|  |  |
| --- | --- |
| 编号： |  |
| 密级： | 内部 |

基于80x51虚拟单片机的计算机数字仿真系统

80c32单片机资源建模

中国科学院卫星软件评测中心

2015年12月1

目 录

[第一章 概述 1](#_Toc437253835)

[1.1 应用背景 1](#_Toc437253836)

[1.2 80c32单片机简介 1](#_Toc437253837)

[1.3 80c32仿真平台整体架构 3](#_Toc437253838)

[第二章 51架构下的指令集模拟器 5](#_Toc437253839)

[2.1 51指令集模拟器 5](#_Toc437253840)

[2.2 中断的建模 8](#_Toc437253841)

[2.3 外部接口 10](#_Toc437253842)

[2.4 调试接口 11](#_Toc437253843)

[第三章 80c32片上外设资源模型 13](#_Toc437253844)

[3.1 定时器0/1 13](#_Toc437253845)

[3.2 定时器2 14](#_Toc437253846)

[3.3 串行通信接口 14](#_Toc437253847)

[第四章 验证 16](#_Toc437253848)

[4.1 测试程序 16](#_Toc437253849)

[4.2 测试环境 16](#_Toc437253850)

[4.3 测试结果 16](#_Toc437253851)

# 第一章 概述

## 1.1 应用背景

以指令集模拟器（Instruction Set Simulator）为核心的全系统仿真平台有着诸多好处。从系统硬件设计的角度来看，传统的以硬件描述语言(比如Verilog HDL，VHDL等)搭建的平台，由于建模精确到了门一级电路，所以在这样的平台上进行功能验证和架构探索，将需要花费大量的开发时间。同时，这种底层平台对架构设计人员的容错性是非常低的，一旦架构设计出了问题，那么前端到后端的所有中间环节都得修改，甚至推倒重来。相反，基于高级建模语言的全系统仿真平台能够在很大程度上避免上述问题。在设计初期，设计人员利用高级语言快速搭建系统的功能模型，然后根据需要确定的参数属性对该功能模型进行有控制的细化，直到系统所有的参数被最终确定。在这个过程中，设计人员可探索不同的软硬件划分方案，同时对用户需求的功能进行充分的验证。一旦软硬件划分确定后，不同的开发小组就可以同时进行软硬件的开发，并以该全系统仿真平台为最初的测试激励产生平台。

从系统软件设计及测试的角度来看，全系统仿真模型在实际硬件实现之前，为软件的设计和测试提供了一个平台。软件设计人员可在该平台上探索不同的系统软件实现方案，以获得最优策略。对于软件测试，与直接在实际的嵌入式硬件平台上进行测试相比，仿真平台一方面可以大幅降低测试过程中对实际硬件平台的需求，进而降低软件测试的成本；另一方面，也增加了软件测试过程中的可控性和可观测性，仿真平台可提供各种组件以便于软件的测试。最重要的是，仿真平台可加速软件的测试，仿真平台的指令执行速度一般都远大于实际的嵌入式硬件平台的指令执行速度。

基于上述优点，系统仿真平台在国内外得到了广泛的研究和开发。在本文中，我们不仅将详细介绍对80c32嵌入式单片机系统的模拟，更重要的是提出了一种处理器系统的建模框架。该框架不仅适用于51指令集架构，对其他指令集架构下的处理器系统建模也有一定的启发意义。

## 1.2 80c32单片机简介

80c32单片机是51微控制器家族中的一员，是标准8051单片机的扩展版。标准8051单片机架构的特点包括以下几点：12MHz的时钟频率；32个数字输入输出管脚，被分为四组，每组8个管脚；128字节的内部数据存储器（RAM）；4K字节的内部程序存储器（ROM）；两个16位的定时器；五个中断源，其中两个是外部中断源，共两个中断优先级；还有一个可编程的全双工串行通信接口。这些内部硬件资源的大致组织结构如图1-1所示。



图1-1 8051单片机内部资源结构

其中SFR（Special Function Register）是51架构中比较特别的地方。它是一个特殊功能寄存器空间，其中包括通用寄存器、累加器以及其他一些辅助指令执行的寄存单元。除此之外，单片机内部外设资源，比如定时器和串口等，它们的控制寄存器也在该SFR空间中，而不像其他架构中那样，外设单元的控制寄存器在其内部，不与处理器内部寄存器统一编址。这一点将使51微控制器的建模与其他架构略有不同，后文将详细描述。8051单片机的外部接口如图1-2所示，通常这是用户所看到的单片机接口。

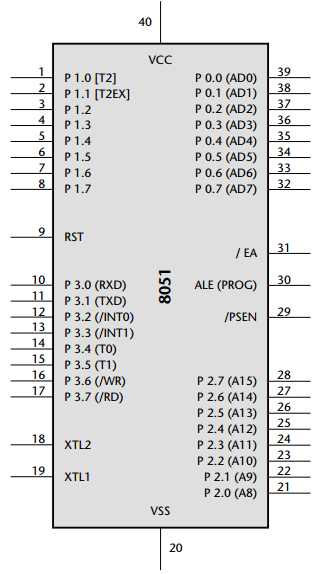


图1-2 8051单片机的外部接口

8051单片机的地址总线宽度为16位，数据总线宽度为8位。其中P2一般用作地址总线，用来传输高8位地址；P0是地址和数据的复用总线，用作地址总线时，可用来传输地址的低8位。P0和P3口一般用作外部I/O，但也可以用作特定的功能，在不同的应用场景，可以通过软硬件设置来进行选择。

