XA 用户手册

7 外部总线

多数XA器件支持通过外部总线对外部代码和/或数据进行操作.外部总线给要操作的器件提供地址信息,然后产生操作需要的信号,并在数据总线上输入或输出数据.典型的总线操作为代码读,数据读,数据写.XA为外部总线提供灵活简单的连接,并能优化代码的获取.
下面的论述基于标准的XA外部总线,一些特殊的XA器件可能有不同的外部总线性能,或者没有外部总线.

7.1 外部总线信号

为了使用上的灵活,标准的XA总线支持8或16-bit的数据传输和用户可选择的地址线数目.最大的地址线数目同器件有关,但最大为24.下面论述XA总线的标准控制信号.

7.1.1 PSEN – 程序存储器启动信号

程序存储器启动信号,用来启动外部程序存储器,例如EPROM.它是低电平有效信号,一般接在外部EPROM的输出使能脚.当不执行代码读时,PSEN保持高电平.

7.1.2 RD – 读信号

总线读信号,低电平有效.当外部进行数据读操作时该控制信号有效.RD一般接到外部外设器件相同名字的管脚上.

7.1.3 WRL – 写信号低字节

WRL是外部总线数据读控制信号.一般接在外部器件的WR脚上.当外部器件使用在16-bit模式时,这个信号只施加在低数据字节上,允许16-bit模式下的字节写.WRL信号是低电平有效.

7.1.4 WRLH– 写信号高字节

对于16-bit的数据总线,类似于WRL,但是为高字节使用. WRH信号是低电平有效.

7.1.5 ALE – 地址锁存使能

由于XA外部总线的一部分用做地址和数据的复用信息,所以部分地址必须在XA外部锁存,这样在读写操作的时序中这部分地址才能保持恒定.高电平有效的ALE信号驱动外部锁存器存储数据地址或代码地址.当ALE信号结束(变低)时,外部锁存器必须关闭并保存这个地址.

7.1.7 复用地址和数据线

XA的部分总线用于数据传输,但也用于地址输出.在输出信号实施相应的总线操作以前,XA输出操作的地址.在总线的复用部分,这个地址被外部锁存器锁存,由ALE信号控制.完成以后,这部分总线就可以用做数据输出或输入.控制信号PSEN,RD,WRL,WRH决定发生的总线操作类型.

7.1.8 等待信号

WAIT信号输入允许在XA的外部总线操作中插入等待状态.在总线控制信号PSEN,RD,WRL,WRH有效以后, 如果WAIT为高,总线操作将延长,控制信号也继续驱动,直到WAIT信号变低为止.为了使用这种特性,必须提供一个外部电路以产生一个合适的外部WAIT信号.

XA有内部总线配置寄存器,允许通过编程控制XA的总线周期的长度,所以在多数应用中,不必使用WAIT信号.这个特性将在以后章节详细说明.

7.1.9 EA – 外部操作

EA脚的输入决定在复位后,XA操作在单片模式还是从外部运行程序代码.如果在Reset变高时EA为低,则第一个代码的是从外部获取的.如果在Reset变高时EA为高,则XA将先运行片内的代码.当地址超过片内代码的上限时则执行外部指令.在复位端变高以前,EA脚的电平被锁存,所以选择的模式将被有效保存直到下一次复位.

在一些XA型号中,EA脚还有别的功能.

7.1.10 BUSW – 总线宽度

外部总线宽度可以配置成8或16-bit宽度.XA允许用两种方法编程总线宽度.在一个系统中,如果被初始化从内部获取指令,则程序可以配置外部总线尺寸(和总线的其它特性),这将优先于总线现在实际使用的总线配置.

当启动代码必须从外部获取代码时,XA在从外部获取第一个代码时,必须知道外部总线的宽度.在一些XA型号中,BUSW功可能同其它功能共同使用一个管脚.在这种情况下,BUSW管脚上的电平在复位释放时被锁存起来,并保存到下一次复位.复位结束后,处理器开始正常的执行代码,这个管脚的另一个功能就可以正常使用了.

不像EA功能,由BUSW脚设置的总线宽度在复位后可以被用户程序优先修改,使用BUSW脚设置总线宽度在大多数系统中是不必要的.在程序的控制下,可以通过总线控制寄存器改变总线尺寸.这个特性将在下一节中详细介绍.

7.2 总线配置

标准XA外部总线有多个配置选择.除了上一节讨论的总线宽度外,外部总线地址线的数目也可以编程,还有总线时序.

7.2.1 8-Bit and 16-Bit 总线数据宽度

标准的XA总线允许8-bit和16-bit的总线宽度.BUSW=0,选择8-bit总线,busw=1选择16-bit总线.在上电后,XA的缺省值为16-bit宽度(由于BUSW脚内部的微弱上拉).总线宽度有复位释放时BUSW的值决定,除非用户设置总线配置寄存器(SCR),见图7.1

BCR	总线配置寄存器	46A				WAIT	BUSD	BC2	BC1	BC0

WAIT: 等待关闭.导致 XA 外部总线接口忽略 WAIT 输入.这允许将 WAIT 连接到高电平,使用内部代码,而不需要 WAIT 功能.

BUSD: 总线关闭.它将关闭 XA 的外部总线.它设计的原意是在代码执行到片内代码的边界时,防止由于予读取而意外激活总线.

BC2-BC0: 这些位用于选择总线的配置,包括数据的位数,地址线的数量.支持的组合如下:

000 8-bit数据总线,12个地址线.
001 8-bit数据总线,16个地址线.
010 8-bit数据总线,20个地址线.
011 8-bit数据总线,24个地址线.
100 <功能保留>
101 <功能保留 >
110 16-bit数据总线,20个地址线.
111 8-bit 数据总线,24 个地址线.

注:复位时,BCR 的值为 0000 0A11,A 的值由 BUSW 脚(P35)决定.

图7.1 总线配置寄存器(SCR)

图7.2和7.3显示了当使用不同的总线宽度时,XA提供的地址和数据功能以及相关脚.

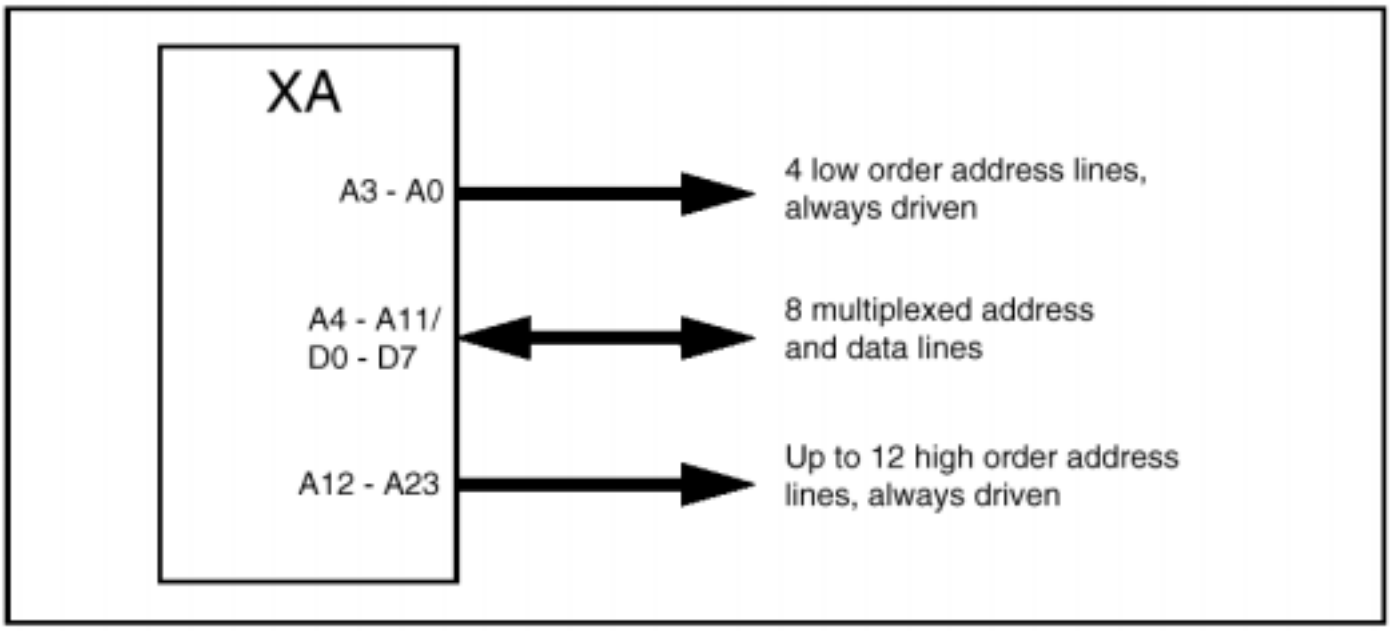

图**7.2 8-bit**外部总线配置

7.2.2 典型外部器件连接

XA总线支持连接多种外部器件,如EPROM,RAM和其它存储器器件,串行口和并行口的扩展.下图显示了在8-bit和16-bit模式中的一般连接.

