第 1 章　C55x 编程基本指南

1.1　引　　言

本节列出了 TMS320C55x 系列 DSP 结构的一些重要特征，给出了创建高效运行代码的推荐方法。

1.1.1　TMS320C55x 结构

TMS320C55x 是一款定点数字信号处理器(DSP)，DSP 的主要部分是中央处理单元(CPU)，它具有下列特征：

- 统一的程序/数据存储器映射，映射包括可由 24 位地址访问的 16M 字节的程序空间和可由 23 位地址访问的 8M 字的数据空间。
- 64K 字的用于和外围设备通信的 I/O 空间。
- 支持 16 位和 32 位出栈、压栈操作的软件堆栈。用户可以用它来做数据存储和取回。CPU 用这些堆栈来进行自动的现场保护(响应调用或中断)和恢复(返回调用或被中断的代码序列)。
- 多套数据和地址总线支持高度的并行性。1 条 32 位的数据总线和 1 条 24 位的地址总线支持取指。3 条 16 位的数据总线和 3 条 24 位的地址总线用于向 CPU 传输数据，2 条 16 位的数据总线和 2 条 24 位的地址总线用于从 CPU 传输数据。
- 1 个指令缓冲器和 1 个独立的取出机制。因此，取指独立于其他的 CPU 动作。
- 具有如下的计算块：1 个 40 位的算术逻辑单元(ALU)、1 个 16 位的 ALU、1 个 40 位的移位器、2 个乘法累加器(MAC)。在一个周期内，每个 MAC 可执行一次 17 位乘 17 位的乘法运算(分数或是整数)和一次带有可选的 32/40 位饱和度的 40 位加法或减法运算。
- 一条受保护的指令流水线。流水线保护机制在必要的时候插入等待周期以防止读写操作脱离预期顺序。
- 数据地址产生单元支持线性寻址、循环寻址和位反转寻址。
- 中断控制逻辑可以停用(或屏蔽)某些可屏蔽的中断。
- TMS320C54x 兼容模式支持原来为 TMS320C54x™DSP 编写的代码。

1.1.2　获得最佳性能的代码开发流程

图 1-1 所示的流程图说明了如何让用户的代码获得最佳的性能和最高的代码生成效率，接着介绍了流程的各个阶段。

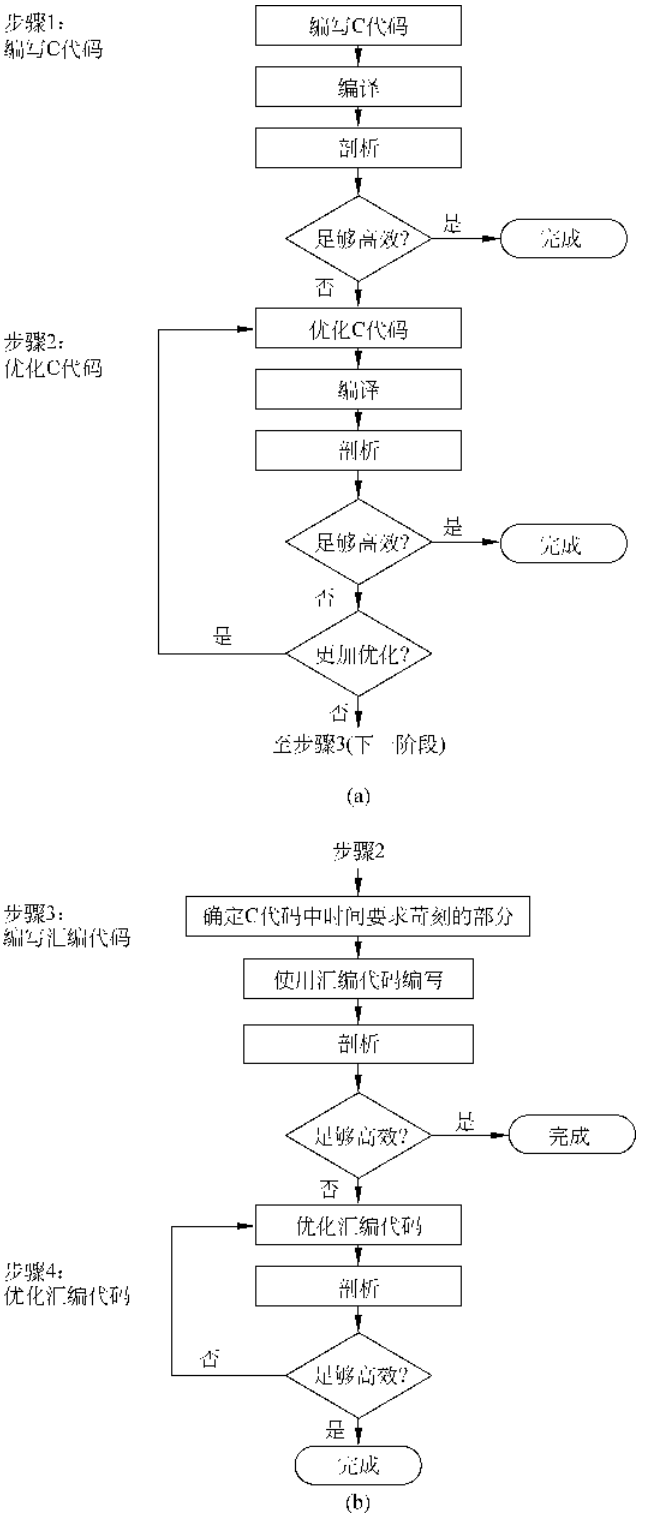
步骤1：
编写C代码
编写C代码
编译
剖析
足够高效?
是
完成
否
步骤2：
优化C代码
优化C代码
编译
剖析
足够高效?
是
完成
否
是
更加优化?
否
至步骤3(下一阶段)
(a)
步骤2
步骤3：
编写汇编代码
确定C代码中时间要求苛刻的部分
使用汇编代码编写
剖析
足够高效?
是
完成
否
优化汇编代码
步骤4：
优化汇编代码
剖析
否
足够高效?
是
完成
(b)