对于8051单片机，还有一个重要的问题，就是它的存储器组织结构。如图1-3所示，在8051架构中，存储器采用哈佛结构来组织，程序和数据分开存储。对于程序存储器，单片机内部ROM和外部存储器统一编址在64K地址空间中。在标准8051中，片上ROM为4Kbyte，占据了地址空间的低4K字节，剩下的地址空间全部留给了外部程序存储器。数据存储器分为三部分：内部RAM，SFR和外部数据存储器。这三部分数据存储器各自独立编址，地址空间存在重叠，处理器可通过寻址方式来进行区分。内部RAM的大小一般为128个字节或者256个字节，低128字节可以通过直接或间接寻址方式来进行访问，高128字节一般只能通过间接寻址的方式来进行访问。SFR的地址空间和高128字节内部RAM的地址空间重叠，但SFR地址空间只能通过直接寻址的方式来进行访问，这样就可以区分二者。对外部数据存储器的访问是通过特有的指令来进行的，虽然其地址空间与内部数据存储器重叠，但不会引起访问冲突。

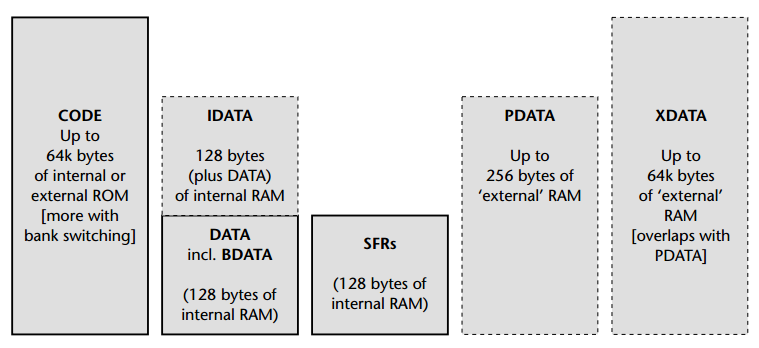


图1-3 51微控制器的存储器组织结构

80c32单片机是标准8051单片机的一个扩展，它在8051的基础上添加了一个定时器timer2，同时为该定时器引入了一个中断源。其内部RAM的大小为256个字节，没有内部ROM。其指令集架构和存储器组织结构均与标准8051单片机相同。要建模一个80c32系统的硬件资源，至少需要完成以下几个组件的模型：51指令集的模拟器、中断控制器、三个定时器和一个串行通信接口。在本设计中采用C高级语言来对这些模块进行建模。

## 1.3 80c32仿真平台整体架构

实现80c32单片机仿真系统主要涉及两个工作，一是要建模如图1-1中所示的单片机硬件资源；另外就是要提供一个模拟硬件系统中处理器和外设异步操作的机制。其中，这种异步操作机制主要由统一时间管理系统来提供，这部分内容在其他文档中介绍，此处不再叙述。本文主要介绍80c32单片机各硬件资源模块的建模细节以及对指令执行的优化方法。



图1-4 80c32系统仿真平台整体结构

图1-4中给出了80c32单片机系统仿真平台的整体架构。其中带阴影的外设方框表示用户添加到系统中的自定义外部设备，除了外部程序存储器EXTROM之外，其他模块均为80c32单片机的片上资源。由图中可以看出，在仿真平台中CPU在执行指令时可以通过两条路径与外围设备进行通信：直接函数调用或通过回调。通常，CPU访问外部设备是通过读写其控制寄存器来实现的。外部设备的控制寄存器可以与数据存储器统一编址，也可以有自己的独立I/O空间，当然，也可以如51微控制器处理片上外设那样，将片上外设的控制寄存器放在SFR地址空间中。不同的编址方式，其对应的操作就不一样，当然，在仿真平台中对应的处理方式也就不同。

直接函数调用方式一般用来模拟有大量数据交换的主从设备之间的通信。主设备发起数据访问请求，从设备在接到请求后立即给出被请求的数据。从设备的通信是完全被动的，只有在主设备有访问请求时才做出应答，这类似于处理器对程序存储器的操作。这种通信方式可以被简单地抽象为一个函数调用，主设备需要数据时调用该函数，从设备只需要实现该函数即可。

回调与直接函数调用方式的不同之处在于，外部设备可将自己的操作与CPU的某一个动作关联起来，一旦CPU产生该动作，就触发该外设操作的执行。比如定时器将计数操作与CPU访问该定时器的控制寄存器关联起来，一旦CPU访问该寄存器，该计数操作将被触发执行。

回调和直接函数调用方式在本质上都是将外设的操作封装到一个函数中，处理器在触发条件满足时调用该函数，来驱动外设的运行。只不过回调方式的触发条件自由度更大，可由软件设计人员根据需求来自由设置，且执行效率极高。在80c32单片机中，由于存储器组织结构复杂，片上外设的控制寄存器不均匀分布在SFR地址空间中，同时考虑到模拟器的执行效率，我们在仿真系统中提供了这两种处理器与外设的通信方式，以应用于不同的场景。

# 第二章 51架构下的指令集模拟器

## 2.1 51指令集模拟器

8051微控制器的CPU为8位，也就是其寄存器和数据总线的位宽为一个字节，指令操作以字节为单位。该系列微控制器采用的指令集是一致的，为CISC结构，总共有111条指令，变长指令编码格式，指令的长度可分别为1、2或3个字节，其中第一个字节为操作码。该指令集支持6种寻址方式，分别为直接寻址，寄存器间接寻址，立即数寻址，特定寄存器寻址和索引寻址。指令主要分为五种类型，分别为算数指令，逻辑指令，数据传输指令，位操作指令和控制转移指令。