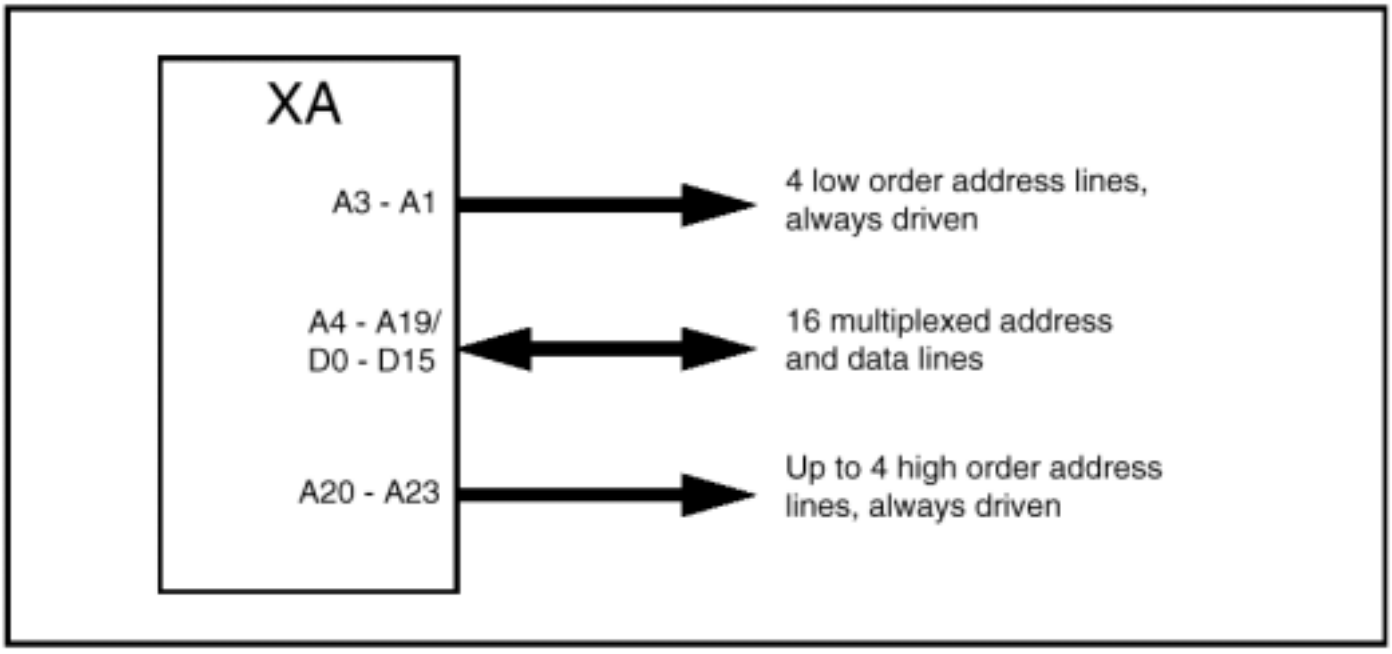

图7.3 16-bit外部总线配置

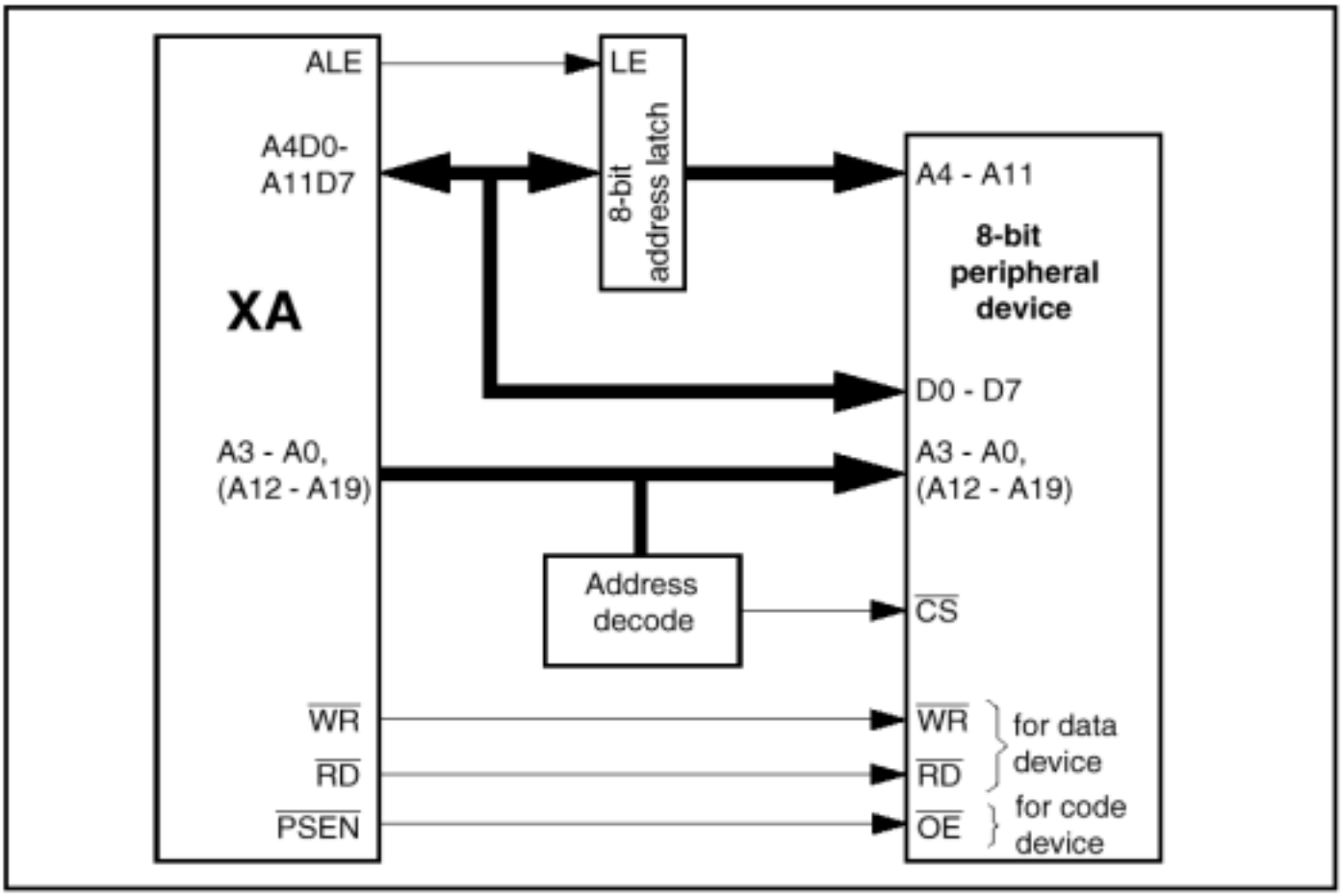

图7.4 XA外部总线与8-bit器件的连接

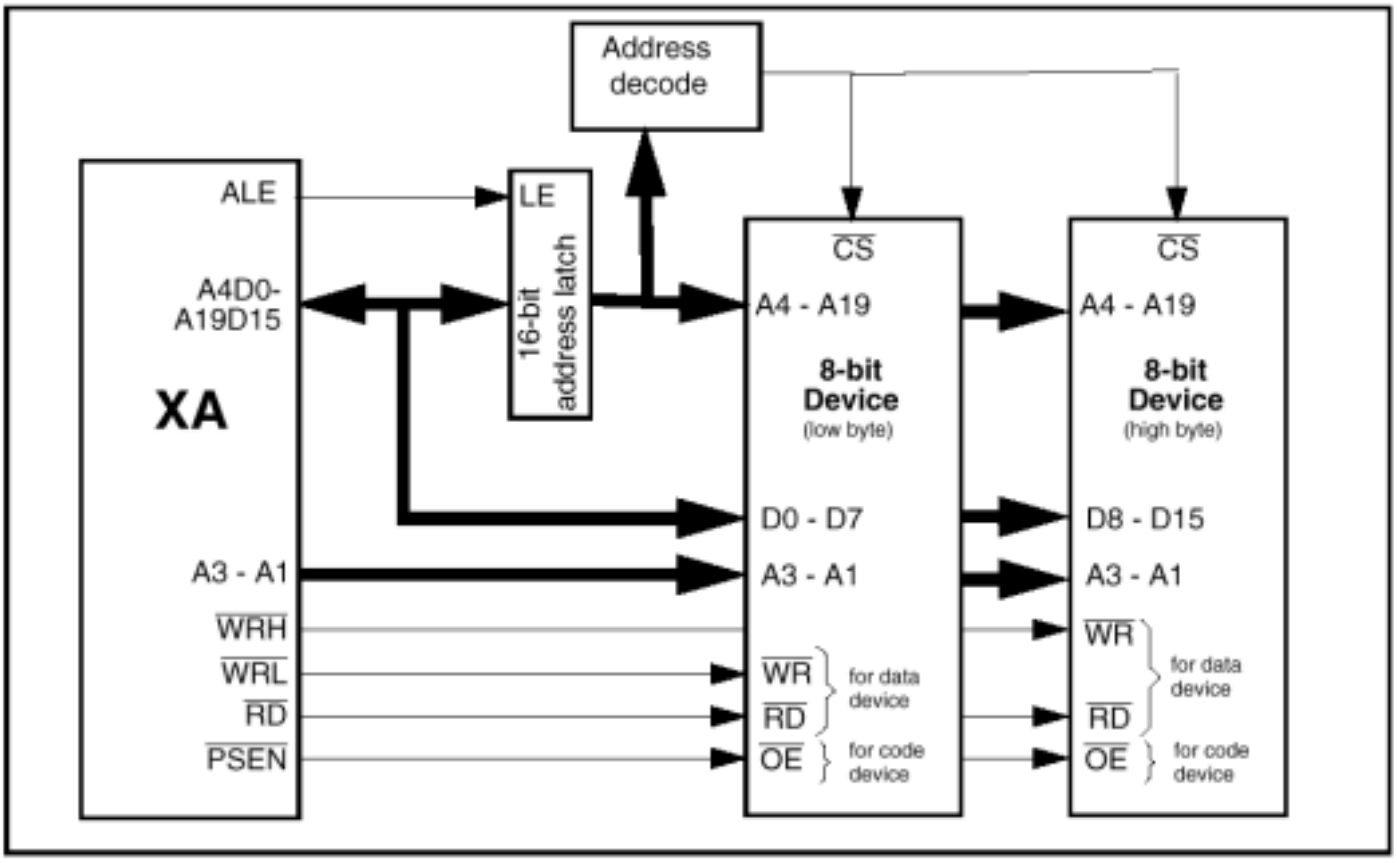

图7.5 XA外部总线与16-bit器件的连接

7.3 总线时序

标准的XA外部总线允许对总线控制信号ALE,PSEN,WRL,WRH和RD的宽度进行编程.还有一个选择

项用于写操作时延长数据保持时间.有效的组合可以使XA同大多数器件直接相连而不必要使用特殊缓冲器和WAIT信号.注意,在任何类型的总线周期以后,总是有一个"间歇时钟",除了串联代码读模式(Burst).也就是说,当一个总线周期完成以后,总线控制信号撤消后,在新的总线周期到来以前总是有一个时钟间隔,在这期间无总线活动.

7.3.1 代码存储器

外部代码存储器一般是EPROM的形式,由PSEN控制信号启动.如果XA在复位时由EA脚配置成执行内部代码,超出内部代码边缘后将自动到外部获取代码.这个边界根据不同的器件,内部代码的尺寸有所变化.由于XA使用予获取队列来优化指令执行时间,当程序执行到内部程序的边缘时,可能要到外部读取程序代码.如果程序分支或子程序返回也接近内部程序边界时,外部程序代码的读取是不必要的;在XA的外部总线口作为其它用途时会发生问题.基于这个原因,总线配置寄存器中的BUSD(总线关闭)位能阻止外部总线进行代码和数据的操作.

