

一般来说，非易失系统都是从环境中获取电能，会经常面临电力不足，而且获取的电能十分有限，因此非易失系统一般并不适合做大量计算（只涉及处理器、内存），相反，非易失系统的工作任务常常是作为终端节点收集环境中的数据并发送给服务端（这种工作任务设计处理器，内存， 以及传感器与网卡等外设）。因此，对于非易失处理器和外设的交互行为的仿真往往是仿真器功能的重中之重。遗憾的是，NVPsim并没有对外设进行合理有效的仿真。

NVPsim基于gem5的Syscall Emulation（SE）模式，在这样一种模式下gem5并不提供任何外设功能，而是将程序运行中产生的对外设的请求（一般包含在系统调用，system call中）直接发送给仿真器运行所在的系统（linux）。这样一来，在使用NVPsim的过程中，用户只能够仿真处理器和内存的行为而不能触及外设，在某种程度上这使得NVPsim有着巨大的局限性，应用场景不足。

**扩展性较弱**

从工程的角度来说，NVPsim并不像gem5本身一样有着良好的可扩展性，这是由于NVPsim并没有对所有模块在掉/上电的行为进行通用的建模并进行合理的分层抽象，而是将整个系统掉、上电行为直接通过修改底层事件队列管理代码来进行描述。当用户需要和NVPsim不同的系统建模时，NVPsim并不能提供有效的接口让用户方便地修改、扩展系统的行为，用户在这种情况下需要大量阅读了解底层代码，十分不便。在这个意义上NVPsim更像是一种特定的非易失处理器的“专用仿真器”，而不是一个通用的仿真平台。

1.5 毕业设计内容

本次毕业设计的任务是基于gem5设计一种非易失系统的仿真框架（或仿真平台）：gem5-NVP，从而解决上文提到的当前仿真方式的不足。此仿真框架设计有如下目标：

1. 对所有模块在系统状态改变时的行为有通用的接口

这需要每一个模块都能够通过重定义相同的接口函数来完成对这个模块掉/上电行为的建模，且用户可以像添加gem5模块一样简便地添加gem5-NVP的模块，仅仅需要定义模块在系统状态发生变化时的行为。这个目标主要针对的是仿真框架中模块行为描述的自由性和仿真框架的可扩展性。

1. 支持外设的仿真，且易于从代码中调用

这个目标要求仿真平台中存在可以在gem5仿真器SE模式下使用的外设模块，且这个模块要能够方便地被用户编写的测试程序（benchmark）调用，不牺牲编写benchmark的简易度。

1. 易于配置，使用户远离复杂的底层代码

这个目标要求仿真框架中的一些参数、模块能够像gem5中的参数、模块一样通过简单的python配置文件进行配置。

为了完成以上目标，本次毕业设计的主要工作任务有：

1. 对gem5的所有模块的底层通用类进行重写，使得所有模块都获取描述有关能量行为的接口。
2. 为gem5添加模块间的有关能量的“连线”（能量接口），即Energy Port，这使得能量本身和有关能量的系统状态控制流（上电、断电请求）能够在模块间传递。此外，用户应该能够在python配置文件中方便地连接这些连线（类似gem5的内存系统中的Port）。
3. 添加能量管理模块，这是一个非易失仿真器所必须的，这个能量管理模块要负责能量采集、系统状态管理、能量检测。
4. 添加仿真器的外设模块，这个外设模块需要能够在SE模式下运行，且可以使用高级语言非常方便地调用。
5. 基于此仿真平台（我们称之为gem5-NVP）对各种非易失系统进行建模、测试（比如不涉及外设的计算系统、涉及外设的信息采集系统、涉及外设的信息发送系统）。

第二章：软件仿真器架构

2.1 软件采取的NVP整体架构

软件仿真器gem5-NVP的架构是建立在gem5的系统调用仿真（System Emulation）模式的基础上的，因此其基本模型和gem5的SE模式的结构类似。Gem5拥有较为自由的模块连接方式，支持用户自定义模块行为，每一个模块拥有“内存”接口，能够向其他模块发送内存访问请求，包含地址、可能的写入值、请求大小等。尽管gem5支持多种多样的模块连接方式，多数模型使用的就是传统计算机使用的体系结构，即中央处理器-总线-内存的结构。Gem5-NVP不需要对这个结构进行修改，所需要做的是在这个结构的基础上添加有关非易失的模块和功能。

