一种 DSP 的快速上下文切换机制 *

刘月吉[1]，张盛兵[1]，黄嵩人[2]
(1. 西北工业大学 计算机学院 系统结构系，西安 710072；2. 中国电子科技集团公司第五十八研究所，江苏 无锡 214035)

摘　要：针对嵌入式系统实时控制和信号处理的需求，建立了一种基于 DSP 架构的快速上下文切换机制，为实时处理提供了有力支持。机制采用两条独立的总线，分别用来传送地址和数据信息，实现地址和数据信息的并行传输，增加了上下文保存和恢复的带宽；同时应用影子寄存器与通用寄存器之间的切换，有效减少了对存储器的访问；引入对上下文的延后保存和提前恢复操作，解决了任务或中断嵌套调用时的低效问题，显著地提高了上下文切换的速度。

关键词：数字信号处理器；上下文切换；影子寄存器；中断嵌套
中图分类号：TP332　　**文献标志码**：A　　**文章编号**：1001-3695(2012)01-0203-04
doi:10.3969/j.issn.1001-3695.2012.01.057

Mechanism of fast context switching based on DSP

LIU Yue-ji[1], ZHANG Sheng-bing[1], HUANG Song-ren[2]
(1. *School of Computer Science & Engineering, Northwestern Polytechnical University, Xi'an* 710072, *China*; 2. *China Electronics Technology Group Corporation No.*58 *Research Institute, Wuxi Jiangsu* 214035, *China*)

Abstract: This paper presented a novel mechanism of fast context switching based on DSP is presented according to the requirements of the real time controlling and signal processing in embedded systems, which provides a strong support for real-time processing. First, the mechanism involved two independent bus, dedicated for address and data information separately, that is to say, data and address information could be transmitted in parallel, which could expand the bandwidth for context saving and restoring; Furthermore, switching between shadow registers and general registers was adopted in order to reduce the amount of memory accessing; What's more, the method of post-saving and pre-restoring the context was used, with which the efficiency of the task or interrupt nesting was enhanced and the speed of context switching was confirmed to be improved significantly.

Key words: DSP; context switching; shadow registers; interrupt nesting

0　引言

传统嵌入式应用中通常采用 MCU(微控制器)和 DSP(数字信号处理器)分别执行控制功能与信号处理算法，当开发包括信号处理与控制功能时，要求两种算法间实现互操作性，而单独的 MCU 或者 DSP 都不能很好地解决这样的问题[1]，于是越来越多的 MCU 和 DSP 集成方案得到广泛应用[2]。

本文论述了一种基于 DSP 架构的上下文切换机制，在满足 DSP 运算需求的同时，实现了快速的任务上下文切换，为嵌入式系统的实时控制提供有力支持。上下文切换是指处理器的控制权由运行任务转移到另外一个就绪任务时所发生的事件。上下文切换是在嵌入式实时系统中频繁发生的动作，其时间的快慢直接影响到整个系统的实时性能。提高上下文切换速度的方法有很多，如增加寄存器有效位，有选择地保存或恢复上下文，从而提高切换速度[3,4]；采用比硬盘访存速度更快的闪存，并精心设计以凸显闪存的速度优势[5]；使用影子寄存器，避免上下文的保存和恢复[6]。但如上的方法都有所不足：文献[3,4]复杂度很高，文献[5]资源成本很高，文献[6]在中断嵌套时效率很低。本文从 DSP 的体系结构和上下文的保存或恢复机制入手，采用简单的调度机制，添加少量的寄存器，实现了快速有效的上下文切换。

1　任务的上下文

上下文是指与任务相关的数据信息，它是处理器用来确定相关任务的状态并使其继续执行(如果任务被中断)的所有信息，包括任务用到的寄存器、程序指针和程序状态信息。

本文的 DSP 采用哈佛结构，包含 32 个 32 bit 通用寄存器，按功能分为数据寄存器(D0 ~ D15)和地址寄存器(A0 ~ A15)两组，每组的数据接口宽度为 64 bit。其中，A0 ~ A7/D0 ~ D7 为低寄存器组，A8 ~ A15/D8 ~ D15 定义为高寄存器组。A10 是栈指针寄存器(stack pointer register)；A11 是返回地址寄存器(return address register)。A0、A1、A8、A9 为全局寄存器，在程序调用或中断时这些寄存器不作为任务上下文保存或恢复。此外，还有两组影子寄存器：数据影子寄存器(TD8 ~ TD15)，地址影子寄存器(TA10 ~ TA15)，作为高寄存器组的备份，用于快速地切换任务的上下文。