注意外部代码读周期有时也会被XA中断.当总线上代码予获取发生时,而XA必须执行一个分支.来自代码予获取的指令数据不再需要,所以总线周期会立即关闭.表现为PSEN没有同ALE共同出现,或PSEN比正常宽度要短.

有ALE的代码读

标准的地址和数据复用操作在每一个总线周期中地址和相关的控制信号都会出现.XA中使用ALE来表明现在总线上出现的地址必须被锁存,并延续该代码或数据操作.下面是使用ALE的代码获取图解.

串联模式代码读(无ALE)

在每一个代码获取时,XA并不是都需要ALE信号.在执行外部代码时,这种特性将通过减少外部器件的操作时间的方法改善性能.由于低4位地址线是直接驱动的,不是复用,所以每个ALE信号的出现XA可以连续操作16个字节(或8个字).这种快速的代码连续读取叫做串联读.当然,任何类型的跳转,分支,中断和程序连续流程的变化都将需要ALE信号以改变代码读取地址,这是一个非连续的现象.XA外部总线的任何数据操作(读和写)也需要ALE周期,同时也会引起由ALE开始的外部代码读取.

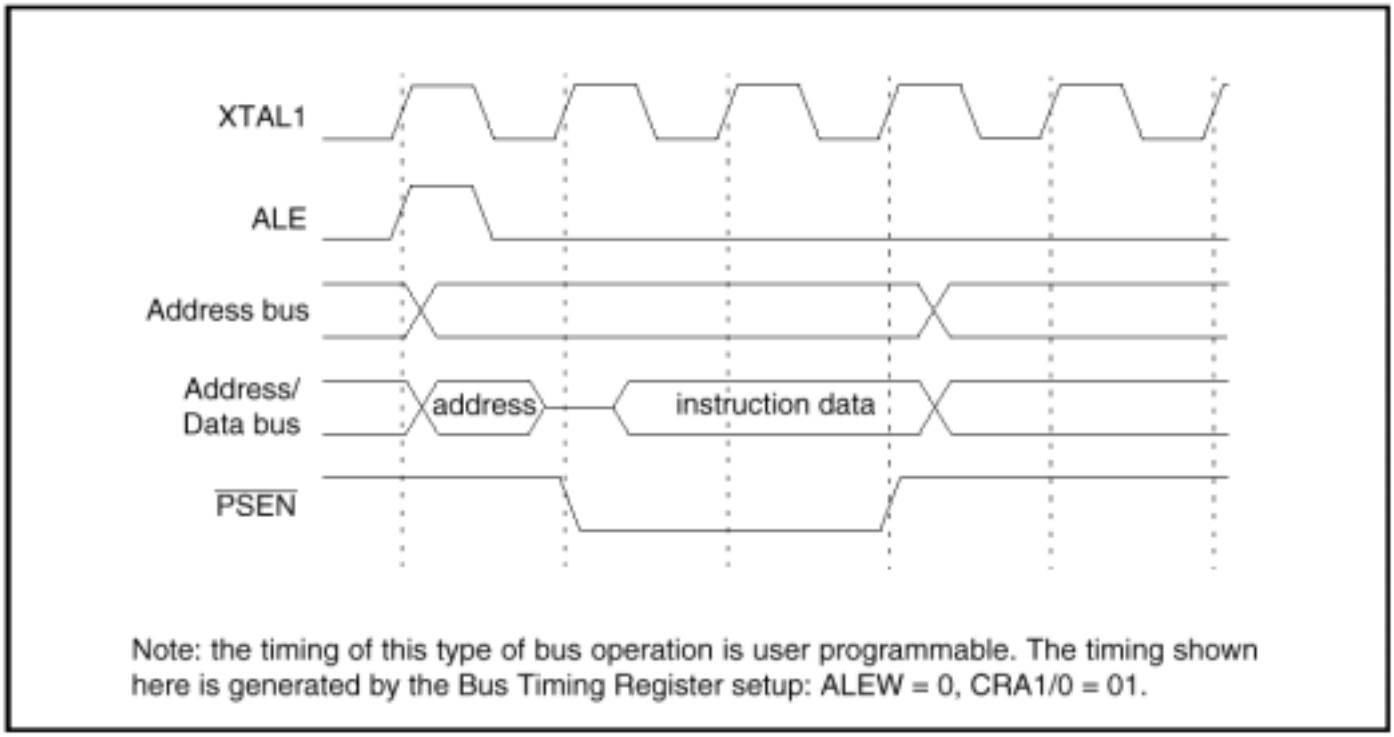

图 **7.6** 典型的有 **ALE** 的外部代码读取

下图表明了一个典型无 ALE 的连续代码读取.注意 PSEN 控制信号不再发生变化,而在串联代码读中一致保持有效电平.