图 1-1　代码开发流程

步骤 1：编写 C 代码。用户可以使用兼容 ANSI 的 C55x 的 C 编译器来开发用户的 C 程序代码而不需要任何的 C55x DSP 知识。通过 CCS 确定 C 代码中可能存在的低效区域。在生成用户的函数代码后，可以通过选择更高级的优化编译器选项提高用户代码的性能。如果还没有达到用户所想要的性能，则进行第二步。

步骤 2：优化 C 代码。寻找代码中潜在的可修改的地方以使性能更佳。可用的方法包括：

- 使用特定的类型(register、volatile、const)。
- 修改 C 代码以更好地适应 C55x 的结构。
- 在可应用时运用 ETSI 函数特性。
- 使用 C55x 编译器内部函数。

修改完用户的代码后，再次使用 C55x 的性能分析工具来检查代码的性能。如果还没达到用户想要的效率，则进行第三步。

步骤 3：编写汇编代码。确定 C 代码中对时间要求比较苛刻的部分，将这段代码重写为 C 语言可调用的汇编语言函数。再次分析用户的代码，如果用户认为效率还不够高，则进行第四步。

步骤 4：优化汇编代码。在写完用户的汇编函数代码后，设法用一些方法来优化用户的汇编代码，这在第四章优化用户的汇编语言代码中将会进行介绍。这些方法包括：

- 并行放置指令。
- 重写或重组代码以避免流水线保护延时。
- 最小化取指令中的延迟。

1.2　基 本 教 程

本教程将带用户步入前一节介绍的代码开发流程，并介绍 TMS320C55x DSP 的编程基本概念。通过一步步的指令和代码举例向用户展示如何使用集成在 CCS 环境下的软件开发工具。

在开始本教程之前先安装 CCS 以便用户可以编辑、编译和调试 DSP 目标程序。如果想了解更多有关 CCS 特征的信息，可以参考 CCS 指南。用户可以在 CCS 软件环境下通过选择 help→tutorial 来访问 CCS 指南。

本教程中的举例使用的是助记符指令集，不过这些概念同样适合代数指令集。

1.2.1　引言

本教程提供了一个简单的汇编代码例子：将 4 个数相加(y＝x0＋x1＋x2＋x3)。这个例子帮助用户熟悉 C55x 编程的基本过程。

读完此教程后，用户应该了解：

- 4 种常用的寻址模式及何时使用它们。

❑ 开发和测试用户软件需要的基本的 C55x 工具。

本教程不能替代其他 C55x 文件中提到的信息，也不打算涵盖进行 C55x 高效编程所需的所有主题。关于 C55x 编程的更多信息，也可以参考 C55x 的 CCS 的在线帮助部分。

为了用户使用方便，所运行的例子可以从 http：//www. ti. com/sc/docs/schome. htm 下载。本章的例子可以在目录 55xprgug_srccode\tutor 中找到。

1.2.2 编写汇编代码

编写汇编代码时包括以下几个步骤：

❑ 为代码、常量、变量分配段。

❑ 初始化处理器模式。

❑ 建立寻址模式并将下列值相加：x0＋x1＋x2＋x3。

编写 C55x 汇编代码时要考虑以下规则：

❑ 标号。

标号的第一个字符必须是字母或是有字母跟着的下划线，并且必须从文本文件第一列开始。标号最多可以包含 32 个数字和字母的字符。

❑ 注释。

当前面加分号(;)时，注释可以从任何一列开始；当前面加星号(*)时，必须从第一列开始。

教程最终的汇编代码见例 1-1。代码执行 x 向量中元素相加的操作。此代码的段分配在用于创建此例的 3 个步骤中尤为突出。

例 1-1 tutor. asm 的最终汇编代码。

```
* 步骤 1: 段分配
* ------
      .def x,y,init
x     .usect "vars",4            ;为 x 保留 4 个未初始化的 16 位存储单元
y     .usect "vars",1            ;为 y 保留 1 个未初始化的 16 位存储单元

      .sect "table"              ;创建已初始化段"table",为 x 赋初始化值
init  .int 1,2,3,4

      .text                      ;创建代码段(默认为.text)
      .def start                 ;定义代码开始的标号
start

* 步骤 2: 处理器模式初始化
* ------
      BCLR C54CM                 ;设置处理器为'55x 固有模式而不是'54x 兼容模式(复位值)
      BCLR AR0LC                 ;设置 AR0 寄存器为线性模式
      BCLR AR6LC                 ;设置 AR6 寄存器为线性模式

* 步骤 3a: 使用间接寻址将初始化值复制到向量 x
* -------
copy
```

```
        AMOV #x,XAR0                ;XAR0 指向向量 x
        AMOV #init,XAR6             ;XAR6 指向初始化列表

        MOV *AR6+,*AR0+             ;复制从"init"开始到"x"
        MOV *AR6+,*AR0+
        MOV *AR6+,*AR0+
        MOV *AR6,*AR0

    * 步骤 3b: 使用直接寻址将向量 x 的元素的值相加
    * ------
    add
        AMOV #x,XDP                 ;XDP 指向向量 x,且通知汇编器
        .dp x
        MOV     @x,AC0
        ADD     @(x+3),AC0
        ADD     @(x+1),AC0
        ADD     @(x+2),AC0

    * 步骤 3c: 使用绝对寻址将结果写入 y
    * ------
        MOV     AC0,*(#y)

    end
        NOP
        B end
```

1.2.2.1　为代码、常量、变量分配段

编写汇编代码的第一步是为用户程序的不同段分配存储空间。

段就是由应用程序成功运行所需的代码、常量、变量组成的模块。这些模块在源程序中通过汇编伪指令来定义。在举例代码中,使用如下基本汇编伪指令来创建段和初始化值。

- .sect"section_name"为代码/数据创建已命名的初始化段。初始化段是定义了它们的初始化值的段。
- unsect"section_name"为数据创建已命名的未初始化段。未初始化段只声明它们的大小(以 16 位字为单位),但不定义它们的初始化值。
- .int value 在存储器中保留一个 16 位字,并定义初始化值。
- .def symbol 将符号设为外部文件可知的全局型,指出符号是在当前文件中定义的。外部文件通过.ref 标识可以访问这个符号。一个符号可以是一个标签或者是一个变量。