收稿日期：2011-05-23；**修回日期**：2011-07-01　　**基金项目**：国家"核高基"重大专项基金资助项目(2009ZX01034-001-002-003)
作者简介：刘月吉(1986-)，女，河北清苑人，硕士研究生，主要研究方向为微处理器设计(liuyueji@mail.nwpu.edu.cn)；张盛兵(1968-)，男，教授，博导，主要研究方向为 VLSI 设计、计算机体系结构；黄嵩人(1972-)，男，教授级高级工程师，主要研究方向为数字集成电路设计.

上下文涉及到的系统寄存器有程序指针 PC(program counter)、程序状态 PSW(program status word)、已用上下文信息 PCXI(previous context information)、空闲上下文信息 FCXI(free context information)。

上下文可进一步分为上文和下文,上文由 A10 ~ A15、D8 ~ D15以及系统寄存器 PCXI 和 PSW 构成,下文由 A2 ~ A7、D0 ~ D7 以及 A11(返回地址)和系统寄存器 PCXI 构成,其中 PCXI 同时作为上文或下文的链接字(link word)。任务切换时对上文和下文的处理是不同的:下文的寄存器 A2 ~ A7,D0 ~ D7 与全局寄存器类似,中断、陷阱和程序调用时对下文的修改,在中断、陷阱和程序调用结束后仍然有效,因此可以用下文的寄存器来传递参数;而上文并不如此,它相当于任务私有的资源,在任务切换时必须保存或者恢复。

在中断、陷阱或者调用程序的任务中,处理器禁止对 A10、A11、PSW、PCXI 涉及到任务状态的寄存器(A10、A11、PSW、PCXI)进行修改,否则会造成不可知的错误。

由上述上文或下文组成的 16 字的存储空间称为 CSA(context save areas),采用 link word 将多个 CSA 连接起来就可以实现任务切换的嵌套,如图 1 所示。另外在上下文保存或恢复时,为了避免频繁地访问存储器获得 link word,增加了系统寄存器 PCXB(PCX backup),用于上下文保存或恢复的 link word。

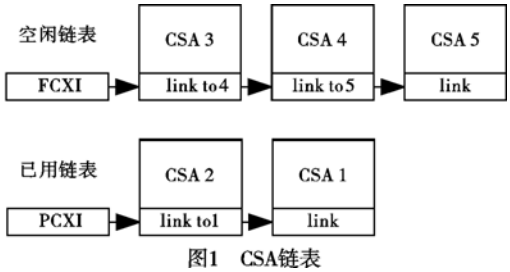

图1 CSA链表

2 上下文指令

上下文指令都是成对出现的,称其为互补指令,如表 1 所示。

表 1 上下文指令

事件或指令	操作	互补指令
中断	保存上文	RFE
陷阱	保存上文	RFE
CALL(function call)	保存上文	RET
BISR(begin interrupt service routine)	保存上文	RSLC
SVLC(save lower context)	保存上文	RSLC
STLC(store lower context)	存储下文	LDLC
STUC(store upper context)	存储上文	LDUC
RFE(return from exception)	恢复上文	中断、陷阱
RET(return from call)	恢复上文	CALL
RSLC(restore lower context)	恢复上文	BISR、SVLC
LDLC(load lower context)	装载下文	STLC
LDUC(load upper context)	装载上文	STUC

当有中断、陷阱或程序调用(CALL)或返回(RFE、RET)时,系统会自动地保存或恢复上文,而如果需要保存或恢复下文,可以使用 SVLC、RSLC 指令显式地对下文进行保存或恢复的操作。