要建模80c32单片机中的处理器，最主要的工作就是实现对指令执行过程的模拟。指令执行一般分为如下几个步骤，首先根据当前程序计数器中的值从程序存储器中取出指令，然后对指令进行解析，再将解析的结果发送到执行单元进行执行，最后用得到的结果更新CPU的状态和存储器中的值。目前，对指令集模拟器（ISS，Instruction Set Simulator）建模方法的研究已存在大量的研究成果，从整体来看，主要分为两种类型：解释型和编译型。编译型ISS又分为静态编译类型和动态编译类型。其中，解释型ISS最直观，其结构如图2-1所示。

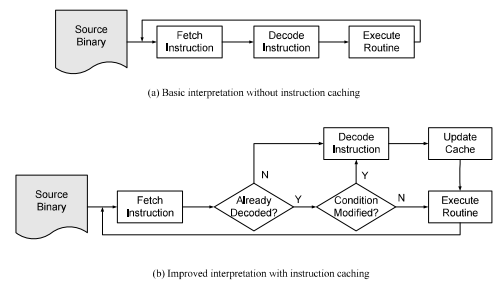
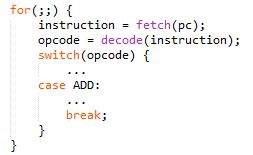


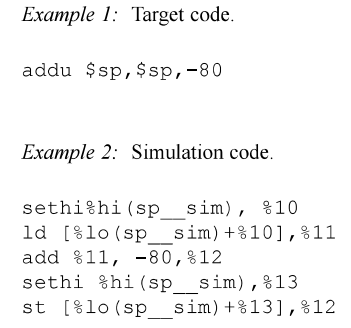
图2-1 解释型ISS的核心结构

图2-1(a)中给出了一个最简单的解释型ISS的框架。仿真目标机的源二进制文件一般在系统初始化时被镜像到了存储器中。ISS在执行时，从存储器中取出一条指令，然后译码执行，更新PC的值，然后重复上述操作。其核心操作的伪代码可表示如下：



这种ISS结构便于理解，且建模比较容易，但其指令执行效率却不高。ISS不维护任何有关指令译码和执行的信息，每一条指令都要重复经历这三个执行步骤，而不管该指令是否刚刚才执行过。如果指令集中指令编码格式复杂，对应的指令译码过程会耗费大量的时间。图2-1（b）中对上述简单结构做出了一点改进：在ISS中维护最近几条指令的译码结果。在执行指令时首先会到该缓存中去查询，看是否该指令已被译码。如果是，就直接将译码结果取出执行；否则，就解析该指令，然后添加到缓存中。由于程序执行存在局部性原理，如果缓存大小选取适当，这种改进将大大提高指令的执行效率。

与解释型ISS直接执行目标代码不同，编译型ISS在执行之前，会将目标代码翻译成宿主机的可执行代码。所以，编译型ISS主要由两部分构成：代码翻译和执行部分。其中代码翻译部分与编译器的功能很类似，它将目标二进制代码转换为在宿主机下对仿真处理器中的数据对象的操作指令。下面给出一个示例：



仿真目标机的一条ADDU指令在宿主机中需要5条指令来完成，sp\_sim是在宿主机内存中维护的一个变量，用来模拟目标机中的SP寄存器。由于目标机中的所有寄存器一般都是用宿主机中的内存变量来进行模拟的，所以一条目标指令通常情况下需要多条宿主机指令来模拟。这种提前编译好的方式使得目标指令在宿主机上的执行几乎不存在任何其他开销，如果翻译过程中的优化被做得足够好，那么目标指令的仿真速度将非常快。

静态编译型ISS是指目标程序在执行前，全部翻译成宿主机下的可执行程序。这种方式将取得最快的指令仿真速度，但其也存在几个问题。首先是在仿真代码执行的过程中，不能动态修改目标程序；另一个问题是对静态编译型ISS添加代码调试功能将变得非常困难。图2-2中给出了一个静态编译型ISS的结构，它是通过将目标代码先翻译成中间代码（类似于C代码），然后再编译链接生成宿主机下的可执行代码。这就将翻译部分分成了两个模块，类C代码的生成器模块和类C代码在宿主机下的编译和链接模块。

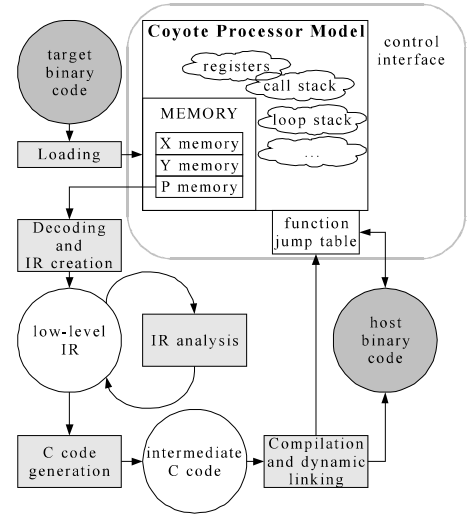


图2-2 一种静态编译型ISS的结构

动态编译型ISS在执行目标代码之前，会将目标代码以控制转移指令为分界进行分块。翻译过程是以代码块为单位进行的。ISS中会维护一个较大的缓存，用于存放最近一段时间执行的代码块的翻译结果。在ISS执行过程中，它首先回去查询该缓存，以判断当前处理指令所在的代码块是否已经被翻译成宿主机代码。如果已翻译，则直接将该代码块取出来执行。否则，将调用翻译模块翻译当前代码块并执行，然后存入缓存中。动态编译型ISS的指令执行效率很高，同时它可以解决静态编译型ISS中存在的问题，一旦代码被修改，如果对应的代码块存在于缓存中，则将被视为无效，需要重新进行翻译。图2-3中给出了一种动态编译型ISS的结构。为了更快的查询缓存，该结构中还建立了一个快表TLB。