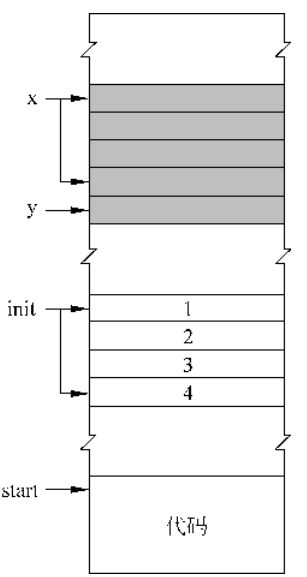

图 1-2　段分配

如例 1-2 和图 1-2 所示,文件 tutor.asm 包含 3 个段:

- vars 包含了 5 个未初始化的存储单元。

(1) 前 4 个单元为向量 x 保留(要相加的输入向量)。

(2) 最后一个单元 y,将要用来存储相加的结果。

- table 用来保存 x 的初始化值,init 标号指向段 table 的起始位置。
- .text 包含汇编代码。

例 1-2 给出用来分配段的部分汇编代码。

例 1-2 tutor.asm 的部分汇编代码(第一步)。

```
* 步骤 1: 段分配
*-----
    .def x,y,init
x   .usect "vars",4          ;为 x 保留 4 个未初始化的 16 位存储单元
y   .usect "vars",1          ;为 y 保留 1 个未初始化的 16 位存储单元

    .sect "table"            ;创建已初始化段"table",为 x 赋初始化值
init.int 1,2,3,4

    .text                    ;创建代码段(默认为.text)
    .def start               ;定义代码开始的标号
start
```

1.2.2.2 处理器模式初始化

第二步是确保设置状态寄存器(ST0_55、ST1_55、ST2_55、ST3_55)去配置处理器。可以设定这些值或是使用默认值。处理器复位后,寄存器内容为默认值。也可以在复位后确定默认寄存器值。

如例 1-3 所示:

- 使用位寻址模式修改状态寄存器位,寄存器 AR0 和 AR6 被设置为线性寻址(而不是循环寻址)。
- 将处理器设为 C55x 固有模式,而不是 C54 兼容模式。

例 1-3 tutor.asm 的部分汇编代码(第二步)。

```
* 步骤 2: 处理器模式初始化
*-----
    BCLR C54CM      ;设置处理器为'55x 固定模式而不是'54x 兼容模式(复位值)
    BCLR AR0LC      ;设置 AR0 寄存器为线性模式
    BCLR AR6LC      ;设置 AR6 寄存器为线性模式
```

1.2.2.3 建立寻址模式

代码中使用了 4 种最常用的 C55x 寻址模式:

- ARn 间接寻址(由 * 标出),可以将辅助寄存器(ARx)用作指针。
- DP 直接寻址(由@标出),提供相对于 DP 寄存器指定的基地址的正的偏移地址。偏移量由汇编器计算并由嵌于指令的 7 位值定义。
- k23 绝对寻址(由 # 标出),可以用标号指定全 23 位数据地址。

❑　位寻址(由位指令标出),可以修改存储单元或者 MMR 寄存器的某一位。

例 1-4 是对本节讨论的寻址方式的使用举例。

在步骤 3a 中,使用间接寻址,table 段的初始化值被复制到向量 x(执行加法的向量)。图 1-3 阐明了扩展辅助寄存器(XAR*n*)的结构。只在寄存器初始化时使用 XAR*n* 寄存器。后续的操作使用 AR*n* 是因为只有低 16 位受到影响(对 AR*n* 的操作限于 64K 字主数据页)。AR6 用来保存 table 的地址,AR0 用来保存 x 的地址。

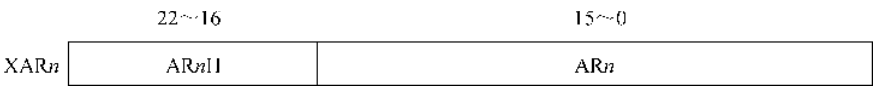

图 1-3　扩展辅助寄存器结构(XAR*n*)

在步骤 3b 中,使用直接寻址将 4 个值相加。注意,XDP 寄存器被初始化为指向变量 x。.dp 汇编伪指令用来定义 XDP 的值,因此,在编译时,汇编器可以计算出正确的偏移量。

最后,在步骤 3c 中,使用绝对寻址将结果存入 y 向量。绝对寻址提供了访问一个存储单元而不用改变 XDP 的简单方法,但是以代码长度增加为代价。

例 1-4　tutor.asm 的部分汇编代码(第三步)。

```
* 步骤 3a: 使用间接寻址将初始化值复制到向量 x
*------
copy
   AMOV #x,XAR0              ;XAR0 指向向量 x
   AMOV #init,XAR6           ;XAR6 指向初始化列表

   MOV *AR6+,*AR0+           ;复制从"init"开始到"x"
   MOV *AR6+,*AR0+
   MOV *AR6+,*AR0+
   MOV *AR6,*AR0

* 步骤 3b: 使用直接寻址将向量 x 的元素的值相加
*------
add
   AMOV #x,XDP               ;XDP 指向向量 x,且通知汇编器
   .dp x
   MOV @x,AC0
   ADD @(x+3),AC0
   ADD @(x+1),AC0
   ADD @(x+2),AC0

* 步骤 3c: 使用绝对寻址将结果写入 y
*------
   MOV AC0,*(#y)

end
NOP
B end
```

1.2.3 理解链接过程

链接器(lnk55.exe)为代码和数据段分配最终地址。这对代码的执行是必不可少的。指导链接器分配地址的文件称为链接器命令文件(tutor.cmd),在例 1-5 中给出。

- 链接器命令文件中给出的所有地址和长度使用字节地址和字节长度。这和 TMS320C54x 链接器命令文件不同,它使用 16 位字地址和字长度。
- MEMEORY 链接伪指令声明系统所有的物理存储器是可用的(例如,位于 0x100 处、长度为 0x8000 字节的 DARAM 存储块)。存储块不能重叠。
- SECTIONS 链接伪指令列出了输入文件中包含的所有段以及用户希望链接器给它们分配的位置。