BISR 指令对上下文的操作和 SVLC 相同,此外,它还使能中断系统,并初始化 DSP 内核的优先序号。

STLC、STUC、LDLC 和 LDUC 与其他指令不同,对 FCXI、PCXI、PSW 等系统寄存器不作修改,即不改变 CSA 链表。

数据同步指令(DSYNC)用来同步对存储器的访问,要求完成在它之前的所有访存操作。

3 本地数据存储器和 CACHE 的结构

本地数据存储器和 cache 共用同一个块 SRAM,它由四个存储体组成,一行是 128 bit,每个存储体一行为 32 bit,每行分为高半字和低半字两部分,并且不是连续编址,按字节编址方式如图 2 所示。本地数据存储器中保存上文的组织方式如图 3 所示(图中的下标 h 或 l 分别表示 32 bit 数据的高半部分或低半部分),下文的组织方式类似。

bank3			bank2			bank1			bank0		
校验	16	6	校验	14	4	校验	12	2	校验	0	10
	1e	e		1c	c		1a	a		18	8
	36	26		34	24		32	22		30	20
	3e	2e		3c	2c		3a	2a		38	28

图 2 存储器结构

bank3			bank2			bank1			bank0		
校验	$D9_h$	PSW_h	校验	$D9_l$	PSW_l	校验	$D8_h$	$PCXI_h$	校验	$D8_l$	$PCXI_l$
	$D11_h$	$A11_h$		$D11_l$	$A11_l$		$D10_h$	$A10_h$		$D10_l$	$A10_l$
	$D13_h$	$A13_h$		$D13_l$	$A13_l$		$D12_h$	$A12_h$		$D12_l$	$A12_l$
	$D15_h$	$A15_h$		$D15_l$	$A15_l$		$D14_h$	$A14_h$		$D14_l$	$A14_l$

图 3 上文在存储器中的组织方式

存储器与 DSP 内核之间的数据交换的最大宽度为 128 bit,包括两条独立的总线,每条总线为 64 bit,分别用来传送地址和数据信息。如保存上文时,将地址寄存器的 A11、A10 和数据寄存器的 D11、D10 同时送入本地存储器的一行,那么一个周期就可以完成 128 bit 的数据传输。

采取这样的编址、上下文存储以及数据接口模式可以提高上下文切换的速度。由于数据和地址信息的并行传输,可以让上下文的保存或恢复获得 128 bit 的最大数据宽度,与 128 bit 的存储器行相对应。

采用非连续的编址方式,可以得到地址连续的 128 bit 的数据或地址(如 D12 ~ D14),并且很容易实现两路组相连的 cache。

如上所述,如果 CSA 空间在本地数据存储器范围内,要保存或恢复一个完整的上文(16-word)至少需要四个周期。如果 CSA 空间不在本地存储器范围内,涉及到 cache 的处理机制,情况会变得复杂,因此以本地存储器的访问说明此机制。

4 上下文控制

4.1 上下文控制的结构

通过上下文指令的执行,在寄存器和存储器间进行数据传送,实现任务的切换。整个过程都依赖上下文的控制。上下文的控制由译码器、状态机和控制器三部分组成。

译码器根据上下文指令进行译码得到上下文指令类型等信息。如 cxt_inst_type[8:0],它采用的是 one-hot 编码,8 ~ 0 依次代表如下指令:DSYNC、RSLC、SVLC/BISR、STUC、STLC、LDUC、LDLC、CALL、RET/RFE。状态机根据当前状态和上下文指令进行判断,决定要执行的上下文指令序列(2 ~ 8 周期不等)并进行状态转换。控制器根据状态机和译码器的输出产生数据通路的控制信号,由于指令需要多周期完成,有些控制

信号需要寄存,以保证时序的正确性。

4.2　上下文状态机的控制

上下文指令的执行主要是依赖状态机进行控制的。状态机的结构如图 4 所示。

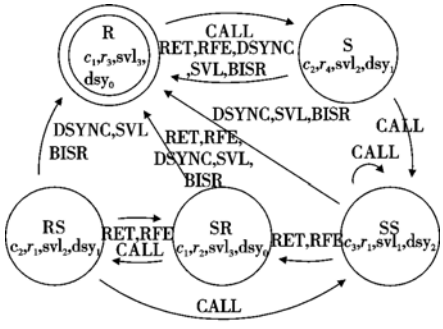

图4　上下文状态机

状态机包含五个状态(R,S,SS,SR,RS),分别表示不同的 CSA 存储状态。其中,R 为初始状态,表示上下文都已保存或恢复好的状态;S 表示上文只保存了一半的状态;SS 表示上文未保存的状态;SR 表示上文只恢复了一半的状态;RS 表示上文只保存了一半的状态(区别于 S 的是,此状态时 CSA 链表中有多个 CSA,恢复时可调用 r_1 序列,而不是 r_4 序列)。状态机在指令结束后,按照指令的类型进行状态转换。如在 R 状态下执行 CALL 指令,CALL 指令结束后,状态机转换到 S 状态;若为其他指令,不满足转换条件,状态机保持原状态。