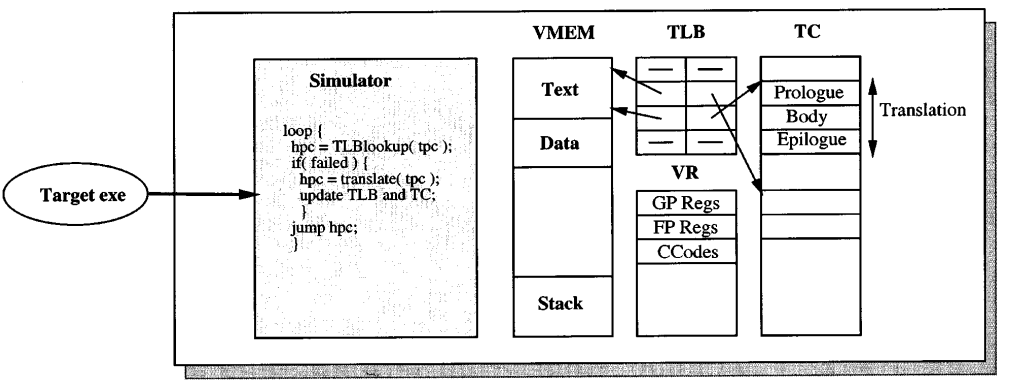


图2-3 一种动态编译型ISS的结构

与其他架构下的指令编码格式不同，51指令的编码格式中操作码均在第一个字节，且不同的操作码代表着不同的操作。所以只要获取了指令的操作码，就可以唯一确定该指令对应的操作例程。基于这个特点，指令译码的过程就可以设计得非常简单且高效。因此，在本设计中不采用复杂的编译型结构，而是采用解释执行的方式来建模51指令集模拟器。为了优化指令译码过程，设计中将每一个操作码对应的操作都封装在一个独立的函数中。模拟器将维护一张表用于存放这些操作函数的地址，每个操作函数的地址在表中的位置，可用操作码来直接索引。一旦获取了指令的操作码，便可以立即得到该指令对应的操作函数地址，调用该函数便开始了指令的执行。



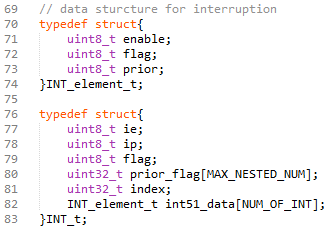
图2-4 51指令集模拟器的指令执行流程

图2-4中给出了51指令集模拟器在执行一条指令时的数据通路。模拟器首先根据当前PC的值从程序存储器中取出一个字节，也就是指令的操作码，然后将该操作码作为指令执行函数地址表的索引，从中取出对应的操作函数地址，然后执行该函数。如果是跳转指令，那么在该函数中将更新PC的值；如果是算数逻辑指令，那么在该函数中将首先根据PC的值，从程序存储器中取出指令的剩余部分，然后到对应的存储器中取出操作数，执行对应的操作，更新CPU状态和存储器中的内容。最后更新PC的值，开始下一条指令的执行。

## 2.2 中断的建模

80c32单片机有6个中断源，其中有三个定时/计数器中断，一个串口中断以及两个来自外部的中断。定时/计数器0和1在计数溢出时会将TCON寄存器中对应的中断标志位TF0/TF1置位，来向处理器请求中断；定时/计数器2在计数溢出时会置位T2CON寄存器中的相应位；串口在发送或接受到一个字节的数据后，将置位SCON寄存器中对应的中断标志位TI或RI；外部中断由INT0和INT1这两个管脚产生，可设置为下降沿触发或电平触发。每个中断源有两个优先级可以选择，允许嵌套中断。在处理高优先级中断服务程序时，不允许被低优先级中断打断，反之低优先级中断的执行可以被高优先级中断抢占。在优先级相同的情况下，采取轮转调度的方式。中断控制器的功能主要是根据优先级或轮转调度算法来从当前的中断源中选择一个，然后将该中断源的编号发送给处理器。处理器在接收到该编号后，会执行相应的中断处理程序。

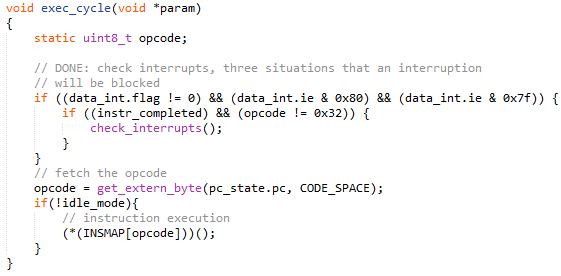
在本设计中，中断控制器的功能在模拟处理器，也就是ISS中实现。在ISS中，将为每一个中断源建立一个数据对象，同时维护一个中断管理对象，用来存储系统中中断涉及的所有信息。其声明分别如下：