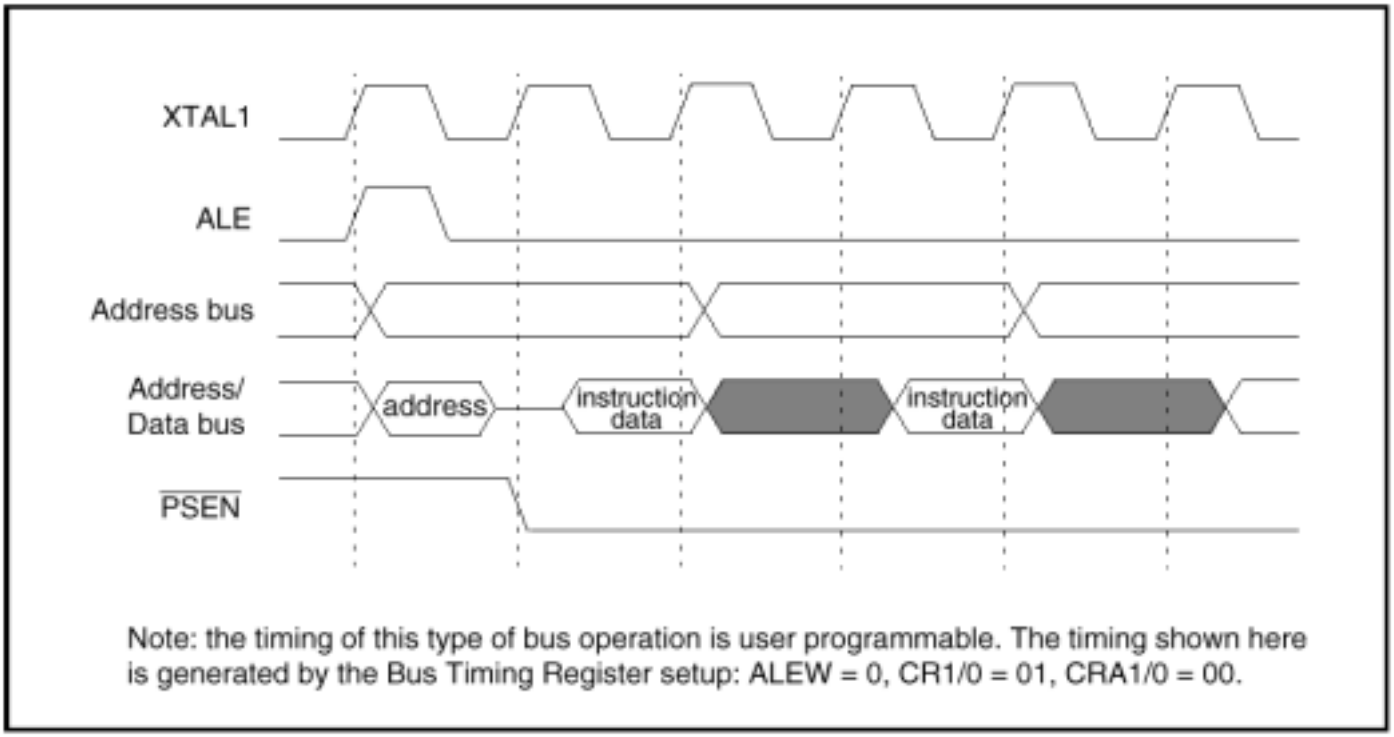

图 7.7 外部代码读的串联模式(连续)

7.3.2 数据存储器

XA中外部总线的读和写有RD,WRL,WRH信号控制.由于XA总线支持8-bit和16-bit宽度,以及字和字节的读写,所以基本的总线周期可能有不同的形式.这将在下面的章节中论述.

数据区域,类似于代码区域,内部的数据区域有一个边界,超出这个边界XA将切换到外部数据区域.这个数据区域边界根据器件的不同而有所不同.

典型数据读

在8-bit或16-bit总线中,一个简单的字节读总是由ALE开始的,同时XA在总线上提供读数据需要的地址位置.随后施加RD信号,将导致外部器件把数据提供在总线上.这个过程如下图解.

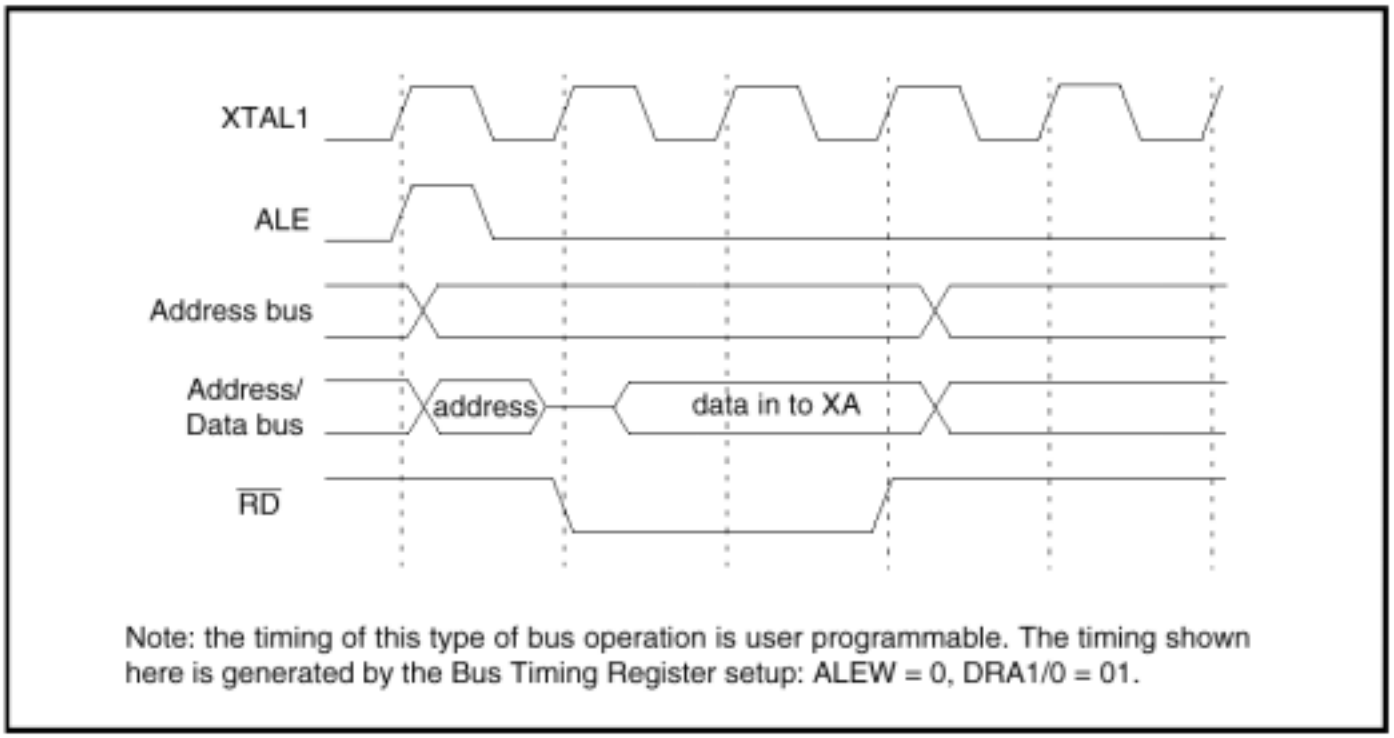

图 7.8 典型的外部数据读

8特殊功能寄存器总线

所有的特殊功能寄存器都通过特殊功能寄存器总线连接到CPU,这样它们就可以被程序读写.这包括所有外设中的寄存器,像定时器,串行口,它们同CPU的寄存器是一样的,如PSW.CPU的寄存器同CPU的通讯是通过直接的连接,但是对它们读写操作是通过SFR总线的.SFR总线为XA核的新增功能附件提供通用的接口,这样就可以制造众多的变化的微控制器系列.见图8.1.