不同的指令有不同的执行序列, CALL 指令有三种执行序列(分别用 c_1、c_2、c_3 表示),RET 或者 RFE 指令有四种执行序列(分别用 r_1、r_2、r_3、r_4 表示),SVLC 或者 BISR 指令有三种执行序列(分别用 svl_1、svl_2、svl_3 表示),DSYNC 指令有三种执行序列(分别用 dsy_0、dsy_1、dsy_2 表示)。与如上指令不同,RSLC、STLC、LDLC 等指令只有一种执行序列,它们的执行不依赖并且不影响状态机的状态。

各指令序列的具体操作如表 2 所示,从表 2 可以看出,有些指令序列一次保存或恢复整个上下文(如 c_3、r_3),有的只保存和恢复了一半的上下文(如 c_1、r_1)。

表 2　指令序列的操作

指令序列	周期	操作
c_1	2	[8,10]→M{FCXI},调整 CSA 链表,切换高寄存器组。将返回地址保存到 A11。先切换写寄存器组,等待 st 读完数据切换读寄存器。
c_2	2	切换高寄存器组,[12,14]→M{PCXI},调整 CSA 链表。将返回地址保存到 A11。切换读写寄存器组,让 st 读取切换后的寄存器数据。
c_3	4	切换高寄存器组,[8,10,12,14]→M{PCXI},调整 CSA 链表。将返回地址保存到 A11。切换读写寄存器组,让 st 读取切换后的寄存器数据。
r_1	2	M{PCXB}→[8,10],调整 CSA 链表,切换高寄存器组。将 A11 赋给 PC。先切换读寄存器组,等待 ld 写完数据切换再写寄存器组。
r_2	2	切换高寄存器组,M{PCXI}→[12,14],调整 CSA 链表。将 A11 赋给 PC。切换读写寄存器组,让 ld 的数据写入切换后的寄存器组。
r_3	4	切换高寄存器组,M{PCXI}→[8,10,12,14],调整 CSA 链表。将 A11 赋给 PC。切换读写寄存器组,让 ld 的数据写入切换后的寄存器组。
r_4	1	调整 CSA 链表,切换高寄存器组。将 A11(返回地址)赋给 PC。切换读写寄存器组,无须 ld 数据。

续表 2

指令序列	周期	操作
svl_1	8	[8,10,12,14]→M{PCXI},[0,2,4,6]→M{FCXI},调整 CSA 链表。上下文 st 的寄存器不同,需两次切换读寄存器,第一次切换后保存上文,保存上文完毕后进行第二次切换,再保存下文,svl 前后的高寄存器组不变。
svl_2	6	[12,14]→M{PCXI},[0,2,4,6]→M{FCXI},调整 CSA 链表。上下文 st 的寄存器不同,需两次切换读寄存器,第一次切换后保存半个上文,保存上文完毕后进行第二次切换,再保存下文,svl 前后的高寄存器组不变。
svl_3	4	[0,2,4,6]→M{FCXI},调整 CSA 链表。无须切换寄存器
dsy_0	1	dsy_0 没有实质的执行序列,相当于空操作。
dsy_1	2	[12,14]→M{PCXI}。切换读寄存器组,保存剩下的半个上文,再次切换读寄存器组, dsy 前后的高寄存器组不变
dsy_2	4	[8,10,12,14]→M{PCXI}。切换读寄存器组,保存上文,再次切换读寄存器组, dsy 前后的高寄存器组不变。
RSLC	4	M{PCXI}→[0,2,4,6],调整 CSA 链表。无须切换寄存器组。

注:[ ]里面 0 表示上文中的 D1、A11、D0、PCXI;2 表示上文中的 D3、A3、D2、A2;4 表示上文中的 D5、A5、D4、A4;6 表示上文中的 D7、A7、D6、A6;8 表示上文中的 D9、PSW、D8、PCXI;10 表示上文中的 D11、A11、D10、A10;12 表示上文中的 D13、A13、D12、A12;14 表示上文中的 D15、A15、D14、A14;→表示存储器和寄存器的数据传递,如[8,10]→M{FCXI}为 st,M{PCXI}→[12,14]为 ld。