当建立工程时,代码产生两个文件:tutor.out 和 tutor.map。为了查证 x、y 和 table 的地址,可以参阅例 1-6 中的 test.map 文件。注意,链接器给出了程序标号 start 和.text 的字节地址和数据标号 x、y 和 table 的 16 位的字地址。C55x DSP 使用字节寻址访问可变长度指令。指令可以为 1～6 个字节长。

例 1-5 链接器命令文件(tutor.cmd)。

```
MEMORY      /* 字节地址,字节长度 */
{
   DARAM: org = 000100h, len = 8000h
   SARAM: org = 010000h, len = 8000h
}

SECTIONS    /* 字节地址,字节长度 */
{
   vars: >DARAM
   table: >SARAM
   .text: >SARAM
}
```

例 1-6 链接器映射文件(test.map)。

```
*******************************************************************************
TMS320C55xx COFF Linker
*******************************************************************************
>>Linked Mon Feb 14 14: 52: 21 2000

OUTPUT FILE NAME:     <tutor.out>
ENTRY POINT SYMBOL: "start" address: 00010008

MEMORY CONFIGURATION
       name  org(bytes)  len(bytes)  used(bytes)   attributes  fill
       ----  ----------  ----------  -----------   ----------  ----
       DARAM 00000100    000008000   0000000a      RWIX
```

```
        SARAM 00010000    000008000   00000040      RWIX

SECTION ALLOCATION MAP

output                                                          attributes/
section  page  orgn(bytes)  orgn(words)  len(bytes)  len(words)  input sections
-------  ----  ----------   ----------   ----------  ---------   --------------
vars     0                  00000080                 00000005    UNINITIALIZED
                              80                       00000005  test.obj(vars)

table    0                  00008000                 00000004
                              8000                     00000004  test.obj
(table)
.text    0     00010008                 00000038
                  00010008                 00000037              test.obj
(.text)
                  0001003f                 00000001              --HOLE-- [fill
 = 2020]
.data    0                  00000000                 00000000    UNINITIALIZED
                              00000000                 00000000  test.obj
(.data)
.bss     0                  00000000                 00000000    UNINITIALIZED
                              00000000                 00000000  test.obj(.bss)

GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name

abs. value/
byte addr  word addr  name
---------  ---------  ----
           00000000   .bss
           00000000   .data
00010008              .text
           00000000   ____bss____
           00000000   ____data____
           00000000   ____edata____
           00000000   ____end____
00010040              ____etext____
00010008              ____text____
           00000000   edata
           00000000   end
00010040              etext
           00008000   init
00010008              start
           00000080   x
           00000084   y
```

```
GLOBAL SYMBOLS: SORTED BY Symbol Address

abs. value/
byte addr   word addr   name
---------   ---------   ----
            00000000    ___ end ___
            00000000    ___ edata ___
            00000000    end
            00000000    edata
            00000000    ___ data ___
            00000000    .data
            00000000    .bss
            00000000    ___ bss ___
            00000080    x
            00000084    y
            00008000    init
00010008                start
00010008                .text
00010008                ___ text ___
00010040                ___ etext ___
00010040                etext

[16 symbols]
```

1.2.4 建立程序

此时,用户应该已经成功安装了 CCS,并且选择了 C55x 仿真器作为 CCS 的配置驱动。现在可以选择 CCS 安装中用到的配置驱动。

在建立程序之前,用户必须设置工作环境并创建一个.pjt 文件。设置工作环境包括下列任务:

- 创建工程。
- 向工作区添加文件。
- 修改建立选项。
- 建立程序。

1.2.4.1 创建工程

创建一个名为 tutor.pjt 的新工程。

(1) 从 Project 菜单中选择 New 命令,输入工程名,如图 1-4 所示。

(2) 单击 Finish 按钮。

现在已经创建了一个名为 tutor.pjt 的工程并将之保存在新的 c:\ti\muproject\tutor 文件夹中。

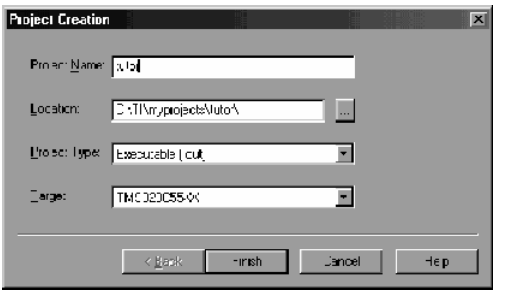

图 1-4　创建工程对话框

1.2.4.2　向工作区添加文件

把文件(tutor. asm 和 tutor. cmd)复制到 tutor 工程目录。

(1) 进入文件所在目录(55xprgug_srccode\tutor 目录),把文件复制到 c: \ti\muproject\tutor 目录。另外,用户也可以通过选择 File→New→Source File 创建自己的源文件,并输入本书中的例子代码。

(2) 把两个文件添加到工程 tutor. pjt。用鼠标右键选中 tutor. pjt,选择 Add Files,找到 tutor. asm 文件,选中它并单击 Open 按钮,如图 1-5 所示。以同样的方式把 tutor. cmd 添加到工程,如图 1-6 所示。