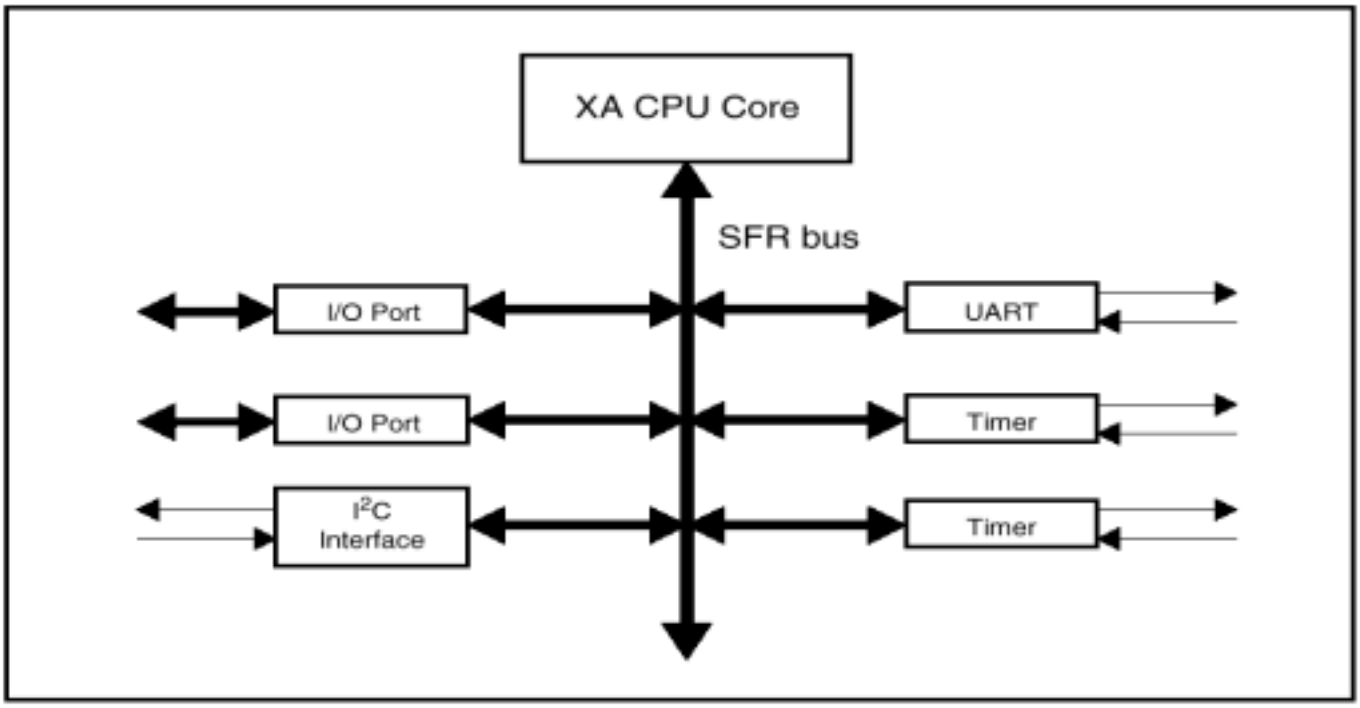

图 8.1 连接到 SFR 总线的功能外设

8.1 执行和可能的增加量

SFR总线接口不是CPU核的一部分,而是一个独立的功能块.由于SFR总线控制器是一个独立块,所以对SFR的写可以同下一条指令同时发生.如果下一条指令仍试图操作SFR总线,而总线正处于忙状态,则指令的执行就要等待,直到SFR总线空闲.通过SFR总线的读和写要占用2时钟周期.但是,在这两个时钟周期的开始还有一个不确定的时钟周期,所以从SFR总线收到申请到执行完毕是2到3个时钟周期.

SFR总线是8-bit接口.这意味这不允许以字为单位的读和写.在将来,更高性能的XA器件可能扩展SFR总线的容量,支持16-bit的操作.SFR性能增强之一是它把16-bit的操作转换成8-bit的操作.虽然实际的SFR总线是8-bit宽度,但用户可以把它当作16-bit的来使用.更高性能的选择是全16-bit的SFR总线.这需要增加XA的硬件来实现,最终可能要靠将来功能增强的XA核来实现.

8.2 读-修改-写锁定

一些通过SFR总线操作的SFR包含中断位和其它信息位,由外设器件直接设置.当一个读-修改-写操作施加到这样的存储器上后,在操作期间可能会有外设对一个标志位进行操作.XA定义了一个标准机制以处理这种情况,叫做读-修改-写锁定. 读-修改-写定义为一个单XA操作,操作期间对SFR进行读-修改-写.

符合上述的指令是写SFR中的位和修改SFR的全部,只是除了MOV指令.这有点像第7章的读口锁存器而不是读口管脚,只是涉及的SFR不同.

在XA外设中使用这样的机制来避免在读-修改-写操作中丢失状态标志,首先将检查这样的操作是否在执行中.CPU发出一个信号到外设用于指明现在的状态.当外设检测到这个信号后,如果在读-修改-写操作以后外设更新了这些状态标志,外设将不允许CPU对这些位进行写.这好象是外设在读-修改-写操作以后刷新的这些位,而不是在它的期间.一端读-修改-写操作结束,CPU对SFR的写的效果才能显示出来.

9 80C51 兼容性

当XA核设计时,很多结构和特征都考虑了兼容80C51.处理器的存储器配置,存储器的寻址模式,指令集

和很多其它的东西都必须考虑到.

9.1 兼容性考虑

选择源代码的兼容作为目标有很多原因. 如果想在本质上获得更高的性能, 同一个已经存在的处理器完全兼容是不可能的. XA结构使用了很多规则来兼容80C51. 一个80C51到XA的源代码转换程序用来提供两种结构之间的兼容. 为了使转换器相当的简单, 所有80C51的指令进行一对一的传输是首先考虑的. XA的指令集同80C51的指令集比较起来更有效, 但执行大约相同的功能. 这样80C51的指令集于是就可以转换成响应的XA指令. XA的地址图和寻址方式是80C51的高级集合, 这样使源代码的转换很容易实现. 其它CPU的特性在可能的范围内尽量的兼容. 在个别的情况下,如果由于重要的原因无法提供兼容时,在保证XA的高性能和低成本的情况下把变化尽量减小.

9.1.1 兼容模式,地址图,和寻址

XA保留一些特殊寄存器以供在代码传输时充当80C51的寄存器. A寄存器, B寄存器, 和数据指针均在XA寄存器组中预先确定(见图9.1). 在XA中, 累加器是这些寄存器当中唯一需要硬件支持的,这是因为累加器可以被某些指令直接读或测试以便产生校验标志.

80C51的4个寄存器组在XA中有双份.唯一不同的是在XA中这些寄存器不占用数据空间低32字节, 而80C51要占用. 为了允许代码的传输, XA中有一个兼容模式,可以把等同于80C51的XA寄存器队列复制到数据空间中. 这个模式的启动可以通过设置系统配置寄存器(SCR)中的CM位.