整个软件仿真系统的框图如图所示：

【】

首先我们需要添加传统体系架构中不存在的能量采集和能量管理状态机模块，这部分模块负责收集能量，并在需要发生系统状态改变时通知/控制其他模块进行状态改变。这个模块的主要任务一是仿真从外界能量幅度（电压、电流、光照）到内部能量储存器的转换电路的行为，二是仿真系统能量状态机不断检查系统储存电压值并控制系统状态改变的行为。

此外，我们需要添加的是gem5中不存在的能量信息通信功能，在非易失系统中，模块间有着很多必需的能量通信，比如说，能量管理模块需要通知系统其他模块断电或者上电，各个模块需要告知能量管理模块自身消耗的能量等，因此，我们需要为模块添加与其他模块进行通信的功能，即每一个模块在“内存”接口之外还需要有“能量接口”。

最后，由于gem5的SE模式下不支持外设的行为，但我们的确在非易失系统中需要大量使用外设，为了日后仿真方便我们需要开发一个通用的外设模块来仿真非易失系统中外设的行为。

2.2 Gem5类继承关系

如图所示，【】。从中可见gem5中的每一个模块均为一个SimObject对象[gem5]，SimObject对象为模块定义好了一些模块通用的功能。这些功能来源于SimObject的三个父类，分别为EventManager、Serializable和Drainable。其中EventManager为SimObject提供了操作异步事件队列的功能，使得SimObject能够方便地在事件队列中添加、取消或者修改自身在其中插入的事件；Serializable提供的功能是使得任何SimObject能够将当前状态输出为串行比特流，这个功能的作用是能够使得SimObject能够设置checkpoint，当用户关心运行时某个状态时，能够将系统涉及到的所有SimObject的状态均保存成比特流形式的checkpoint，日后的运行就不需要再次从头执行，只需读入checkpoint即可；Drainable为SimObject提供的功能是能够使得SimObject能够正确结束自身的工作状态。

Gem5通过以上方式为所有模块的开发打下了基础，在SimObject的基础上开发者开发了各式各样的类，典型的类有各种各样的内存类（MemObject），包含了各种建模的处理器（CPU）、各种建模的内存模块（SimpleMemory、DRAM等等）以及总线模块（CoherentXBar），这些模块的共同点是都需要向其他模块传输内存访问、读取、写入信息，事实上MemObject正是提供了这样的“内存”接口，正如“软件采取的NVP整体架构”一章中所说。

2.3 为Gem5的仿真模块引入能量相关功能

在开发gem5-NVP的过程中，一个内在的逻辑就是任何模块都应当有消耗能量的接口，同时应当能够接收系统有关能量的通知（如“开关机”等等），Gem5-NVP不应该只针对少数我们关注的模块开发能量消耗和开关机功能，而是应给为所有模块提供通用的能量操作功能，基于此，我们认为和能量相关的功能同EventManager、Serializable、Drainable提供的功能处于同样的地位。为了实现这些要求，我们引入了“能量对象”（EnergyObject）作为SimObject的父类之一，如图所示【】。

EnergyObject类为模块提供了一个一些和能量有关的接口，这些接口将会在第三章中介绍。

2.4 Python控制端

除了上述C++端功能之外，gem5还使用了SWIG（Simplified Wrapper and Interface Generator）来对SimObject提供了对应的python端接口，任意从SimObject中派生的类在python端都有一个对应的同名类，在实际使用中，用户一般直接通过配置python端对象的方式来配置这个SimObject的行为，为了仿真某一系统，用户需要为这个系统创建一个python脚本，在这个脚本中定义这个系统中被使用的模块，配置这些模块的参数，并按照实际系统连接这些模块，这些模块（SimObject）存在一定的树状关系，比如说，CPU、总线、内存都是system模块的子模块，具体体现就是这些模块的对象是system模块对象的成员变量，定义好整个系统的参数和连接方式后，用户需要使系统的最上层system模块成为根模块root的成员变量，并告知gem5对root模块进行仿真。在运行时，gem5会采用深度优先搜索的方式遍历模块树，为每一个python模块建立对应的C++后端模块，接下来同样采用深度优先搜索的方式来为每一个C++后端模块进行初始化，在初始化的过程中，最初的事件被放入事件队列，这使得整个仿真过程开始。

Gem5-NVP中和能量有关的功能同样拥有从python端进行控制的功能，提供了众多方便的python接口，然而每一项功能的配置方式都有所不同，正如gem5本身的诸多模块一样，和能量有关的python配置细节将会在后几个章节中被分别详细介绍。

