User's Guide

C29 CPU 上的应用软件优化

摘要

本用户指南介绍使用 C29 CPU 优化应用性能的具体技术。

内容

商标

所有商标均为其各自所有者的财产。

English Document: SPRUJG0

1 引言

本指南介绍使用 C29 CPU 优化应用性能的具体技术。提倡的方法涵盖编译器设置、存储器配置、代码构建和配置，以及最后的应用级优化。

2 性能优化

2.1 编译器设置

本节讨论了影响性能的关键编译器设置。

2.1.1 启用调试和源代码交叉列

在初始开发过程中，建议使用 -g compiler 选项生成调试信息。然后，使用以下命令，输出可执行文件可以生成含有交叉列出源代码的反汇编文件。有关更多信息，请参阅开发流程差异。

```
c29objdump --disassemble -S <>.out > <>.cdis
```

2.1.2 优化控制

建议使用 -O3 来提高速度，特别是对于循环的软件流水。出于调试目的，可以使用“optnone”属性有选择地关闭某些函数的优化。

```
__attribute__((optnone))
void foo()
{
   ..
}
```

2.1.3 浮点数学

-ffast-math 是建议用于浮点计算的编译器选项。此选项是多个选项的集合，其中两个显著提高性能的选项是 -fapx-funcs 和 -freassoc。这使编译器能够对浮点数学做出积极假设，让精度只会出现有限的损失。有关控制浮点行为的详细信息，请参阅 Clang 编译器用户手册和 C29 Clang 编译器工具用户指南。有关 -ffast-math 的详细信息，请参阅 Clang 编译器用户手册和 C29 Clang 编译器工具用户指南。

使用 -ffast-math 时，编译器会将对许多标准 RTS 库函数的调用替换为相应的 TMU 指令。TMU 内置在 C29 CPU 中。

使用 C '/' 运算符的单精度浮点除法是使用 PREDIVF、SUBC4F（7 次）和 POSTDIVF 指令实现的。使用 C '/' 运算符的双精度浮点除法是使用 PREDIVF、SUBC3F（19 次）和 POSTDIVF 指令实现的。借助-ffast-math 编译器选项，单精度浮点除法是使用 DIVF（估算分母的倒数并与分子相乘）指令实现的。

2.1.4 定点除法

在有符号或无符号 32 位或 64 位整数除法中，C '/' 运算符是由编译器使用必要指令来实现的。支持三种类型的除法 — 传统除法、欧几里德除法和取模除法。C 标准和编译器本身支持传统除法，其中余数具有分子的符号。在取模（或取整）除法中，余数具有分母的符号。欧几里德除法是控制运算的首选，其中商对于 0 是线性的，余数始终为正。

备注

要实现欧几里德除法或模数除法，需要使用内联函数。有关更多信息，请单击此处。

2.1.5 单精度与双精度浮点

如果 FPU64 可用（F29H85x 上的 CPU3），则可以高效执行双精度浮点运算。要使用 FPU64，请使用编译器选项：

```
-mfpu=f64
```

在 C29 上，EABI 是唯一受支持的可执行格式。不支持 COFF。对于 EABI，双精度浮点类型为 64 位。根据 C 标准，未带后缀 'f' (1.54f) 而直接使用的常量 (1.54) 的用户代码会被解析为双精度类型。这导致其他相关变量隐式转换为双精度类型，当 FPU64 不可用时（F29H85x 上的 CPU3），这会对性能产生负面影响。

发生上述情况时，使用以下编译器选项生成警告：

```
-wdouble-promotion
```

或者，可以使用以下编译器选项将浮点常量限制为单精度。这样就无需在每个浮点常量后添加一个“f”。但是，采用这种方法时，所有浮点常量都变成单精度常量。

```
-cl-single-precision-constant
```

2.1.6 链接时优化 (LTO)

从版本 2.0.0.STS 开始，编译器工具可在链接时对整个程序进行模块间优化。此功能通常称为*链接时优化 (LTO)*。这可以显著提升性能。如需更多有关 LTO 的信息，请单击此处。

备注

为使应用程序中包含的库能够参与 LTO，每个库都必须使用 -flto 编译器选项进行构建。这会将库设置为在链接期间支持 LTO。

2.2 存储器设置

本节讨论会影响性能的关键存储器设置。

2.2.1 从 RAM 执行代码

要了解哪些 RAM 对 C29 CPU 的程序代码具有 0 等待状态访问，请参阅 *F29H85x 和 F29P58x 实时微控制器技术参考手册* 中的*存储器子系统(MEMSS)* 一章。例如，CPU1 和 CPU2 对 LPAx RAM 上的程序代码具有 0-WS 访问权限。CPU1 和 CPU3 对 CPAx RAM 上的程序代码具有 0-WS 访问权限。