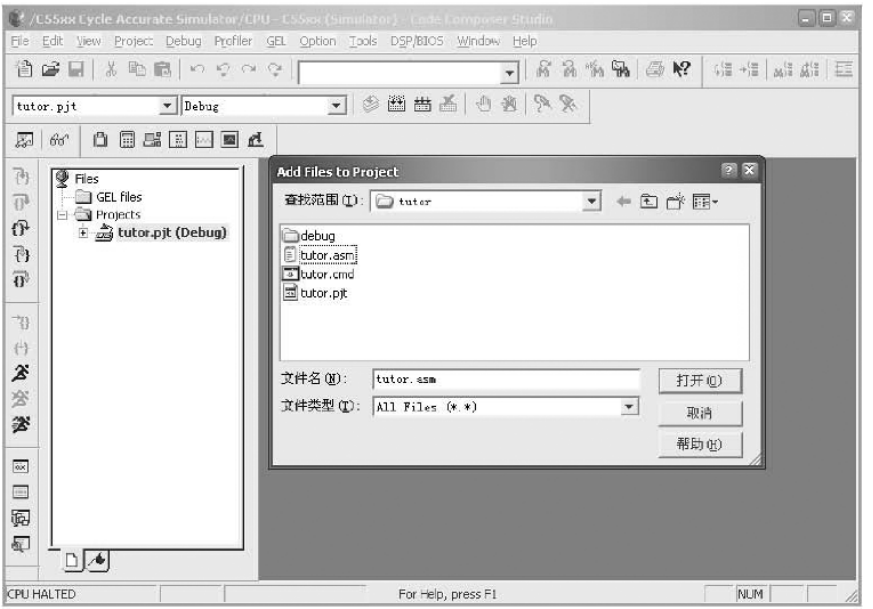

图 1-5　将 tutor. asm 添加到工程中

1.2.4.3　修改建立选项

修改链接选项。

(1) 从 Project 菜单选择 Build Options 命令。

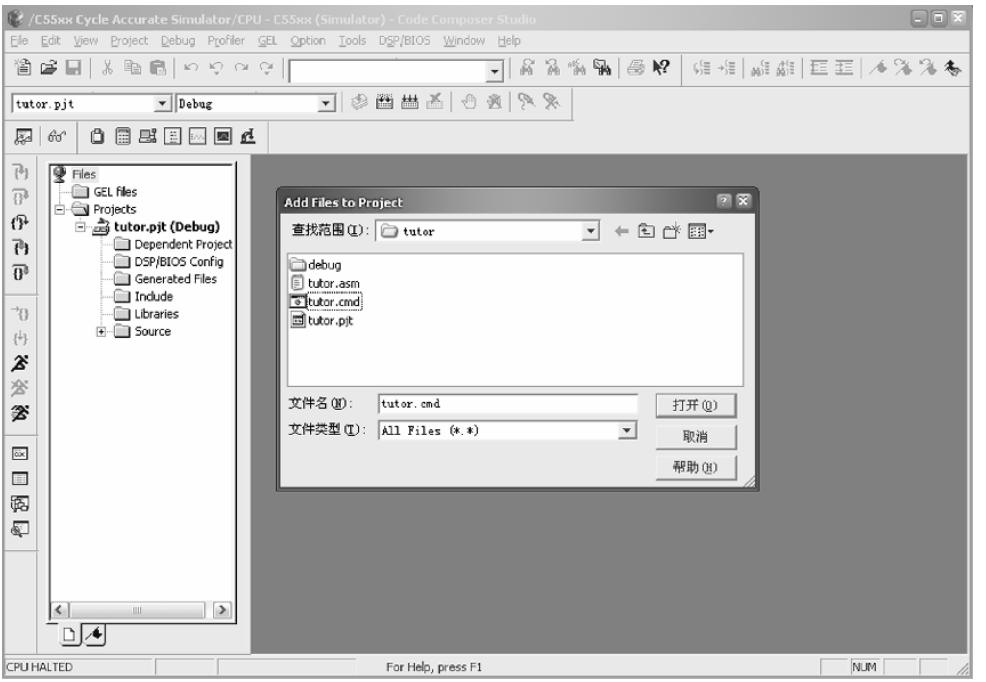

图 1-6　将 tutor.cmd 添加到工程中

(2) 选择 Linker 选项卡，如图 1-7 所示输入字段。

(3) 完成时单击 OK 按钮。

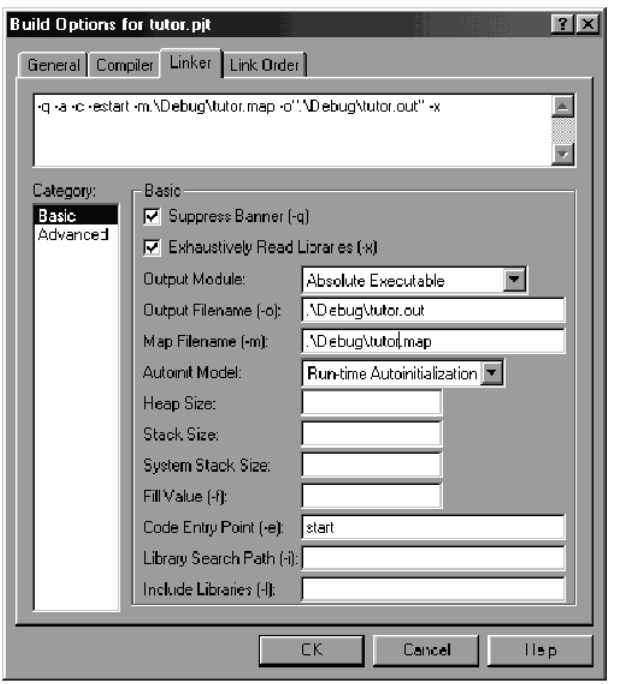

图 1-7　Build Options 对话框

1.2.4.4　建立程序

从 Project 菜单中选择 Rebuild All 命令。在重建过程完成后，屏幕会出现图 1-8 所示提示。

```
〈Linking〉
>> warning: entry point other than _c_int00 specified

Build Complete,
  0 Errors, 1 Warnings,0 Remarks.
```

图 1-8　重建完成界面

当建立工程时，CCS 一步完成对代码的编译、汇编和链接。汇编器读取汇编源文件，把 C55x 指令转换为它们对应的二进制编码。汇编处理的结果是产生一个标准 COFF 二进制格式的目标文件 tutor.obj。目标文件包含了所有的代码和变量，但是没有分配不同代码段的地址。这些分配发生在链接过程中。

因为工程中没有 C 代码，所以没有使用任何编译器选项。

1.2.5　测试代码

为了测试代码，应使用 C55x 仿真器检查代码执行情况。

装载 tutor.out。

(1) 从 File 菜单选择 Load program 命令。

(2) 找到 tutor.out(在\debug 目录)，单击 Open 按钮。

此时，由于在链接命令文件中定义了入口符号(-e start)，所以 CCS 会在 start 标号的起始处显示 tutor.asm 的源代码。否则，会显示复位向量所指的位置。