第三章：模块间能量信息交互

模块间能量信息的传输是一个软件非易失仿真器所有功能的基础，这一个部分包含了如下内容：模块间交换的能量信息有哪些类型，如何为模块添加能量信息传递的功能，模块之间的能量流路线是如何连接的。这一章节将会对这些问题提出解释。

3.1 “能量接口”

Gem5中存在内存接口“Port”，描述了系统中的各个模块如何与其他模块进行内存访问相关的通信，借鉴这个概念，gem5-NVP中提出了“能量接口”这一概念，能量接口和内存接口存在诸多相似之处，也有一些不同，这一部分将对能量接口进行介绍。首先，能量接口在后端C++代码中体现为“EnergyPort”类，能量接口能够互相连接，并能够互相发送信息。

3.1.1 能量信息

当需要发送信息时，能量接口会将如下简单的数据结构传送给对面的能量接口：

此数据结构中type代表着此能量信息的类型，一般来说，type为0意味着此能量信息代表发送端消耗了数值为value的能量，而其他数值的type对应的消息类型一般由能量管理模块EnergyMgmt（后文中会进行介绍）中的系统状态机进行定义，一个最简单的例子是，系统默认的简易状态机的消息种类有两种，type=1代表系统关机消息，type=2代表系统开机消息，用户仿真的系统如果有其他状态或者其他信息，可以自行编写系统状态机并进行定义。

3.1.2 能量接口种类

在仿真能量相关系统过程中，不同模块所处的地位不同，例如，常规模块的地位都是能量的消耗者和能量消息的接受者，而能量管理模块是能量的采集者、消耗者和能量消息的发送者。这两种模块的能量信息处理方式是不同的，因此能量接口种类应该有两种，事实上能量接口有两种，主接口（MasterEnergyPort）和从接口（SlaveEnergyPort）。

在连接时，主接口和从接口之间能够相互连接，一个主接口可以连接多个从接口，但是一个从接口只能有一个连接的主接口，也就是说，主接口和从接口之间的连接关系是一种“一对多”关系。此外，主接口和从接口拥有的功能也不同，由于主接口是管理者，管理多个从能量接口，因此它能够“广播”能量信息，将信息发送给自己管理的所有从能量接口，而从能量接口应该可以告知单个主能量接口能量信息（如自身消耗了能量）。

接下来的段落中描述了两种接口共有的和特有的成员变量和成员函数。

|  |  |
| --- | --- |
| 成员变量 | 描述 |
| port\_id | 此接口的接口号，调试用 |
| port\_name | 此接口的名称，调试用 |
| port\_type | 此接口的类型（主/从），调试用 |
| owner | 此接口的所有者，应该是一个SimObject对象，用于对象调用接口或者接口通知对象 |

|  |  |
| --- | --- |
| 成员函数 | 描述 |
| getPortId/setPortId | 读出/写入接口号 |
| getPortName/setPortName | 读出/写入接口名称 |
| setOwner | 告知接口的所有者（SimObject） |
| handleMsg | 处理接口收到的消息，处理方式为将消息告知接口的所有者（owner） |

|  |  |
| --- | --- |
| 成员变量 | 描述 |
| master | 此从接口对应的主接口 |

|  |  |
| --- | --- |
| 成员函数 | 描述 |
| setMaster | 告知接口对应的主接口 |
| singalMsg | 向对应的主接口发送能量消息 |

|  |  |
| --- | --- |
| 成员变量 | 描述 |
| slave\_list | 此主接口的所有从接口队列 |

|  |  |
| --- | --- |
| 成员函数 | 描述 |
| bindSlave | 将某一从接口放置在此主接口的从接口队列中，绑定从接口 |
| broadcastMsg | 向自身管理的所有从接口发送消息 |

3.2 能量接口与模块（SimObject）的关系

从上文的表格中可以看出，能量接口中存在“owner”变量，可以将其接收到的能量信息上传到上层的模块（SimObject）来处理，这是能量信息自下而上的传递方式，那么能量信息是如何自上而下从模块传递到能量接口的？上文中写道“能量模块”（EnergyObject）是为SimObject提供能量相关处理函数的父类，事实上，“能量模块”（EnergyObject）就是通过拥有能量接口的方式来使得模块（SimObject）所需要发送的能量信息传递到下层的能量接口并发送给其他模块的。在gem5-NVP中，每一个“能量模块”（EnergyObject）拥有两个能量接口（EnergyPort），在两个能量接口中有一个是主能量接口，有一个是从能量接口，这样设计的原因是每一个模块既有可能成为能量的消耗者又能够成为能量的管理者，比如说，能量管理模块采集能量，控制系统状态的同事，自身也在消耗能量，对于这种模块，其自身的从能量接口会连接到自己的主能量接口上。