SVLC、BISR、DSYNC 这些指令,都要求将未完成的上下文保存或恢复执行完毕,故执行完这些指令后,状态机恢复初始的 R 状态。

4.3　程序实例

本节以一段程序实例进一步说明上下文切换的细节。表 3 给出了一个实例程序运行时系统的状态变化,初始的空闲 CSA 链表为 FCXI→M{100}→M{200}→M{300}→M{400}→M{0},已用 CSA 链表为空。

表 3　示例程序以及系统状态变化

程序	状态	G/T	PCXB	PCXI	FCXI	已用 CSA 链表
初始	R	G	0	0	100	空
CALL(c_1)	S	T	0	100	200	{100|G *}
CALL(c_2)	SS	G	100	200	300	{200| }→{100|G}
CALL(c_3)	SS	T	200	300	400	{300| }→{200|T}→{100|G}
RET(r_1)	SR	G	200	200	300	{200|T ↑}→{100|G}
CALL(c_1)	RS	T	200	300	400	{300|G *}→{200|T}→{100|G}
RET(r_1)	SR	G	200	200	300	{200|T ↑}→{100|G}
RET(r_2)	R	T	200	100	200	{100|G}
RET(r_3)	R	G	200	0	100	空

注:G/T 表示当前使用的寄存器组,G 表示通用寄存器,T 表示影子寄存器;{}表示一个 CSA 存储区域;|左侧为地址,右侧为存储数据所在的寄存器组;" "表示还没有保存数据;"*"表示此 CSA 只保存了一半的上文(对应 S 或 RS 状态); 表示此 CSA 已经恢复了一半上文(对应 SR 状态)。

从表 2 和表 3 的描述可以看出,当有程序的嵌套调用时,上下文操作如下:

第一层程序调用(c_1)时,先保存当前寄存器(即通用寄存器)的一半上文,然后再切换读寄存器组到影子寄存器,而写

寄存器组一开始就切换到影子寄存器,用来保存返回地址。CSA 链表的状态转变为{100|G*},状态机进入 S 状态。这次程序调用的上下文切换只需要两个周期,比传统的保存上文节省了两个周期。

第二层程序调用(c_2)时会保存剩下的一半上文,先切换读寄存器组到通用寄存器组,才能保证上次剩余的通用寄存器的另一半上文被正确保存,同样地,返回地址的保存也需要先切换写寄存器到通用寄存器组。CSA 链表增加了一个 CSA,变为{200| }→{100|G},状态机进入 SS 状态。可以看到,表头的 CSA 还没有保存数据,这是因为影子寄存器组为数据进行了缓冲,如果没有更深一层的程序调用,那么两层嵌套的上下文切换只需要四个周期,而不是保存两个上文所需要的八个周期。

而第三层程序调用(c_3)就必须一次保存整个上文。但是要保存的上文是上一层调用时的现场,即上一层 CSA 表头还没有保存的影子寄存器的数据,所以要先切换读写寄存器组,再进行数据和返回地址的保存。此时 CSA 链表为{300| }→{200|T}→{100|G}。很显然,当前 CSA 链表中 CSA 的个数就是当前程序所在的调用层数。

子程序返回与程序调用相反,在 RS 和 SS 状态时会恢复 CSA 链表第二个 CSA(即 PCXB 指向的空间)的一半(r_1);进入 SR 后,再恢复剩余的一半(r_2);在 R 状态下,则需要对整个上文进行恢复(r_3);在 S 状态下,CSA 链表可能只有一个 CSA,此时无须恢复上文,直接切换寄存器修改 CSA 链表即可(即 r_4)。

4.4　效率分析

此机制增加了 14 个 32 bit 的影子寄存器,并且利用通用寄存器和影子寄存器之间的切换,设置寄存器组的读切换标志和写切换标志,避免了上下文保存或恢复进行的存储器与寄存器间的数据移动。采用延后保存和提前恢复的方法,即任务调度后,延后保存半个上下文(再次调度时执行 c_2 序列);在任务返回前,又提前恢复半个上下文(执行 r_1 序列时,恢复 PCXB 所指的半个上下文),解决了利用影子寄存器任务或中断嵌套效率低的问题[6],同时提高了上下文切换的速度。