需要从 RAM 执行的函数可置于 RAM 部分，并且链接器命令文件可用于控制在启动时将此函数复制到 RAM。更多相关信息，请单击此处。

```
Source file:
__attribute__((section("ramfunc"), noinline)) void foo() {.. }
```

```
Linker command file:
ramfunc : load=FLASH, run=RAM, table(BINIT)
```

2.2.2 从闪存执行代码

要了解根据 CPU 时钟频率所需的存储器等待状态次数，请参阅 *F29H85x 和 F29P58x 实时微控制器数据表* 的*存储器参数*部分。此外，确保启用预取、预读取和缓存。Flash_initModule() 可用于执行以下运算：

```
voidFlash_initModule(uint16_twaitstates)
{
   ..
   // Set waitstates according to frequency
   Flash_setWaitstates(waitstates);
   ..
   // Enable data cache, code cache, prefetch, and data preread to    improve performance of code//
executed from flash.
   Flash_configFRI(FLASH_FRI1, FLASH_DATAPREREAD_ENABLE | FLASH_CODECACHE_ENABLE |
FLASH_DATACACHE_ENABLE | FLASH_PREFETCH_ENABLE);
   ..
}
```

2.2.3 数据放置

要了解哪些 RAM 对 C29 CPU 程序数据具有 0 等待状态访问权限，请参阅 *F29H85x 和 F29P58x 实时微控制器技术参考手册* 中的*存储器子系统 (MEMSS)* 一章。例如，CPU1 和 CPU2 对 LDAx RAM 上的程序数据具有 0 等待时间访问权限。CPU1 和 CPU3 对 CDAx RAM 上的程序数据具有 0 等待时间访问权限。

对 RAM 的并行访问可能导致仲裁，并且当它们发生在同一个 RAM 块（LDAx、CDAx - 每个“x”对应不同的 RAM 块）上时，会导致停滞。编译器会尽可能尝试执行并行加载：

```
LD.32    M2,*(ADDR2)(A7++)
||LD.32    M3,*(ADDR2)(A4+A0<< 2)
```

为避免停滞，请确保对不同的块进行访问。例如，对于 FIR 滤波器，并行加载滤波器系数和历史缓冲器值的情况就可能发生。将每个组件都放入自己的 RAM 块。

2.3 代码结构和配置

本节讨论了会影响性能的代码构造和配置。

2.3.1 内联

通过消除极小函数的函数调用和返回的开销，从而允许编译器在函数调用周围代码的上下文中执行优化，内联可以带来性能优势。对于那些只调用几次的大型函数，这也可能是有益的。

要启用内联，编译器需要优化级别达到 -O1 或更高（达到 -O0，属性可以强制内联），并且需要能够在编译时查看函数的定义。因此，可以内联对同一源文件中定义的函数的调用，以及头文件中使用“static”定义的函数，其中头文件包含在源文件中。

备注

链接时优化 (LTO) 使编译器能够内联源文件中定义的函数，这些源文件与调用它们的源文件不同。

2.3.2 内联函数

当使用 -ffast-math 编译器选项时，C29 包含的指令支持高效实现许多标准 RTS 函数。编译器还支持与这些指令相对应的内置函数或内联函数。这 F29-SDK 在 examples/rtlibs/fastmath/tmu 中提供了示例，说明如何使用这些内联函数。支持的示例包括 asinf()、acosf()、atan2f()、ceilf()、cosf()、divf()、expf()、floorf()、fmodf()、roundf()、sinf()、truncf()。

- 此外，编译器支持使用 ISQRTF 指令实现 sqrtf() 和 1/sqrtf()。相应的内联函数如下面的代码块中所示。

```
float __builtin_c29_i32_isqrtf32_m(float x);
```

- IEXP2F 的内联函数如下面的代码块中所示。

```
float __builtin_c29_i32_iexp2f32_m(float f0);
```

备注

由于 IEXP2F 在基值或指数较大时存在精度限制，因此编译器不支持通过 IEXP2F 实现 expf()、exp2f()、1/expf() 或 1/exp2f()。但是，F29x-SDK 在 examples/rtlibs/fastmath/tmu/ccs/expf_example 中包含了此情况的示例，展示如何使用有一定输入范围的内联函数。指令（和内联函数）对一定范围的基值/指数值仍然准确且有用。

- 编译器支持通过 LOG2F 指令进行对数计算。此外，LOG2F 的内联函数如下面的代码块中所示。其可用于实现 logf()、log2f() 和 powf()。

```
float __builtin_c29_i32_log2f32_m(float f0);
```