除了拥有两个能量接口之外，EnergyObject还拥有一些成员函数来处理从能量接口中获得的能量消息或者发送某一些能量消息，下表【】中列举了和能量模块（EnergyObject）中和能量接口相关的成员变量和成员函数。

|  |  |
| --- | --- |
| 成员变量 | 描述 |
| \_seport | 这个模块拥有的从能量接口，这个接口被模块用来在消耗能量时向其能量管理单元发送能量消息或者从能量管理单元获得系统能量状态变化消息 |
| \_meport | 这个模块拥有的主能量接口，使用这个接口时，模块是作为管理者出现的，也就是接受其他模块的消耗能量消息或者发送系统状态改变消息，一般来说，只有能量管理模块（EnergyMgmt，后文会介绍）会用到这个能量接口 |

|  |  |
| --- | --- |
| 成员函数 | 描述 |
| getSlaveEnergyPort | 获取模块的\_seport，在模块初始化时用来连接各个模块的能量接口用 |
| getMasterEnergyPort | 获取模块的\_meport，在模块初始化时用来连接各个模块的能量接口用 |
| consumeEnergy | 这个函数的作用是告知此模块的能量层面的管理者这个模块消耗了一定能量，这个函数会通过模块拥有的\_seport发送消息给管理模块。注：为什么没有单独提取出发送能量消息的函数？这是因为一个模块拥有主从两个能量接口，能量消息有可能从这两个接口中的任何一个发送出去，但是消耗能量的能量消息一定是从\_seport发送出去的，因此我们只提取出了消耗能量这一函数。 |
| handleMsg | 处理其他模块通过能量接口发送过来的能量消息，当模块拥有的能量接口接收到消息时，将会通过此函数通知模块，然而这个由于每一个模块的处理方式不同，这个函数实际是一个虚函数，需要在具体的模块（比如某种建模的CPU）中被重写。 |

3.3 Python端配置

定义好了底端接口和代码之后，面临的问题就是如何连接这些模块，或者说，如何得知一个主能量接口连接了哪些从能量接口，一般来说，这种配置信息gem5都是通过python端（前端）来定义的，能量模块的连接方式同样也使用这种方法。编写C++的连接两个能量模块的函数极为简单，只需要获取二者对应的能量接口并且连接就可以了，我们可以使用SWIG（Simplified Wrapper and Interface Generator）将这个C++函数包装成python函数，来在python端进行调用，然而，这种方法存在着很严重的问题。上文提到了，在完成python端配置的最后gem5才会初始化所有的后端（C++）对象，也就是说在运行仿真的配置脚本时C++对象还没有被分配到内存中，如果直接调用这个swig函数的话，将无法获取对应模块（SimObject）的内存地址，造成错误。如果真的需要调用这个函数的话，需要在python的开始仿真函数的初始化C++对象后进行调用，这对于用户来说十分复杂，而且会破坏代码的层次性，将后端实现暴露给用户。为解决这个问题，可以在python端引入“能量接口引用”类EnergyPort（在python端的类，有别于C++端实现），这个类的作用是作为后端C++能量接口的引用，这个类同样派生出主能量接口引用和从能量接口引用。在python端同样每一个SimObject拥有一个主能量接口引用（命名为m\_engy\_port）和一个从能量接口引用（命名为s\_engy\_port）。

self.\_\_dict\_\_['m\_energy\_port'] = MasterEnergyPort(self)

self.\_\_dict\_\_['s\_energy\_port'] = SlaveEnergyPort(self)

我们可以直接为python端的引用编写函数来相互连接，当我们在python端连接两个引用，引用内部会记录此接口都连接了哪些其他接口，这样一来，我们就可以在gem5初始化C++端对象时根据引用记录下的信息来相互连接。