在小于三层的程序嵌套调度中,上下文每次切换只需两个周期;在大于等于三层的程序嵌套调度中,上下文切换需要四个周期,而原来每次上下文切换都需要四个周期。采用了这种机制后的上下文切换时间如式(1)所示。其中,$0 < q < 1$,为低于三层的任务切换占所有切换的比重。

$$\frac{(1-q)\times 4 + q\times 2}{4} = 1 - \frac{q}{2} \tag{1}$$

显然,低于三层的嵌套任务或中断(即 $q=1$),上下文切换速度提高了一倍,时间为原来的 50%。在包含三层及三层以上的嵌套任务切换时,q 越大,效率提升越高,上下文切换所用时间越短。由于深度任务或中断嵌套的任务调度并不常用,即 q 很大,所以速度的提升是很可观的。根据文献[6]的实验数据,5 min 内的中断嵌套率为 0.068%,即中断嵌套层数大于等于 2 占所有中断的比重,因此三层及以上的中断嵌套比重 $(1-q)$ 肯定小于 0.068%,那么,$q > 99.932\%$。若取 q 为 99.932% 时,那么上下文切换时间为原来的 50.034%,平均的上下文切换时间只有 2.00136 个周期。

5　结束语

本文提出了一种基于 DSP 架构的上下文切换机制,该机制有助于在处理密集型信号、复杂的数学函数以及影像的同时,增强其中断响应、任务切换的控制,为实时控制提供了有力支持。其主要优势体现为:a)根据 DSP 的体系结构,采用了两条独立的数据总线,同时对地址和数据信息进行操作,将上下文保存和恢复的带宽提高了一倍;b)将下文作为任务间的共用资源,有效地减少了任务切换需要保存的内容,同时也减少了影子寄存器的硬件资源开销;c)丰富的上下文指令为各种任务切换提供了灵活的选择,如支持多种中断方式;d)有效的状态机调度算法,为中断或程序嵌套提供了有力支持;e)对时间开销的分析可以看出任务切换的速度提升是显著的。

本文的上下文切换机制也存在需要改进的地方:a)虽然两条独立的数据总线提高了存储器和 DSP 之间数据交换的带宽,但是同时使用两条数据总线的情况只在上下文切换时发生,数据总线资源的利用率很低;b)当状态机处于 S 状态时,上文只保存了一半,并且要执行 RET 指令,那么可以根据此时的 CSA 链表是否只有一个 CSA 来决定要执行的 RET 序列,而不是一律执行 r_4 序列,若 CSA 链表不只一个 CSA,显然执行 r_1 序列更有利于提高上下文恢复的速度。然而,提高数据总线利用率、改进状态机会使控制变得复杂,状态机要增加新的状态和转换规则,这需要在复杂度和切换效率间作一个权衡,这将是进一步研究工作的重点。

参考文献:

[1] MARTIN D, OWEN R E. A RISC architecture with uncompromised digital signal processing and microcontroller operation, acoustics, speech and signal processing[C]//Proc of IEEE International Conference. on Acoustics, Speech and Signal Processing. 1998: 3097-3100.

[2] HUANG Chung-wen, HSIEH Kun-yuan, LI Jia-jhe, *et al*. Support of paged register files for improving context switching on embedded processors[C]//Proc of Inernational Conference on Computational Science and Engineering. Washington DC: IEEE Computer Society, 2009: 352-357.

[3] GYGER A C, MASGONTY S, MORGAN J M, *et al*. Low-power 32-bit Dual-MAC 120uW/MHZ 1.0V icyflex DSP/MCU core[C]//Proc of the 34th European Solid-State Circuits Conference. 2008: 190-193.

[4] ZHOU Xiang-rong, PETROV P. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications[C]//Proc of the 43rd Annual Conference on Design Automation. New York: ACM Press, 2006: 352-357.

[5] LEE S, HYUN S, KOH K. Selective context switching scheme on flash memory[C]//Proc of International Conference on Computational Science and Its Applications. 2009: 89-94.

[6] 孙康, 沈海斌, 王继民, 等. 基于映像寄存器构建的实时操作系统内核[J]. 清华大学学报: 自然科学版, 2007, 47(S2): 1899-1902.