同时，ISS还为设备请求中断提供了一个接口，其声明如下：



其中参数type表示中断类型，src表示引起该中断的事件，比如对于串口中断来说，是接收一个字节产生的中断，还是发送一个字节完成后产生的中断。当外部设备在执行过程中，满足中断条件后，将调用该中断请求函数来设置ISS中维护的中断相关的数据对象。ISS在开始执行一条指令之前，首先将通过调用check\_interrupt()函数来查询当前是否有可执行的中断。该函数实现了中断控制器的所有功能，它查询ISS维护的中断相关的数据对象，获取当前请求中断的中断源及其优先级等信息，然后根据仲裁算法选出当前需要执行的中断，最后更新PC的值，清除相应的中断标志位。每周期指令执行的过程如下代码所示：



在80c32单片机中，处理器进入中断服务程序是通过一个硬件LCALL操作来实现的，但该硬件LCALL操作可能会被三种情况阻塞：一是已经有相同优先级或更高优先级的中断服务程序正在执行；二是当前周期并不是一条指令执行的最后一个周期，也就是说当前周期有指令正在执行；最后，如果当前执行的指令是RETI，或者任意对IE或IP寄存器进行的写操作，那么硬件LCALL需要在这些操作完成，并额外执行一条指令之后，才能被执行。这一点，在模拟器的执行过程中已得到处理。

## 2.3 外部接口

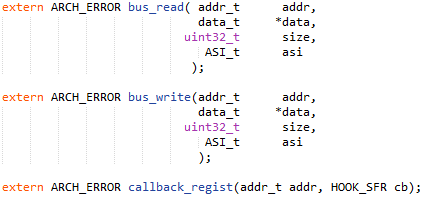
在80c32仿真平台中，处理器与外设之间的通信主要通过两种方式，直接函数调用和回调。在1.3节中对这两种仿真通信的方式已进行了详细的介绍。对于存储器映射总线的地址空间，一般通过直接函数调用的方式来发起总线传输；而对于特殊地址空间，或外设对与处理器之间的通信有特殊要求的情况下，则多采用回调的方式。由于80c32单片机特殊的存储组织结构，处理器与外设之间的通信方式被组织如下：

对片上外设资源如定时器、串口等，由于其控制寄存器在SFR空间中，且分布不规律，一个寄存器可能控制多个外设，所以在仿真平台中采用回调的方式处理；处理器在访问SFR中的控制寄存器时，将触发对应的回调函数执行；

对片外外设资源如ROM、RAM等，由于其支持的操作单一，在仿真平台中处理器采用直接函数调用的方式来与之通信；

对片外其他外设资源，用户在建模的过程中可以选择任意一种方式来实现处理器与外设之间的通信。

系统提供了三个接口来支持这两种通信方式。其声明如下：



其中bus\_read和bus\_write为直接调用接口。参数addr表示被访问的地址，data指向将被读/写的数据，size为访问数据的大小，以字节为单位，asi表示访问的地址空间。因为在80c32单片机中，片外数据存储器和程序存储器的地址空间是重叠的，需要这样一个变量来进行区分。在建模程序存储器ROM时，只需要实现这两个函数就可以了。如果系统中有多个外设都是通过直接函数调用的方式被处理器访问，那么就需要一个总线模块来实现这两个接口函数，用于转发处理器访问到对应的外设中。

在回调方式中，处理器中维护着一个回调地址表，表中每项都对应于一个外设的控制寄存器。处理器在访问这些外设控制寄存器时，将触发地址表中对应函数的执行。这些回调函数在外设模块中实现，通过callback\_regist接口将该函数的地址注册到处理器中的回调地址表中。回调函数的实现一般是将寄存器中的值转存到外设的本地变量中，然后对其进行解析，触发相应的外设操作。该函数的实现将在第三章被详细叙述。

## 2.4 调试接口

在80c32系统仿真平台中，处理器还支持调试功能。它向外提供了一个调试接口，可用于GDB远程调试。GDB对应用进行调试可分为两种情况，一种是远程调试，一种是本地调试。但GDB是一个与系统无关的跨平台调试器，它是基于ptrace来实现的，ptrace实现了底层的调试原语。如图2-5所示，传统gdb运行在调试机上，gdbserver运行在目标机上，两者之间的通信数据格式由gdb远程串行协议规定，gdbserver的启动方式相当于运行并调试一个新创建的进程。ptrace 是一个系统调用，它提供了一种方法，使得父进程可以观察和控制其它进程的执行，检查和改变其核心映像以及寄存器。