为了进一步使用户连接能量接口变得便捷，我们可以直接介入python端SimObject的內建函数\_\_setattr\_\_，这个函数能够自定义python为对象的成员变量赋值时的行为，可以在这个函数里判断被赋值的是否是能量接口引用，如果是的话则自动把赋值行为变为连接能量接口行为，这样一来，就可以通过如下简单的代码连接两个模块的能量接口（假设为cpu的从能量接口和engy\_mgmt模块的主能量接口）：

cpu.s\_engy\_port = engy\_mgmt.m\_engy\_port

第四章：能量管理模块

本部分将会介绍为非易失处理器（NVP）引入的能量管理模块，能量管理模块是非易失系统与传统系统区别的一个重要部分，接下来本文将会从简介、能量收集、系统（能量）状态机几个角度来对能量管理模块进行介绍。

4.1 能量管理模块简介

能量管理模块（EnergyMgmt）如图【】所示，它在系统中扮演的角色是收集能量、储存收集到的能量、不断采集当前能量储存器（电容）的电压，根据能量状况控制系统的其他模块进行启动、备份、恢复、断电等状态变化。能量管理模块会接收用户提供的能量配置文件（energy profile），这个文件内部的内容代表着环境中与非易失系统能量收集相关的物理量随时间的变化，这个物理量可能是光强、电压、或者是震动幅度，用户需要为能量配置文件定义时间单元，这代表着能量配置文件中两个相邻条目的采集时间差。能量配置文件是非易失仿真器运行所必需的，是非易失系统中能量管理模块采集能量的依据。

能量管理模块的工作模型如图所示【】，从图中可以看出，能量管理模块的工作事件有两条触发路线：

第一条触发路线是能量的收集，收集能量的事件是周期性触发的，也就是说用户可以规定一定的时间作为能量管理模块的能量收集时间单元，每经过这样一段时间，能量管理模块将从能量配置文件（energy profile）中读入一个外界环境能量强度条目，得到此时外界的能量强度，接下来能量管理模块将读取此时电容的电压值，并将外界能量强度和内部电容电压值两个参数告知能量收集模块，能量收集模块计算出收集后电容电压返回给能量管理模块，能量管理模块修改电容电压值，并将电压值的改变告知系统状态机。

第二条触发路线是能量管理模块所管理的其他模块消耗了能量，如上一章所说，模块可以通过触发“consumeEnergy”函数来消耗能量，一旦调用了这个函数，这个消息将会通过能量接口传送给能量管理模块，能量管理模块将所消耗的能量从电容中扣除，计算出电容电压的变化，并且将电压改变告知系统状态机。

根据上述描述，能量管理模块负责有关能量管理的整体流程控制，一些具体的工作，比如外界能量（能量配置文件energy profile中的条目）是如何转换成电容中的能量的，或者系统状态是如何改变的，是由能量收集模块和系统状态机模块控制的，下文将分别具体地介绍这两个部分并介绍整个能量管理模块的python端配置方式。

4.2 能量收集功能

在非易失系统中，都会存在较为复杂的电路来完成能量收集的工作，这些电路将会将环境中的待收集的目标能量首先转换成电能，再经过一定的整流、变压、负载匹配等步骤转换成能够储存到能量储存装置（一般是电容）中的电能【】。Gem5-NVP是一个行为级仿真器，因此无意对复杂的电路细节进行仿真，仅此忽略了电能转换的中间步骤，在Gem5-NVP看来，外界环境中的某种能量强度经过一系列的函数变化被成为了电容中电能的一部分，因此Gem5-NVP的能量收集模块在一次能量收集时得到的信息有两个：外界能量强度，电容此时状态（电压），能量收集模块根据某一些规则将计算出此次能量收集后电容的状态（电压），并且将这个数值返回给能量管理模块。

Gem5-NVP的一个原则是，如果非易失体系架构中的某一些方面值得被研究或是研究的热点，则将这一个部分在软件仿真器中编写为可插拔的模块，以方便用户进行功能的改变和仿真。显然，能量收集单元是一个当前的研究热点，因此能量收集模块是一个可插拔的模块。Gem5-NVP为能量收集模块提供了基类BaseHarvest，这个类的类成员函数double energy\_harvest(double energy\_harvested, double energy\_remained) 需要在子类中被重写，也就是说，用户如果想自定义能量收集的方式，就需要从BaseHarvest中派生一个简单的类，并重写计算函数energy\_harvest，最后将一个这个类的对象“插入”能量管理模块。