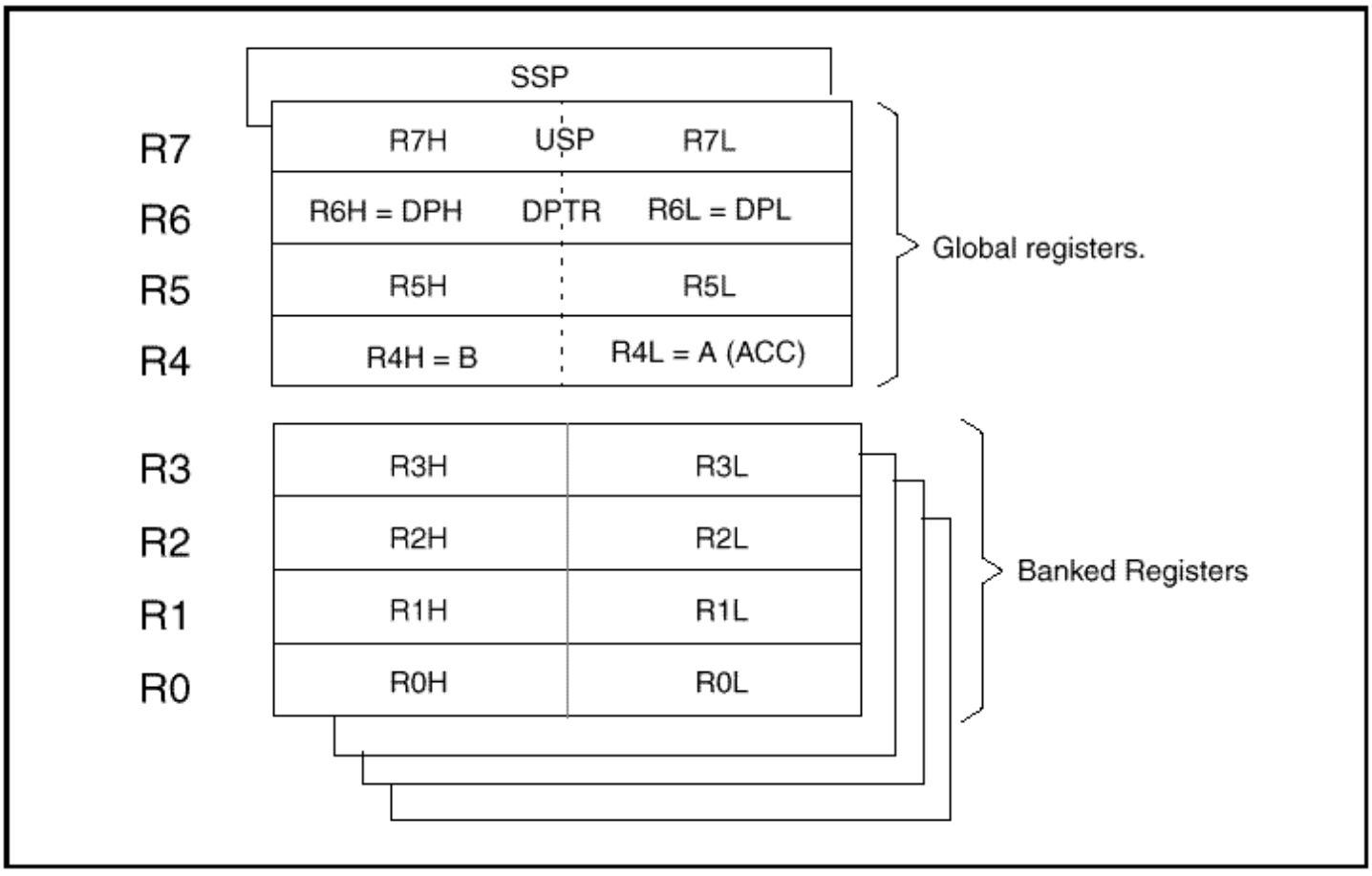

图 **9.1. XA** 寄存器队列

其它80C51的重要寄存器以其它方式提供.在XA中程序状态字(PSW)同80C51相比有轻微的不同, 所以XA保留了一个特殊地址以提供在代码传输时同80C51兼容使用.这个变化了的PSW,叫PSW51,见图9.2. F0和F1是简单的可读写位.P标志是80C51 A寄存器的偶校验位,总是反映该寄存器当前的内容.注意,P标志,F0标志,F1标志仅仅出现在PSW51寄存器中.

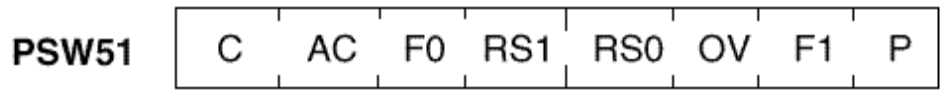

图 **9.2 PSW CPU** 状态字

80C51 的数据区域间接寻址模式,使用 R0 或 R1 作为指针,需要 XA 中特殊支持,这里的指针通常是

16-bit 长度.XA 中 80C51 的兼容模式也模仿 80C51 的间接寻址方式,使用寄存器队列当中开始的两个字节作为间接指针,每个从 0 可以扩展到 16-bit 的地址.得益与这种特性和前面所说的寄存器覆盖到数据区域的特性,打开兼容模式时,XA 可以执行绝大多数经过转换的 80C51 代码.除了上述的变化,在 XA 中再没有由于兼容模式而发生变化的功能.

80C51 把特殊功能寄存器(SFR)放在直接寻址空间,从 80H 到 FFH.SFR 的获取必须在指令中包含全部的 SFR 地址,所以在传输到 XA 中也是相当的简单.由于在 80C51 源代码中调用 SFR 通常都使用它们的名字,所以传输器只需要把它们的名字复制到 XA 代码输出中就可以了.如果一个 SFR 调用的是它们的地址,必须找到它们的名字以便插入到 XA 代码中.这就需要原来写代码时使用的 80C51 器件的 SFR 表.

XA 有另外一个模式对 80C51 的传输特别有用.为了节省堆栈空间和加快运行,Page 0 模式存储在堆栈中的返回地址只是 16-bit,而不是通常的 24-bit(由于 XA 中必须字相邻所以占用 32bit).全部的程序和数据寻址被强制为 16-bit.如果全部的 80C51 应用程序转换到 XA,它也适合于 64K 的限制,也允许使用这种模式.

XA 处理器的其它方面,堆栈也发生了变化.XA 采用的是标准 16-bit 处理器的堆栈生长方式.于是 XA 的堆栈是向下生长的,在数据区域中从高到低.如果需要,现在堆栈可以有 64K 的尺寸,从它的数据段中任意的地址开始,所以在传输 80C51 应用程序时可以容易的移到任何位置.这种 XA 堆栈方向的变化对于堆栈目录和通常的数据区域操作之间的匹配是很重要的.

80C51 代码传输到 XA 中,运行中需要更多的堆栈空间,原因有两个.第一,在 XA 中当中断和异常处理时,PSW 都自动存储.在 80C51 中也需要外部存储 PSW,但 XA 的 PSW 是 16 位的.第二,XA 的启动执行只允许在堆栈当中以字存储.虽然字节和字都可以操作,但是这两中类型都使用 16 bit 的堆栈空间.

由于堆栈尺寸的增大,加上堆栈生长方向的变化,所以在一个完整的 80C51 程序传输到 XA 中,必须做一些修改.

9.1.2 中断和异常处理

XA所特有的异常处理功能同80C51相比就强大的多.由于增加了这种功能,所以在80C51代码转换时有一些不同必须注意到.

前面说过,在中断处理时,XA将自动存储PSW.如果80C51的程序延续这种做法而不加修改,将无法工作.但是这种情形在通常的程序中很少发生.

XA有15个中断优先级,而80C51只有2个优先级,即使一些新型号的80C51也只有4个优先级.这些优先级存储成4-bit,这样一个相同的SFR中有两个中断的优先级数据.这是XA不同于也是强于80C51之处,但在代码传输时需要另外的修改.

在XA中,使用存储在代码地址低端的矢量表进入中断处理程序.每一个中断和异常中断都有一个矢量,包含相应的中断处理程序的入口地址和一个新的PSW,当进入中断处理程序时调入这个新的PSW.这同80C51固定地址的中断处理程序相比更显灵活和有效.所以,当一个完整的80C51应用程序要转换成XA代码时,中断服务程序必须重新定位于矢量表以上,并把新地址存储在矢量表中.

9.1.3 片上外设

80C51的标准片上外设在XA中基本得到了保留,但是功能得到了加强.这些外设包括Uart,T0,T1,T2.

XA的Uart功能虽然得到加强,但不会影响80C51的传输.一些附加的新功能是通过使用新SFR加入的,例如浈检查,溢出检查,结束检查.

T0和T1基本一致,但一项功能不同,时序也不同.变化的功能是删除了8048定时器(模式0),代替的是更有用的16-bit自动装载模式.为了支持新模式0,添加了16-bit重装载寄存器RTHn和RTLn.在模式2,RTLn代替THn,作为8-bit重装载寄存器.所有定时器相对与微处理器时钟的记数率也发生了变化.由于加入了灵活的可编程特性,允许所有定时器的记数率可以为时钟的1/4,1/16,1/32.由于XA的高操作性能,一个80C51应用程序在转换成XA后,不在需要相同的时钟频率.