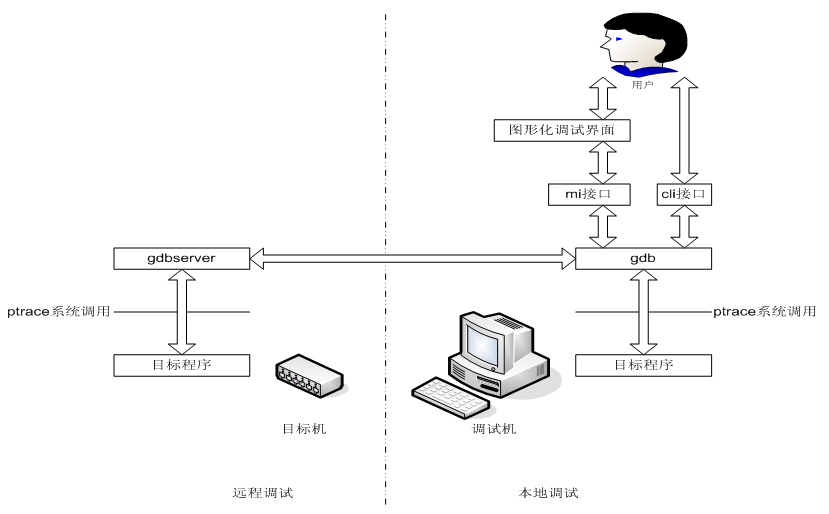
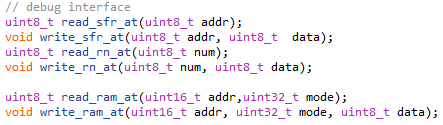
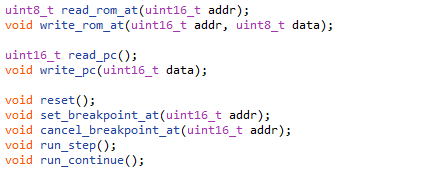


图2-5 GDB调试框架

在本项目中，调试机在兼任调试任务的同时，也是仿真CPU（目标机）的宿主机。传统意义上的GDB调试器是直接对调试机的控制器进行操作，而在此则是对仿真CPU（目标机）的控制器进行操作。为了支持GDB调试，最主要的工作是在开源GDB源码的基础上，增加调试器与仿真CPU代码之间的接口。在80c32仿真平台中，处理器提供了一个与调试器进行通信的接口，其声明如下。具体实现可参考代码，在此不再详述。



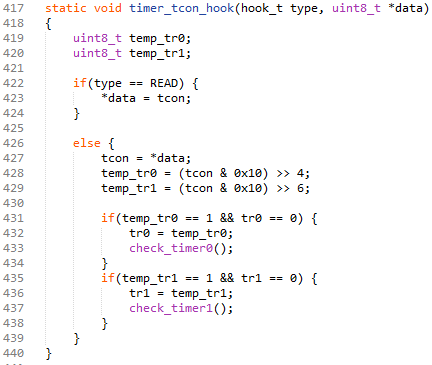


# 第三章 80c32片上外设资源模型

## 3.1 定时器0/1

80c32有三个16位的定时/计数器，它们均可以通过配置来选择运行在定时还是在计数状态。其中定时器0和定时器1均有四种运行模式，二者除了在第四种运行模式上不同之外，其他模式下的操作均相同。在模式0中，定时器0/1被配置成了13位的定时/计数器；模式1对应16位定时/计数器；在模式2中，定时器0/1被配置成了8位自动装载的定时/计数器，一旦TLx计数溢出，就将THx寄存器中的值装入TLx中（x的值为0或1）；在模式3中，定时器0被配置成了两个独立的8位计数器，TL0计数由定时器0来进行控制，TH0计数由定时器1来进行控制。定时器1在模式3中仅保持其计数值不变。

用来控制定时器0/1的寄存器有两个，分别为TMOD和TCON。其中TMOD用来设置定时器0/1的运行模式；TCON用来控制定时器0/1的运行与停止，同时存储着定时器的溢出标志，用来产生中断请求。这两个寄存器均在SFR地址空间中，所以在实现定时器模块时，需要分别为其编写回调函数。这里给出TCON的回调函数，代码如下所示：

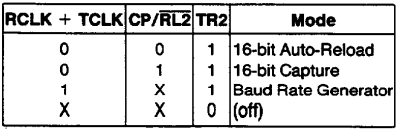


在该回调函数中，如果处理器读取该寄存器中的值，则将定时器本地tcon变量中的值返回，同时更新处理器中TCON寄存器的值。如果处理器将一个新值写入到该寄存器，那么在更新了TCON寄存器之后，将进一步更新定时器的本地变量。如果定时器的开始位被置一，则执行定时器的主功能函数check\_timer。在定时器的功能函数中，将根据TMOD寄存器中的配置来判断定时器将运行在哪个模式，然后根据TLx和THx中的值，计算出定时/计数器的溢出周期，之后向UTM系统注册溢出事件。当定时器溢出时，将置位TFx，同时向处理器发起中断请求。这些操作被封装在溢出事件对应的回调函数中，并被注册到了UTM系统。当定时器溢出对应的仿真时间点到来时，UTM系统将执行该回调函数，向处理器发起中断请求。这就是整个系统的执行流程，可参考图2-4的内容。

当然，对于寄存器TLx和THx，也需要向处理器注册回调函数，以保证定时器中寄存器的本地备份与处理器中寄存器的值保持一致。具体的实现可参考源代码。

## 3.2 定时器2

定时器2有三种操作模式，分别为16位自动装载模式，16位捕获模式和波特率产生器模式。三种模式的选择信号如下表所示。



其中，在16位自动装载模式下，如果T2CON寄存器中的EXEN2标志位为0，那么当定时器溢出时，将设置TF2标志位为1，同时将捕获寄存器RCAP2L和RCAP2H中的值自动装载到TL2和TH2寄存器中；如果EXEN2标志位为1，那么当定时器溢出时，上述操作照常进行。只是在任何时候，T2EX管脚上的下降沿输入也将触发上述自动装载操作，同时设置EXF2表示位。16位捕获模式下的操作与自动装载模式下的操作正好相反，具体内容可参考相关文档。