如果用户不进行任何配置，Gem5-NVP提供的能量收集模块是将外界能量强度以线性的方式累加到电容中的简单能量收集模块SimpleEnergyHarvest。

4.3 系统状态机模块

从图【】中可以看出，每当系统电容中储存的能量发生变化时，能量管理模块将通知一个系统状态机。这个系统状态机的主要功能是，随时监测系统此时的能量状态（电容的电压），并根据这个电压的变化来判断系统是否需要发生状态改变（比如是否开始备份、是否休眠、是否开机等），而一旦需要进行状态改变时，状态机将在能量接口上向被能量管理模块管理的其他模块发送状态改变的消息。这样一来，可以给系统状态机抽象出两个任务：任务一是定义系统的状态和状态转换，并在运行时实时监控系统能量并维护这个状态机的行为；任务二是为系统的状态改变定义响应的能量信息，也就是定义上文所述的能量信息EnergyMsg除了通知能量消耗（type为0）之外还能够通知哪些信息（type为其他值的意义），在运行时，系统状态机会将系统的状态改变时对应的状态信息通过能量管理模块发送给系统的其他模块。

由于系统状态机也是非易失处理器的一个研究重点，系统状态机模块和能量收集模块相似，都是可插拔的，也就是说，Gem5-NVP为其提供了拥有通用接口的基类BaseEnergySM，用户需要根据自身需求来进行继承，并实现某些成员函数的行为。

基类提供了接口void update(douhble \_energy)，这个成员函数是一个虚函数，需要在子类中被重新定义。接口通知了系统能量的变化，参数\_energy是变化后的电容电压，根据发送的这个能量值和此前发送的能量值，用户可以在系统状态机类内部维护系统的状态，进行必要的状态更新，当系统状态机需要通知系统进行状态改变时，需要将对应信息放入一个EnergyMsg中，并通过基类提供的函数接口void broadcastMsg(const EnergyMsg &msg)通知能量管理模块发送这一条信息。需要值得注意的是，系统的状态种类、系统的状态迁移都是由这个系统状态机模块进行定义的，因此对应状态改变的信息也需要这个模块进行定义（比如说EnergyMsg::type为1代表什么，为2又代表什么），当用户完成系统状态机设计后，还需要编写被能量管理模块所管理模块的handleMsg函数，根据定义的能量消息类型在接收到消息时进行合适的操作。

当用户未自定义系统状态机模块时，默认的系统状态机是SimpleEnergySM，这个状态机只有开机、关机两个状态，当系统能量由负到正时控制系统开机，当系统能量由正到负时控制系统关机。此外，Gem5-NVP还提供了拥有不同开关机阈值的简易状态机，称为TwoThresSM，可以在Python端由用户直接配置使用。

4.4 能量管理模块类成员介绍

这一部分简单介绍能量管理模块（EnergyMgmt）拥有的成员变量和成员函数。

成员变量：

|  |  |
| --- | --- |
| 成员变量 | 简介 |
| time\_unit | 能量配置文件和能量采集的时间单元 |
| energy\_remained | 系统电容的电压值 |
| energy\_harvest\_data | 从能量配置文件中读取的所有环境能量强度条目 |
| state\_machine | 指向系统状态机的指针 |
| harvest\_module | 指向能量收集模块的指针 |
| event\_harvest | 触发能量收集的周期性事件 |
| event\_msg | 向被管理模块发送状态变化信息的事件 |
| msg\_togo | 系统状态机告知发送的信息 |
| path\_energy\_profile | 能量配置文件的路径 |

成员函数：

|  |  |
| --- | --- |
| 成员函数 | 简介 |
| consumeEnergy | 能量消耗的处理器 |
| broadcastMsg | 发送系统状态改变消息的函数 |
| broadcastAsEvent | 以事件形式发送系统状态改变的函数（不直接调用broadcastMsg是为了避免出现程序出现同时性缺陷） |
| handleMsg | 接收并处理能量信息（从基类继承） |
| energyHarvest | 从能量配置文件中读取条目并触发能量采集事件 |
| readEnergyProfile | 初始化是读取能量配置文件的函数 |

4.5 Python端配置

能量管理模块EnergyMgmt在Python端也存在着对应同名实现，用户需要创建系统的能量管理模块的对象：

接下来可以通过四个参数对能量管理模块进行配置：

|  |  |
| --- | --- |
| 变量 | 说明 |
| path\_energy\_profile | 能量配置文件（energy profile）的路径 |
| energy\_time\_unit | 能量配置文件和能量采集的时间单元，如“10us” |
| state\_machine | 使用的系统状态机，默认为SimpleEnergySM |
| harvest\_module | 使用的能量采集模块，默认为SimpleHarvest |