9.1.4 总线接口

原先的80C51总线控制信号在XA都继续使用.为了提供更好的性能,这些信号作了一些细微的变化,或加入了新的内容.在现有的ALE, PSEN, RD, WR,和EA的基础上,还加入了WAIT,WRH.WAIT信号在总线操作中将在总线时钟中插入等待状态,直到WAIT信号撤消.WRH在XA配置成16-bit总线宽度时,用于

高顺序字节写.

复用地址/数据总线也作了一些变化.为了在系统执行外部程序代码时获得教高的性能,XA把最低的4位地址线分离出来而拥有单独的管脚.这是因为这些地址线变化的最频繁,如果像80C51一样复用,每个代码的读取都需要ALE信号.80C51有时间这样做,因为它的的性能还不够高.作为XA,仅仅使用必要的时钟执行每一条指令,所以每次的代码读取都需要ALE将显得缓慢.有了这种变化,XA总线可以一次读取16比特的代码而不必每次都插入ALE信号.XA的每一种总线操作(代码读,数据读,数据写)所需的时钟数目都是可以编程的,所以即使较慢的器件在同XA协同工作时也不需要外部等待信号产生器.
由于提到的总线变化,如果使用外部总线,XA器件不能同80C51在管脚上兼容.所以在应用中硬件要作小而简单的变化.
XA指令兼容的最终目标是每一条80C51指令转换成一条XA指令.除了一条指令外,都可以实现.80C51指令XCHD,不能这样变换.XCHD在80C51中是一条很少用的指令,由于XA的内部结构而无法实现,否则将要增加很多的处理电路.所以,在80C51指令传输中遇到这条指令后,将用多条指令来实现该指令的相同功能.

```
PUSH R4H;          保存临时寄存器
MOV R4H,(Ri);      获取数据
RR R4H,#4;         交换一个字节
RR R4L,#4;         交换第二个字节("A" 寄存器)
RL R4,#4;          交换字
                   ;结果是A和R4H的半字节交换
MOV (Ri),R4H;      储存结果
POP R4H;           恢复临时寄存器
```

如果应用程序需要这个过程不能被中断,必须另加一些指令以关闭和开启中断.在本节的尾部的表格中列出了80C51指令的XA代换.
XA的指令集比80C51要有效的多,作为直接代换,代换后平均的字节数要多一些.对于为XA写的代码,由于单字节指令增多,所以全部XA程序的大小要比用80C51写相同程序的字节数要小.当然,这还要看是否使用了XA的新特性.当代码是由80C51源代码转换来的,字节数要大一些.
在使用跳转表的情况下,JMP @A+DPTR指令用来跳到一个表内,这个表内包含其它跳转指令,XA不能模仿表中长度为2字节的跳转指令,如AJMP指令.这样就必须对表目录进行计算和调整,使代码转换能够完成.对于数据表,用MOVC @A+PC获取数据,表的深度可能发生变化,所以需要对目录进行调整.
由于XA对每条指令进行了时序优化,所以80C51的源代码同传输后的XA代码时序会不相同.如果指令的时序对应用程序很关键,传输后的代码必须改变,也许要加上一些NOP指令,或循环延迟,以提供必要的时序.
为了演示一个简单的90C51到XA源代码的传输是如何工作的,下面给出了一个从正常运行的80C51程序中提取的子程序,后面的传输结果表,和提供的其它的规则.80C51源代码为:

```
;StepCal – 计算马达的行程点满量程的百分数(0 - 100%).
;调用目标数值在A中. 返回结果在A中和"StepResult".
StepCal:
        MOV Temp2,A ;          保存步进目标,以后使用.
        MOV B,#Steplow ;       得到步进增量的低字节.
        MUL AB ;               乘以步进目标.
        MOV StepResult,B ;     保存高字节部分结果.
        MOV Temp1,A ;          保存低字节以备旋转使用.
        MOV A,Temp2 ;          回存步进目标.
        MOV B,#StepHigh ;      取得步进增量的高字节,
        MUL AB ;               相乘.
        ADD A,StepResult ;     两个部分结果相加.
        JNB Temp1.7,Exit ;     最小有效字节> 80h?
        INC A ;                如果是, 靠近精确结果.
Exit:
```

```
        ADD A, #MotorBot ;          加上0步进增量.
        MOV StepResult,A ;          保存最后步进目标.
        RET

        ;传输后的XA代码为:
StepCal:
        MOV Temp2,R4L ;
        MOV R4H,#Steplow ;
        MULU.b R4,R4H ;
        MOV StepResult,R4H ;
        MOV Temp1,R4L ;
        MOV R4L, Temp2 ;
        MOV R4H, #StepHigh ;
        MULU.b R4, R4H ;
        ADD R4L, StepResult ;
        JNB Temp1.7,Exit ;
        ADDS R4L,#1 ;
Exit:
        ADD R4L,#MotorBot ;
        MOV StepResult,R4 ;
        RET
```

在这种情况下,传输后的代码实际上变化很小.首先,80C51的寄存器名被XA中的保留名代替.INC指令被短的ADDS代替,助记符MUL变成了MULU8.
表9.1是关于这些简单代码一些基本的统计信息.这些统计信息显示了XA代码大幅度提高了性能.这种提高还仅仅是简单的代码传输,还没使用XA的任何独特性能.

统计	80C51 代码	XA 传输	注释
代码比特	28	40	XA 中分支相邻处加一个 NOP 指令.
执行的时钟数目	300	78	包含 XA 予读取分析,实际运行 66
20MHz 下的执行时间	15 μ s	3.9 μ S	无任何优化下 4 倍改善

9.2 代码传输

表9.2列出每一个80C51指令类型和替换的XA指令.实际上的XA源代码转换也可以使用这个表,但要注意本节说明的兼容性的特例,和对XA源代码结果做适当的调整.

Table 9.2: 80C51 到 XA 指令转换

80C51指令	**XA转换**
数学运算	
ADD A, Rn	ADD.b R, R
ADD A, #data8	ADD.b R, #data8
ADD A,dir8	ADD.b R, direct
ADD A, @Ri	ADD.b R, [R]
ADDC A, Rn	ADDC.bR, R
ADDC A, #data8	ADDC.bR, #data8
ADDC A,dir8	ADDC.bR, direct
ADDC A, @Ri	ADDC.bR, [R]
SUBB A, Rn	SUBB.bR, R
SUBB A, #data8	SUBB.bR, #data8

SUBB A, dir8	SUBB.bR, direct
SUBB A, @Ri	SUBB.bR, [R]
INC Rn	ADDS.bR, #1
INC dir8	ADDS.bdirect, #1
INC @Ri	ADDS.b[R], #1
INC A	ADDS.bR, #1
INC DPTR	ADDS.wR, #1
DEC Rn	ADDS.bR, #-1
DEC dir8	ADDS.bdirect, #-1
DEC @Ri	ADDS.b[R], #-1
DEC A	ADDS.bR, #-1
MUL AB	MULU.bR, R
DIV AB	DIVU.b R, R
DA A	DA R
逻辑操作	
ANL A, Rn	AND.b R, R
ANL A, #data8	AND.b R, #data8
ANL A, dir8	AND.b R, direct
ANL A, @Ri	AND.b R, [R]
ANL dir8, A	AND.b direct, R
ANL dir8, #data8	AND.b direct, #data8
ORL A, Rn	OR.b R, R
ORL A, #data8	OR.b R, #data8
ORL A, dir8	OR.b R, direct
ORL A, @Ri	OR.b R, [R]
ORL dir8, A	OR.b direct, R
ORL dir8, #data8	OR.b direct, #data8
XRL A, Rn	XOR.b R, R
XRL A, #data8	XOR.b R, #data8
XRL A, dir8	XOR.b R, direct
XRL A, @Ri	XOR.b R, [R]
XRL dir8, A	XOR.b direct, R
XRL dir8, #data8	XOR.b direct, #data8
CLR A	MOVS R, #0
CPL A	CPL.b R
SWAP A	RL.b R, #4
RL A	RL.b R, #1
RLC A	RLC.b R, #1
RR A	RR.b R, #1
RRC A	RRC.b R, #1
CLR C	CLR bit
CLR bit	CLR bit
SETB C	SETB bit
SETB bit	SETB bit
CPL C	XOR.b PSWL, #data8
CPL bit	XOR.b direct, #data8
ANL C, bit	AND C, bit
ANL C, /bit	AND C, /bit
ORL C, bit	OR C, bit
ORL C, /bit	OR C, /bit
MOV C, bit	MOV C, bit
MOV bit, C	MOV bit, C
数据传输	
MOV A, Rn	MOV.b R, R