在波特率产生器模式，定时器2的操作与常规模式有几点不同。首先是定时器的计数对象是2个时钟周期，而不是一个机器周期。定时器2的操作与自动装载模式下的操作很相似，只是在定时器2溢出时，并不设置TF2标志位，也就是不产生中断。如果EXEN2标志位为1，那么在T2EX管脚上的下降沿输入将触发定时器2设置EXF2标志位，但不会触发自动状态操作。

以上给出了定时器2在不同操作模式下的硬件执行特性。在用代码模拟定时器2的功能时，其框架与模拟定时器0/1时一样，因此这里不再详细描述，具体实现细节请参考代码。

## 3.3 串行通信接口

串口有四种操作模式。在mode0中，传送8位数据，数据通过RXD管脚接收和发送，其波特率为机器周期对应的频率，TXD管脚输出偏移时钟；在其他模式中，数据均是通过RXD管脚接收，TXD管脚发送。其中，在mode1中，总共传送10位数据，包括起始位、8位有效数据和一个结束位。在接收时，结束位放入SCON寄存器中的RB8位。在该模式下，波特率为定时器的溢出频率，是可变的。在mode2中，总共传输11位数据，包括起始位、8位有效数据、可编程位和一个结束位。发送时，可编程位中存放SCON寄存器中TB8位的值，可被设置为0或1；接收时，可编程位进入RB8，结束位被忽略。在该模式下，波特率为32或64分频时钟。串口在mode3下的操作与在mode2下的操作是一致的，唯一的区别在于mode3下的波特率为定时器溢出频率，是可变的。

在所有四种模式中，一旦有任何指令将SBUF寄存器作为目的寄存器，就将触发发送操作；接收的触发条件如下：在模式0中，RI为0且REN = 1；在其他模式中，REN = 1同时接收到开始位。

当串口工作在模式2或模式3时，可用于多处理器通信。首先是硬件的特点：如果SCON寄存器中的SM2位为1，那么只有当接收到的RB8为1时才触发中断；在mode0中，SM2没有作用；在mode1中，如果SM2为1，那么如果没有接收到有效的停止位，就不触发中断。那么两个处理器在通过串口进行通信时，就可以通过设置可编程位的值，同时利用中断触发来进行；主机先发一个地址，可编程位置1；从机均将SM2置1，那么这时候所有的从机在收到后，都会触发中断，但只有访问的地址与自己匹配的从机，才在中断中将自己的SM2位置0；主机在发送地址之后，会紧接着发送数据，但这时候的可编程位为0；所以，只有从机的SM2为0，才能触发中断；也就是之前地址选中的那个从机。

对于mode0和mode2，串口的波特率固定，在mode2中的两种情况可由SMOD位来进行选择。对于mode1和mode3，串口可以由timer1或timer2的溢出来提供波特率。其波特率来源如图3-1所示：



图3-1 串口操作时钟的来源

如果是timer1，那么可以配置成8位的自动加载模式，将中断使能关掉；也可以配置成16位的timer，将中断打开，然后在中断服务程序中，重新设置计数值，以实现较大的波特率。

这里详细介绍了串口的硬件特性，其仿真模型的框架和定时器是一致的，只要将此处的硬件特性全部实现就可以。具体实现内容请参考源代码。

# 第四章 验证

## 4.1 测试程序

本章主要内容是评估设计中51架构下指令集模拟器的指令执行效率，同时比较模拟器在虚拟硬件下运行的不同操作系统平台上的执行性能与宿主机上得到的结果的差异。对于80c32单片机仿真平台的性能测试结果，可参考介绍UTM的文档，此处不再赘述。由于51处理器没有完善的测试程序集，所以这里的测试程序由设计人员自己编写。为了尽量覆盖51架构下的所有指令，测试程序用汇编语言编写而成，覆盖了指令集中的大部分指令。详细的测试代码可参考工程中的文件。

## 4.2 测试环境

51指令集模拟器的执行环境在这里分为两类，一类是宿主机，另一类是在宿主机上运行的虚拟机环境。宿主机的型号是ThinkServer140，处理器频率3.3GHz，4个处理器核心，内存大小4G，磁盘大小1T，操作系统为64位的Windows7 ultimate SP1（在表中用win7x64sp1来表示）。虚拟机搭建平台采用的是VMware，在宿主机上执行，其版本号为Vmware.Workstation.v11.0.0。虚拟出的硬件平台及其上执行的操作系统可归纳如表4-1所示。

表4-1 虚拟测试环境

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编号 | 操作系统 | CPU数量 | 内存 | 磁盘 |
| #1 | Winxpsp2 | 1 | 512M | 40G |
| #2 | Winxpsp3 | 1 | 512M | 40G |
| #3 | Win7x86sp1 | 1 | 1G | 60G |
| #4 | Debian7.7.0-amd64 | 1 | 512M | 20G |

在windows平台下，模拟器的可执行代码是通过Visual Studio2013编译，以静态链接方式产生的release版本；在Linux平台下则是通过GCC编译优化，静态链接标准C库文件产生的模拟器可执行文件，GCC的版本号为4.7.2。