如：

最后，我们需要将所有被控制模块的从能量接口与能量管理模块的主能量接口连接，比如，在某个系统中如果需要管理cpu的上电、掉电行为，需要配置为：

第五章：外设行为建模

这一部分将会介绍在Gem5-NVP中引入的外设模块（称为虚拟外设，Virtual Devices），虚拟外设能够帮助我们进行有关非易失处理器与外部硬件进行交互的过程。事实上，对于外设的仿真可能是一个软件非易失仿真器最重要的功能，因为绝大多数非易失处理器的工作方式都是与外设交互，由于非易失处理器功率低，运算速度慢，因此非易失处理器所在的节点一般不处理大规模的运算负载，而是作为终端获取环境信息并进行简单的预处理，常用的非易失节点，比如心脏起搏器监测节点、山体应力监测节点、水流监测节点，都遵从这个工作模式，因此，非易失处理器的行为是和外设紧密相关的，一个可用的非易失处理器软件仿真平台必须拥有对外设进行建模的功能。

本部分会先对虚拟外设进行简介，接下来介绍虚拟外设的概念与工作模式，最后介绍虚拟外设的地址解析与使用方式。

5.1 虚拟外设简介

虚拟外设也是外设，正如Gem5的文件夹“src/dev”中的外设一样，它通过总线与CPU联系在一起，且和CPU、内存一样拥有同样的内存访问协议，也和传统的Gem5外设一样拥有一定的物理地址空间。然而，虚拟外设与Gem5传统的外设有着一些区别，正是由于这些区别的存在，引入虚拟外设才是必要的。下面首先将介绍Gem5传统外设在进行非易失处理器仿真时面临的问题，接下来介绍虚拟外设与传统外设的不同（实际上是为了解决问题）。

Gem5传统外设涵盖了计算机外设的方方面面，从PCI设备到鼠标、键盘应有尽有，然而Gem5传统外设虽然全面并且精细，却并不易于使用。在Gem5中，只有完整系统仿真模式（Full System Mode，FS Mode）才能够调用外设，这是因为一般而言外设的工作流程一般涉及到操作系统，比如说：外设会触发中断，这时就需要操作系统有线程切换管理功能；外设需要用对应的物理地址调用，这时需要操作系统在虚拟地址空间上开辟一段专用的空间供程序来调用外设。但是，运行FS模式需要用户寻找兼容特定指令集的系统镜像，同时正确配置全部硬件，再将需要运行的程序再这个系统镜像中编译运行，这无疑是十分复杂的，如果用户仅仅是为了仿真某些特定任务在非易失系统下的工作效率，那么以上复杂的配置使不必要的。

此外，Gem5传统外设并没有对发生断电、或者电力恢复时的特性进行建模，由于硬件种类太多，对非易失、易失特性建模的难度十分巨大，这为非易失仿真器中使用这些传统外设带来了额外的难度。

基于以上的原因，我们需要引入能够在系统调用仿真模式（Syscall Emulation Mode， SE Mode）下能够方便进行调用，且能够对大部分外设的行为进行通用有效建模的外设，此外设还需要能够对能量有关的事件（上电、掉电）进行建模，供用户进行非易失处理器仿真用。这个外设就是所谓的“虚拟外设”（Virtual Devices）。

首先，虚拟外设并不像传统外设一样进行实际有意义的操作，传统外设每一个都有专属的工作，比如说网卡就会向外部发送网络包，磁盘外设就仿真磁盘的寻址、写入、读出功能，虚拟外设并不进行上述工作，它只是装作繁忙，这是因为用户在仿真非易失系统中外设时往往只在意外设的时间片调度，而不在意外设真的做了什么。

其次，虚拟外设能够工作在SE模式下，这是因为Gem5-NVP引入了虚拟地址解析功能，使得用户在SE模式下没有操作系统时同样可以将某些虚拟地址映射到外设的物理地址上，并进行调用，且虚拟外设在触发中断时可以不经过操作系统（涉及到中断向量等），直接在CPU上触发跳转。

最后，虚拟外设对能量相关的行为进行了通用的建模，能够有效仿真外设在发生断电、上电时的行为。

5.2 虚拟外设工作流程

虚拟外设的设计原则是使用非常少的参数就能对外设的整个工作流程进行建模，用户往往并不关心外设真的做了什么，而只关心外设工作时需要的时间和能量，因此，有关虚拟外设的参数大多都与时间和能量相关。