通过设置 Memory Window 选项来显示数组 x、y 和 init。

(1) 从 View 菜单中选择 Memory 命令。

(2) 在 Title 字段中输入 x。

(3) 在 Address 字段中输入 x。

(4) 对 y 重复(1)～(3)。

(5) 通过选择 View→Memory 来显示 init 数组。

(6) 在 Title 字段输入 Table。

(7) 在 Address 字段输入 init。

(8) 通过选择 View→CPU Registers→CPU Registers 来显示 AC0。

标签 x、y 和 init 对仿真器来说是可见的(使用 View→Memory)，因为它们被作为符号输出(在 tutor.asm 中使用.def 伪指令)。-g 选项用来使能汇编源程序调试。

现在，通过选择 Debug→Step Into 单步执行到 end 标号。检查 x 存储器窗口来确定 table 中的值被移到 x 中，并且 y 的值变为 0xa(1＋2＋3＋4＝10＝0xa)，如图 1-9 所示。

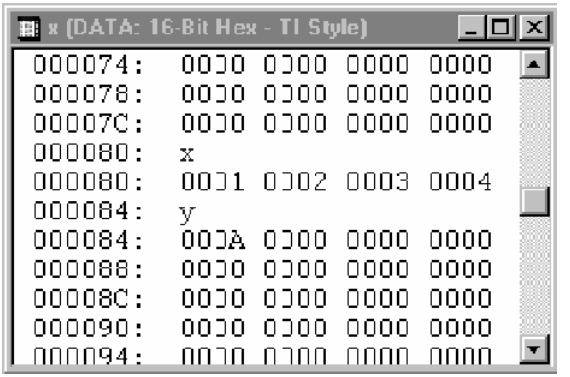

图 1-9 x 存储器窗口

1.2.6 代码计时

在确定代码能正确执行后,用户可以使用 CCS 来计算代码执行所需的周期数。

重新装载代码:从 File 菜单中选择 Reload Program 命令。

使能分析时钟:

(1) 从 Profiler 菜单中选择 Enable Clock 命令。

(2) 从 Profiler 菜单中选择 View Clock 命令。

设置断点:

(1) 选择 tutor.asm 窗口。

(2) 在要计算时间的代码的起始处设置一个断点(start 后的第一条指令):在 copy 标号的下一条指令上右击选择 Toggle Breakpoint 命令。

(3) 在末尾设置一个断点:在 end 标号下一条指令上右击选择 Toggle Breakpoint 命令。

代码计时:

(1) 通过选择 Debug→Run 执行到第一个断点。

(2) 双击 Clock 窗口清除周期数。

(3) 通过选择 Debug→Run 执行到第二个断点。

(4) Clock 窗口显示在两个断点之间的代码执行的周期数,本例中近似为 17。

1.3 定点算术运算

TMS320C55x DSP 是一款 16 位的定点处理器。本节介绍了使用 DSP 进行标准和扩展精度的定点运算的重要的方法和注意事项。提供了汇编语言代码的例子来阐明这些概念。

1.3.1 定点运算指南

数字信号处理器已经发展成为处理那些需要大量运算的复杂算法的工具。DSP 可以划分为两种:浮点 DSP 和定点 DSP。

典型的浮点 DSP 使用由 24 位尾数和 8 位指数组成的 32 位字，可以提供从 $2^{-127}\sim2^{128}(1-2^{-23})$的动态范围。浮点设备中如此大的动态范围意味着在设计中可能会忽略动态范围的限制。浮点设备比定点设备更加昂贵而且功耗更大。

定点的 DSP，如 TMS320C55x DSP，通常使用 16 位字。这些定点设备使用更少的硅元件，比浮点设备造价低且功耗小。因为定点算术的规则和动态范围的限制，设计者必须在定点 DSP 系统的设计中发挥更大的作用。设计者需要决定 16 位字表示的是整数还是分数，在需要时还要使用定标系数并且对可能的寄存器溢出进行保护。

1.3.1.1　补码

在二进制中，一个数可以被表示成一个带符号的数，最左端的位表示符号位，剩下的位表示数值如：

$$+52 = 0011\ 0100\text{b}$$
$$-52 = 1011\ 0100\text{b}$$

以上表示方法没有用在 DSP 当中，因为加法的算法会因两个加数的符号位的不同而不同。DSP 使用补码的数据格式，在该格式中，正数表示为简单的二进制数，而负数表示为所有位取反后加 1。

例 1-7 显示了十进制数 353 的 16 位二进制带符号数的形式。每个位的位置表示 2 的幂次，最低位表示 2^0，最高位表示 2^{15}。二进制的 0 和 1 决定了 2 的幂次是如何加权的(乘以 0 或乘以 1)，所有位加权求和后得到 353。因为该数是带符号的，2^{15}被给定负号。例 1-8 给出了如何计算一个负数的补码。

例 1-7　带符号补码二进制数扩展为等效的十进制数。

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	1

$$=(0\times(-2^{15}))+(0\times2^{14})+(0\times2^{13})+(0\times2^{12})+(0\times2^{11})+(0\times2^{10})+(0\times2^9)+(1\times2^8)+(0\times2^7)+(1\times2^6)+(1\times2^5)+(0\times2^4)+(0\times2^3)+(0\times2^2)+(0\times2^1)+(1\times2^0)$$
$$=(1\times2^8)+(1\times2^6)+(1\times2^5)+(1\times2^0)$$
$$=256+64+32+1=353$$

例 1-8　计算一个负数的补码。

从正的二进制数开始(十进制的 353)：		0000 0001 0110 0001
所有位取反，得到反码：		1111 1110 1001 1110
加 1 得到补码：	+	1
结果：负的二进制数(十进制的−353)：		1111 1110 1001 1111

1.3.1.2　整数与分数

在 DSP 编程中，最常用的数据格式是整数和分数。在信号处理中，分数的表示更加常见。两个分数相乘，结果仍然是分数。因此不会发生乘法溢出。但要注意的是，分数相加有可能发生加法溢出。

图 1-10 表示了如何把补码数理解为整数。给最高位一个负的权值，该整数就等于所有可利用的位的加权求和值。若某一位是 1，则其权值包含到求和中；若为 0，则该权值不可用