## 4.3 测试结果

将测试程序加载到51模拟器的内部程序存储器中，然后从地址0处开始执行。模拟器每执行10M条指令，统计一次执行的时间。这里给出模拟器执行300M条指令的统计结果。

模拟器在不同平台下的指令执行时间如表4-2所示。由表中的统计结果可得到如图4-1的模拟器在不同平台下的性能曲线，图4-2是该性能曲线图在数据集中区域放大的结果。由图4-1可以看出，该模拟器的性能大约为110MIPS，不同平台比较的结果大致为Winxpsp2<Winxpsp3<Win7x86sp1<Win7x64sp1（host）<debian7.7.0。也就是说，在Windows平台下，该模拟器在宿主机上的执行速度略高于在虚拟机中的执行速度，而在Linux平台下的执行速度明显高于Windows平台，即使是宿主机上的Windows平台。

表4-2 51CPU核在不同平台下的执行时间

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **指令数/M** | **不同平台下的执行时间/ms** | | | | | **不同平台下的性能/MIPS** | | | | |
| **xpsp2** | **xpsp3** | **win7x86sp1** | **win7x64sp1** | **debian7.7.0** | **xpsp2** | **xpsp3** | **win7x86sp1** | **win7x64sp1** | **debian7.7.0** |
| 10 | 171 | 125 | 93 | 124 | 110 | 58.48 | 80.00 | 107.53 | 80.65 | 90.91 |
| 20 | 265 | 218 | 171 | 202 | 190 | 75.47 | 91.74 | 116.96 | 99.01 | 105.26 |
| 30 | 359 | 296 | 265 | 296 | 270 | 83.57 | 101.35 | 113.21 | 101.35 | 111.11 |
| 40 | 453 | 390 | 358 | 374 | 350 | 88.30 | 102.56 | 111.73 | 106.95 | 114.29 |
| 50 | 546 | 484 | 452 | 468 | 420 | 91.58 | 103.31 | 110.62 | 106.84 | 119.05 |
| 60 | 640 | 578 | 530 | 546 | 500 | 93.75 | 103.81 | 113.21 | 109.89 | 120.00 |
| 70 | 734 | 656 | 624 | 639 | 580 | 95.37 | 106.71 | 112.18 | 109.55 | 120.69 |
| 80 | 828 | 750 | 717 | 733 | 660 | 96.62 | 106.67 | 111.58 | 109.14 | 121.21 |
| 90 | 921 | 843 | 811 | 811 | 740 | 97.72 | 106.76 | 110.97 | 110.97 | 121.62 |
| 100 | 1000 | 921 | 904 | 904 | 820 | 100.00 | 108.58 | 110.62 | 110.62 | 121.95 |
| 110 | 1093 | 1015 | 982 | 982 | 890 | 100.64 | 108.37 | 112.02 | 112.02 | 123.60 |
| 120 | 1187 | 1109 | 1076 | 1076 | 970 | 101.10 | 108.21 | 111.52 | 111.52 | 123.71 |
| 130 | 1281 | 1187 | 1170 | 1154 | 1050 | 101.48 | 109.52 | 111.11 | 112.65 | 123.81 |
| 140 | 1375 | 1281 | 1263 | 1248 | 1120 | 101.82 | 109.29 | 110.85 | 112.18 | 125.00 |
| 150 | 1453 | 1375 | 1357 | 1341 | 1200 | 103.23 | 109.09 | 110.54 | 111.86 | 125.00 |
| 160 | 1546 | 1453 | 1450 | 1419 | 1280 | 103.49 | 110.12 | 110.34 | 112.76 | 125.00 |
| 170 | 1640 | 1546 | 1544 | 1513 | 1360 | 103.66 | 109.96 | 110.10 | 112.36 | 125.00 |
| 180 | 1734 | 1640 | 1622 | 1591 | 1440 | 103.81 | 109.76 | 110.97 | 113.14 | 125.00 |
| 190 | 1812 | 1718 | 1716 | 1684 | 1520 | 104.86 | 110.59 | 110.72 | 112.83 | 125.00 |
| 200 | 1906 | 1812 | 1809 | 1762 | 1590 | 104.93 | 110.38 | 110.56 | 113.51 | 125.79 |
| 210 | 2000 | 1906 | 1903 | 1856 | 1670 | 105.00 | 110.18 | 110.35 | 113.15 | 125.75 |
| 220 | 2093 | 1984 | 2012 | 1934 | 1740 | 105.11 | 110.89 | 109.34 | 113.75 | 126.44 |
| 230 | 2171 | 2078 | 2106 | 2028 | 1820 | 105.94 | 110.68 | 109.21 | 113.41 | 126.37 |
| 240 | 2265 | 2171 | 2184 | 2121 | 1900 | 105.96 | 110.55 | 109.89 | 113.15 | 126.32 |
| 250 | 2359 | 2250 | 2262 | 2199 | 1980 | 105.98 | 111.11 | 110.52 | 113.69 | 126.26 |
| 260 | 2453 | 2343 | 2355 | 2293 | 2050 | 105.99 | 110.97 | 110.40 | 113.39 | 126.83 |
| 270 | 2531 | 2437 | 2449 | 2371 | 2130 | 106.68 | 110.79 | 110.25 | 113.88 | 126.76 |
| 280 | 2625 | 2500 | 2542 | 2464 | 2200 | 106.67 | 112.00 | 110.15 | 113.64 | 127.27 |
| 290 | 2718 | 2593 | 2636 | 2542 | 2280 | 106.70 | 111.84 | 110.02 | 114.08 | 127.19 |
| 300 | 2812 | 2687 | 2714 | 2636 | 2360 | 106.69 | 111.65 | 110.54 | 113.81 | 127.12 |

图4-1 51CPU核在不同平台下的性能

图4-2 不同平台下模拟器的性能差别