图【】是一个虚拟外设在常规情况下（能量充足，不发生断电重启）时的工作流程。首先，CPU会操作和虚拟外设相关的虚拟内存地址（如何映射后文会提到），写入这些内存的作用是模拟CPU配置外设的过程，这段配置时间在图中为“delay\_set”。当虚拟外设接收到请求时，如果其本身并没有在工作，则进入工作状态，工作时间称为“delay\_self”，当工作完成后，虚拟外设会向CPU触发一个中断，在这个中断中CPU完成接下来的收尾工作，中断消耗的时间称为“delay\_cpu\_interrupt”。

图【】描述了当出现断电重启事件时虚拟外设的工作流程。当发生断电时cpu和虚拟外设都停止工作，当重启后，CPU可能会需要一些时间来重新初始化这个外设，这段时间称为“delay\_recover”，这段初始化时间结束后，外设继续工作，工作时间为“delay\_remained”，值得注意的是，当外设属于工作可以被打断的类型时，图【】中delay\_remained=delay\_self-time\_before\_poweroff，也就是说，delay\_remained不是被提前确定的，这段时间应当是完成剩余工作的时间。

综合以上的描述，一个虚拟外设的行为主要由以下参数确定：

|  |  |
| --- | --- |
| 参数 | 描述 |
| delay\_self | 虚拟外设完成工作需要的时间 |
| delay\_set | CPU最初调用外设初始化所需时间 |
| delay\_recover | 当断电重启后CPU重新配置外设所需时间 |
| delay\_remained | 外设从断电重启后继续完成工作所需时间 |
| delay\_cpu\_intrerrupt | 虚拟外设触发的中断所需时间 |
| is\_interruptable | 外设是否是可以被打断的，如果是，则delay\_remained会在运行时被确定 |

5.3 外设地址解析

一般来说，CPU调用外设的方式是将请求通过写入外设对应的地址的，正常来说，外设对应的地址是由操作系统（Operating System）来确定的，但是虚拟外设工作在SE模式，不存在操作系统也不存在对外设的调用，为了使CPU能够调用虚拟外设，我们需要手动为虚拟外设进行地址解析。在运行有关虚拟外设的仿真之前，用户需要定义两个地址空间，第一个是虚拟外设的物理内存地址空间，这段空间不应当和内存的地址空间冲突，第二个空间就是虚拟外设在虚拟地址空间中的内存范围，这段虚拟的内存空间的大小需要和外设的物理内存空间拥有相同的大小。

虚拟外设的地址空间中的第一个字节为控制字节，其结构如下：

|-high-----------------------low-|

|-4 bits-|correct|finish|work|set|

这个字节的最低位被用来调用虚拟外设，程序可以将这一位写入1，这意味着CPU此时要调用虚拟外设，这个字节的其他位为只读，代表着虚拟外设的工作状态。

5.4 Python端配置

虚拟外设同样有前端（Python）和后端（C++）两部分的代码，Python端中虚拟外设的类名为VirtualDevice，其成员变量如下包含了前文描述的虚拟外设的所有时间参数。

为了仿真虚拟外设的行为，除了创建虚拟外设对象之外，用户还需要在系统中开辟虚拟外设对应的物理内存空间与虚拟内存空间，并将二者对应，系统对象System中被引入了vdev\_ranges（虚拟外设物理地址空间）与vaddr\_vdev\_ranges（虚拟外设虚拟地址空间）两个成员变量，供用户定义外设的内存地址空间，在确定好这两个变量后，用户还需要将物理地址空间中的对应条目分配给特定的外设对象的range成员变量。

以下为一个配置虚拟外设的例子：

5.5 程序端配置

用户编写的可执行程序如果想调用虚拟外设，可以直接访问外设的虚拟地址范围，使用的函数为内存地址映射的mmap函数。

以下为访问虚拟地址处于1000MB处的虚拟外设的例子：

第六章：测试与仿真

第七章：结论

参考文献

[1] Gu, Yizi, et al. "NVPsim: A simulator for architecture explorations of nonvolatile processors." *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*. IEEE, 2016.

[2] Graphics, Mentor. "ModelSim." (2007).

[3] Binkert, Nathan, et al. "The gem5 simulator." *ACM SIGARCH Computer Architecture News* 39.2 (2011): 1-7.

[4] Wang, Yiqun, et al. "A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops." *ESSCIRC (ESSCIRC), 2012 Proceedings of the*. IEEE, 2012.