(即其有效权值为 0)。为了简单起见,图 1-10 给出的是 4 位的二进制值;但是,这个概念是易于扩展到大的二进制数值的。比较图 1-10 中的 4 位的数和图 1-11 中的 8 位的数,可以看出,二进制整数的最低位的位权值总是 1,而位权值的绝对值向着最高位增大。向二进制整数的左边添加位不改变原二进制码的绝对位权值。

4位的2的补码的二进制整数	MSB			LSB	
位权值	$-2^3=-8$	$2^2=4$	$2^1=2$	$2^0=1$	
最小正数值	0	0	0	1	0−0+0+1 1
最大正数值	0	1	1	1	=0−4+2+1=7
最小负数值	1	1	1	1	=−8 1 4 1 2 1 1=−1
最大负数值	1	0	0	0	=−8+0+0+0=−8
其他例子	0	1	0	1	=0 4 1 0 1 1=5
	1	1	0	1	=−8+4+0+1=−3

图 1-10 4 位补码整数表示

MSB							LSB
$-2^7=-128$	$2^6=64$	$2^5=32$	$2^4=16$	$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$

图 1-11 8 位补码整数表示

图 1-12 表示了如何把补码数理解为分数。这个概念与图 1-10 所示的很相似,只不过其位权值是分数,且该分数的绝对值不大于 1。比较图 1-12 中的 4 位的数和图 1-13 中的 8 位的数,可以看出,二进制分数的最高位的位权值总是−1,而位权值的绝对值向着最低位减小。与向二进制整数的左边添加位不同,向二进制分数的左边添加位会改变原二进制码的位权值。

4位二进制分数	MSB			LSB	
位权值	$-2^0=-1$	$2^{-1}=1/2$	$2^{-2}=1/4$	$2^{-3}=1/8$	
最小正值	0	0	0	1	0−0+0+1/8 1/8
最大正值	0	1	1	1	=0−1/2+1/4+1/8=7/8
最小负值	1	1	1	1	=−1 1 1/2 1/4 1 1/8=−1/8
最大负值	1	0	0	0	=−1+0+0+0=−1
其他例子:	0	1	0	1	=0 1/2 1 0 1 1/8=5/8
	1	1	0	1	=−1+1/2−0−1/8=−3/8

图 1-12 4 位补码分数表示

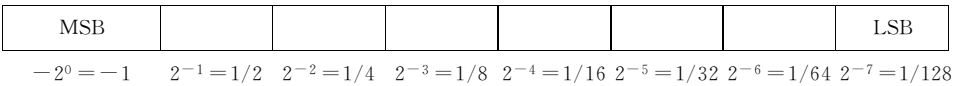

图 1-13　8 位补码分数表示

1.3.1.3　补码的算法

补码格式的一个显著优点是对于所有数值，加法的算法是一致的。为了熟悉补码的算法，请参照本节的例子。

例 1-9 给出了补码加法的例子。这些二进制操作是完全独立于程序员把这些数值转换为十进制数的转换方法的。这里给出整数加法和分数加法各一例。为简单起见，例子使用了 8 位二进制数值，这是易于扩展到更大的二进制数值的。

例 1-9　补码二进制数值的加法。

例 1-10 给出了减法的例子。在例 1-9 中，分别给出了整数和分数的加法表示。需要注意的是，补码减法等同于一个正数和一个负数相加。第一步是求出减数的补码，第二步是用这个负数进行加法计算。

补码加法	整数表示	分数表示
1　（进位） 0000　0101 + 0000　0100 0000　1001	5 + 4 9	5/128 + 4/128 9/128
1　1 1　（进位） 0000　0101 + 0000　1101 0001　0010	5 + 13 18	5/128 + 13/128 18/128

例 1-10　补码二进制数的减法。

补码减法	整数表示	分数表示
原来的形式： 0000　0101 − 0000　0100	5 − 4	5/128 − 4/128
减数的补码： 11（进位） 1111　1011 + 1 1111　1100		
加法的形式： 11111　1　（进位） 0000　0101 + 1111　1100 0000　0001 （忽略最后的进位位）	5 + (−4) 1	5/128 + (−4/128) 1/128

续表

补码减法	整数表示	分数表示
原来的形式： 0000 0101 − 0000 1101	5 − 13	5/128 − 13/128
减数的补码： 1111 0010 + 1 1111 0011		
加法的形式： 111（进位） 0000 0101 + 1111 0011 1111 1000	5 + (−13) −8	5/128 + (−13/128) −8/128

例 1-11 给出了补码的乘法的例子。为简单起见，该例子使用了 4 位数与 4 位数相乘，并假设结果放入 8 位的累加器中。注意，7 位的运算结果带符号扩展，填满了累加器。C55x DSP 以这种方式进行乘法结果的带符号扩展，把结果扩展到 32 位或 40 位。可以在例 1-11 整数和分数的乘法中看到这种带符号扩展的效果。整数不因扩展而改变，但分数却可能因扩展而发生错误。带符号扩展增加了一个额外的符号位。如果用户的程序假定结果的最高位是唯一的符号位，必须把结果左移一位来去掉额外的符号位。在 C55x DSP 中，有一个称为 FRCT 的控制位自动地进行该移位操作。当 FRCT＝1 时，DSP 自动在乘法操作之后左移一位。用户可以通过以下指令清除 FRCT 位：

```
BCLR FRCT       ;清除 FRCT
BSET FRCT       ;设置 FRCT
```

例 1-11 补码二进制数的乘法。

补码乘法		整数表示	分数表示
0100 ×1101	被乘数 乘数	4 ×(−3)	4/8 ×(−3/8)
0000100 000000 00100 1100	（见注意事项）		
1110100	7 位算术结果	−12	−12/64 （结果的最高位是唯一的符号位）
11110100	在累加器中的 8 位算术结果带符号	−12	−12/64
注意事项：因为最高位是符号位，所以最高位的部分乘积就是被乘数的补码。			如果正确地理解，结果是 −12/64；如果不正确地理解，结果是 −12/128；左移一位，以去掉最高位的额外的符号位。