MOV A, #data8	MOV.b R, #data8
MOV A, dir8	MOV.b R, direct
MOV A, @Ri	MOV.b R, [R]
MOV Rn, A	MOV.b R, R
MOV Rn, #data8	MOV.b R, #data8
MOV Rn, dir8	MOV.b R, direct
MOV dir8, A	MOV.b direct, R
MOV dir8, #data8	MOV.b direct, #data8
MOV dir8, Rn	MOV.b direct, R
MOV dir8, dir8	MOV.b direct, direct
MOV dir8, @Ri	MOV.b direct, [R]
MOV @Ri, A	MOV.b [R], R
MOV @Ri, dir8	MOV.b [R], direct
MOV @Ri, #data8	MOV.b [R], #data8
MOV DPTR, #data16	MOV.w R, #data16
XCH A, Rn	XCH.b R, R
XCH A, dir8	XCH.b R, direct
XCH A, @Ri	XCH.b R, R
XCHD A, @Ri	替代序列(见文字说明)
PUSH dir8	PUSH.bdirect
POP dir8	POP.b direct
MOVX A, @Ri	MOVX.b R, [R]
MOVX A, @DPTR	MOVX.b R, [R]
MOVX @Ri, A	MOVX.b [R], R
MOVX @DPTR, A	MOVX.b [R], R
MOVC A, @A+DPTR	MOVC.b A, [A+DPTR]
MOVC A, @A+PC	MOVC.b A, [A+PC]
相对分支	
SJMP rel8	BR rel8
CJNE A, dir8, rel	CJNE.b R, direct, rel
CJNE A, #data8, rel	CJNE.b R, #data8, rel
CJNE Rn, #data8, rel	CJNE.b R, #data8, rel
CJNE @Ri, #data8, rel	CJNE.b [R], #data8, rel
DJNZ Rn, rel	DJNZ.b R, rel
DJNZ dir8, rel	DJNZ.b direct, rel
JZ rel	JZ rel
JNZ rel	JNZ rel
JC rel	BCS rel
JNC rel	BCC rel
Jump,Call,Return等	
NOP	NOP
AJMP addr11	JMP rel16
LJMP addr16	JMP rel16
JMP @A+DPTR	JUMP [A+DPTR]
ACALL addr11	CALL rel16
CALL addr16	CALL rel16
RET	RET

9.3 XA中的新指令

X的指令同80C51类似,但有更大的地址范围,更多的状态标志,等等,XA还有全新的指令和寻址方式,使XA写入的代码更容易,更有效.新的寻址方式使XA在高级语言汇编下工作的更好.XA新指令的完全列表和寻址模式见表9.3.

表 **9.3: XA** 的新指令和寻址方式

alu.w ..., ...	所有的 80C51 数学和逻辑运算,但具有 16-bit 宽度.
SUBB R,...	减(无借位),所有的寻址模式.
alu [R], R	从一个寄存器到间接地址的数学和逻辑运算(ADD, ADDC,SUB, SUBB, CMPAND, OR, XOR,和MOV)
alu R, [R+]	从一个间接地址(8 或 16-bit 偏移)到寄存器的数学/逻辑运算.
alu R,[R+offset8/16]	从一个间接偏移地址(8 或 16-bit 偏移)到寄存器的数学/逻辑运算.
alu direct, R	80C51 只有 MOV direct,R
alu [R], R	80C51 只有 MOV [R],R
alu [R+], R	从一个间接地址到寄存器的数学/逻辑运算,间接指针自动增加.
alu [R+offset8/16], R	从一个寄存器到间接偏移地址的数学/逻辑运算(8 或 16-bit 偏移).
alu direct, #data8/16	8 或 16-bit 立即数到直接地址的数学/逻辑运算.
alu [R], #data8/16	8 或 16-bit 立即数到间接地址的数学/逻辑运算.
alu [R+], #data8/16	8 或 16-bit 立即数到间接地址的数学/逻辑运算,间接指针自动增加.
alu [R+offset8/16], #data8/16	8 或 16-bit 立即数到间接偏移地址(8 或 16-bit 偏移)的数学/逻辑运算.
MOV direct, [R]	从间接地址到直接地址移动数据.
ADDS R, #data4	80C51 只能对寄存器增 1 或减 1.ADDS 可以有+7 到-8 的范围.
ADDS [R], #data4	加一个短数值到间接地址.
ADDS [R+], #data4	加一个短数值到间接地址,间接指针自动增加.
ADDS [R+offset8/16], #data4	加一个短数值到间接偏移地址.(8 或 16-bit 的偏移)
ADDS direct, #data4	加一个短数值到直接地址.
MOVS ..., #data4	加一个短数值到目的地址,可使用同 ADDS 相同的寻址模式.
ASL R, R	数学左移位一个字节,字,或双字,可达 31 个位置,移位数来自寄存器.
ASR R, R	数学右移位一个字节,字,或双字,可达 31 个位置,移位数来自寄存器.
LSR R, R	逻辑右移位一个字节,字,或双字,可达 31 个位置,移位数来自寄存器.
ASL R, #DATA4/5	数学左移位一个字节,字,或双字,可达 31 个位置,移位数来自寄存器.
ASR R, #DATA4/5	数学右移位一个字节,字,或双字,可达 31 个位置,移位数来自寄存器.
LSR R, #DATA4/5	逻辑右移位一个字节,字,或双字,可达 31 个位置,移位数来自寄存器.
DIV R, R	有符号的 32-bit/16-bit,或 16-bit/8-bit
DIVU R, R	无符号的 32-bit/16-bit,或 16-bit/8-bit
MUL R, R	有符号的 16-bit*16-bit,或 8-bit*8-bit
MULU R, R	无符号的 16-bit*16-bit,或 8-bit*8-bit
DIV R, #data8/16	有符号 32-bit 寄存器除 16-bit 立即数, 16-bit 寄存器/8-bit 立即数
DIVU R, #data8/16	无符号 32-bit 寄存器除 16-bit 立即数, 16-bit 寄存器/8-bit 立即数
MUL R, #data8/16	有符号 16-bit 寄存器乘 16-bit 立即数, 8-bit 寄存器/8-bit 立即数
MULU R, #data8/16	无符号 16-bit 寄存器乘 16-bit 立即数, 8-bit 寄存器/8-bit 立即数
LEA R, R+offset8/16	调入有效地址,调入 8-bit/16-bit 偏移地址并存入寄存器.
NEG R	取寄存器的补数.
SEXT R	符号延续,从上一次的操作中复制符号标志到 8/16-bit 寄存器.
NORM R, R	标准化.左移一个字节,字,双字,直到最高位变成 1,移位次数放置在寄存器中.
RL, RR, RLC, RRC R,#data4	全部的 80C51 旋转模式,但为 16-bit 宽度,还有可变旋转次数.
MOV [R+], [R+]	块移动.从一个间接地址到另一个间接地址移动数据,两个指针增加.
MOV R, USP 和 USP, R	允许系统代码到或从用户堆栈指针.方便于多任务程序.
MOVC R, [R+]	从代码中的间接地址移动地址到寄存器,间接指针自动增加.
PUSH 和 POP Rlist	用一条指令 PUSH 或 POP 8 个字寄存器.
PUSHU 和 POPU Rlist 或直接地址	允许系统代码写或读系统堆栈.方便于多任务系统.
条件分支	完整的条件分支,包括BEQ, BNE, BG, BGE, BGT, BL, BLE, BMI, BPL,BNV, 和BOV.
CALL [R]	间接调用,到达包含在一个寄存器内的地址.
CALL rel16	调用任意位置在+/-64K 范围.
FCALL addr24	远距离调用,可以在 XA 的 16M 代码空间的任意位置
JMP [R]	间接跳转,跳转到由寄存器内容指定的地址
JMP rel16	跳转到+/- 64K 的范围
JMP [[R+]]	跳转到两次间接寻址,自动地址增加.用来分支到列表中的顺序地址

BKPT	断点,调试特性.
RESET	允许软件用一条指令复位 XA.
TRAP #data4	调用 16 个系统服务程序的一个.像一个立即中断.