1.3.2　扩展精度的加法和乘法

定量分析、浮点运算以及其他的操作可能需要超过 32 位精度的算术操作。因为 C55x 设备是 16/32 位的定点处理器，所以在进行超过 32 位精度的算术操作的时候需要软件的支持。这些算术函数是分部分完成的，与常规算术操作类似。

C55x DSP 具有如下一些使扩展精度的运算更加高效的特性。

- 进位位：其中一个特性是进位状态位，该状态位受大部分的 D 单元 ALU 指令影响，也受旋转和移位操作影响。进位位取决于 M40 状态位的设置。当 M40＝0 时，进位/借位在位 31 检测。当 M40＝1 时，进位/借位在进位位中的反映情况是在位 39 检测的。用户的代码也可以通过装载 ST0_55 或使用状态位清除/置位指令来修改进位位。为了进行正确的扩展精度的算术运算，应该清除饱和状态位(SATD＝0)以防止累加器在运算过程中的饱和。
- 32 位加法、减法和装载：两条 C55x 数据总线 CB 和 DB，允许某些指令在单周期内进行 32 位操作。常字装载和双精度加/减法指令使用 32 位的操作数并且可以高效地完成扩展精度的算术运算。
- 16 位带符号/无符号乘法：硬件乘法器可以进行 16 位带符号/无符号数值的乘法，也可以进行两个带符号数值和两个无符号数值的乘法。这样可以使 32 位乘法操作效率更高。

1.3.2.1　64 位加法举例

例 1-12 是进行两个 64 位数值相加，得到一个 64 位的结果。以下是代码的一些主要部分：

- 64 位加法的部分求和由以下的指令高效地执行，这些指令在单周期内处理 32 位操作数。

  助记符指令：
  ```
  MOV40 dbl(Lmem),ACx
  ADD dbl(Lmem),ACx
  ```
  代数指令：
  ```
  ACx = dbl(Lmem)
  ACx = ACx + dbl(Lmem)
  ```
- 对于部分求和的高位部分，以下的指令使用了由 32 位部分求和的低位部分产生的进位位。每个部分求和的结果使用 MOV ACx，dbl(Lmem)或 dbl(Lmem) = ACx 保存在两个存储器位置中。

  助记符指令：
  ```
  ADD uns(Smem),CARRY,ACx
  ```
  代数指令：
  ```
  ACx = ACx + uns(Smem) + CARRY
  ```
- 通常，如果运算不产生进位，进位位就被清除。如图 1-14 所示，带有 16 位移位的 ADD 指令是例外的情况，因为它只能对进位位进行置位。如果该指令不产生进位，进位位也保持不变。这样就允许 D 单元 ALU 在加上累加器的高位部分或低位部分而导致进位时，会产生合适的进位。

  助记符指令：
  ```
  ADD Smem << #16,ACx,ACy
  ```

代数指令：ACy = ACx + (Smem << #16)

图 1-14 说明了一些 32 位加法对进位位的正常影响(如图中的 C 所示)。

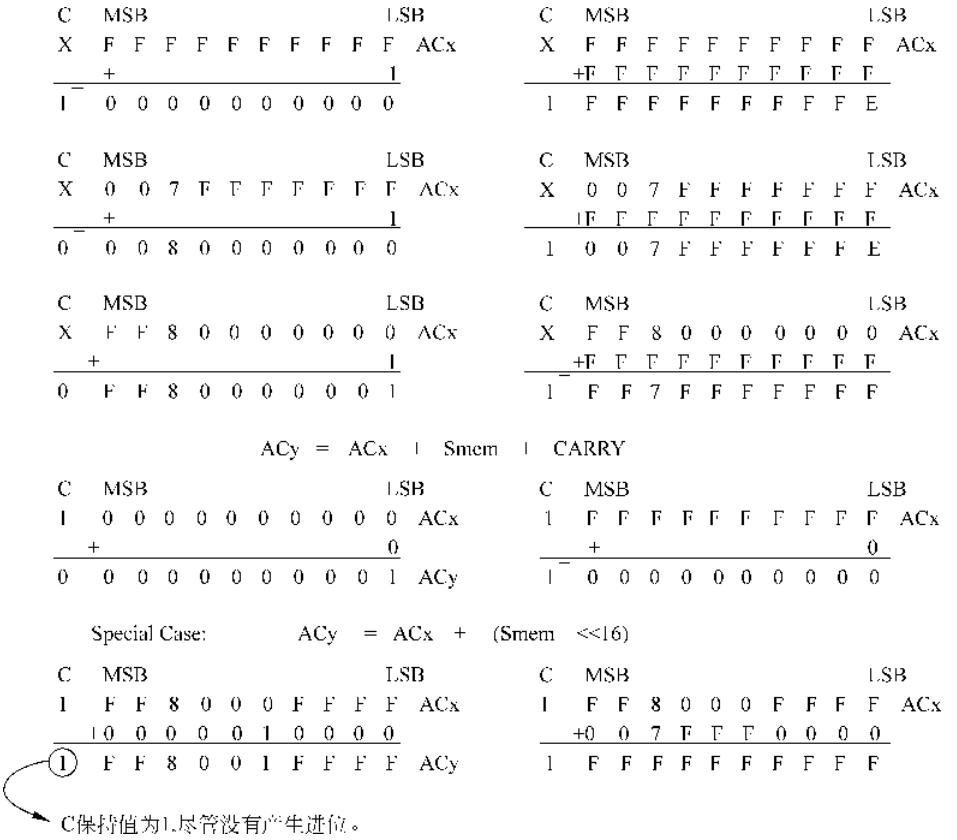

图 1-14 加法操作对进位位的影响

例 1-12 64 位加法。

```
; ****************************************************************************
;     64 位加法                指针分配：
;
;     X3 X2 X1 X0      AR1 ->  X3 (偶地址)
;   + Y3 Y2 Y1 Y0              X2
;   -------------              X1
;     W3 W2 W1 W0              X0
;                      AR2 ->  Y3 (偶地址)
;                              Y2
;                              Y1
;                              Y0
;                      AR3 ->  W3 (偶地址)
;                              W2
;                              W1
;                              W0
;
```