- atanf() 可以通过 PUATANF 实现，使用内联函数 float __builtin_c29_i32_puatanf32_m(float f0)

```
Example:
// x is per-unit in [-1,1]
// y is per-unit in [-0.125, 0.125] i.e. [-pi/4, pi/4] radians
y = __builtin_c29_i32_puatanf32_m(x);
```

- 编译器支持通过 PUATANF 和 QUADF 指令进行 atan2f() 计算。此外，可使用内联函数 float __builtin_c29_i32_puatanf32_m(float f0) 和 float __builtin_c29_quadf32(unsigned int * tdm_w_uip0, float * rw_fp1, float * rw_fp2) 实现 atan2f()

```
Example:
test_output =puatan2f32(y_input,x_input);

static inline float32_t puatan2f32(float32_t y, float32_t x)
{
   uint32_t flags;
   return __builtin_c29_quadf32(&flags, &y, &x) +      __builtin_c29_i32_puatanf32_m(y / x);
}
```

2.3.3 易失性变量

变量的“Volatile”关键字会向编译器指明，该变量可以由已知程序流程以外的程序（例如 ISR）修改。这可确保编译器完全按照 C/C++ 代码中编写的内容保留对全局变量进行读写的次数，而不会消除冗余读取、写入或重新排序访问。访问表示存储器映射外设的存储器位置时，必须使用 Volatile 关键字。

备注

仅在绝对需要的情况下才建议对变量使用 Volatile 关键字，如在 ISR 和映射外设的存储器内部更新的变量。使用易失性数据类型时，可以使用局部变量进行中间计算而不直接引用易失性数据结构，从而提高性能。

2.3.4 函数参数

当指针作为函数参数传递时，在指针上使用“Restrict”关键字可以提高性能。通过对指针 **p** 的类型声明应用限制，编程器向编译器提供以下内容：

在 **p** 的声明范围内，只有 **p** 或基于 **p** 的表达式用于访问 **p** 指向的对象。

编译器可以利用这一点来生成更高效的代码。

```
Example:
void matrix_vector_product(float32_t *restrict A, float32_t *restrict b, int nr, int nc, float32_t
*restrict c)
{
   int i, j;
   float32_t s;
   for(i = nr -1; i >=0; i--)
   {
     s =c[i];
     for(j = nc -1; j >=0; j--)
     {
        s = s +A[j*nr+i]*b[j];
     }
     c[i] = s;
    }
}
```

备注

在将结构作为函数参数传递时，传递结构指针而不是结构成员会提高性能。

2.3.5 启用更广泛的数据访问

嵌入式系统中的许多操作涉及对存储器的连续数据访问（读取或写入）。由于 F29x 上的数据总线宽度为 64 位，此架构允许 64 位数据读取和写入。然而，大多数用户代码限制为 32 位数据，因此访问也限于 32 位。在某些情况下，尤其是数组访问，通过将代码重写为执行 64 位访问而非 32 位访问，可以实现显著性能提升。例如，下面的第一个代码块表示通过 32 位访问执行简单的内存缓冲区读取操作。

```
uint32_t mem_read_16k_cn(uint32_t *src)
{
    uint32_t i =0;
    uint32_t x =0;
    for (i=0;i<LEN_16K;i++)
    {
        x +=*src++;
    }
    return x;
}
```

下一个代码块表示使用 64 位访问实现的相同操作，其效率是原来的两倍。

```
uint32_t mem_read_16k_opt(uint32_t *src)
{
    uint32_t i =0;
    uint64_t *s2 = (uint64_t*) src;
    uint32_t x =0;
    uint32_t x2 =0;
    for (i=0;i<LEN_16K>>1;i++)
    {
        uint64_t temp =*s2++;
        x += (temp>>32);
        x2 += (temp&0xFFFFFFFF);
    }
    return x+x2;
}
```

在许多情况下，编译器无需显式重写代码，而是通过将特定属性应用于底层数据，即可提高性能。如果数组已对齐，请使用 __attribute__((aligned(val)) 向编译器指示这样做。例如，数组上的 __attribute__((aligned(8)) 指示 8 字节对齐，并允许编译器一次加载 64 位而不是 32 位。这可以为矩阵乘法等运算带来性能优势。如果数组是全局数组并在函数中直接访问，则上述对齐规范就足够了（请参阅下面的第一个代码块）。另一方面，如果对齐数组作为指针传递到函数中，则需要将 __builtin_assume_aligned 属性应用于指针，以告知编译器该对象的对齐情况（请参阅下面的第二个代码块）。

```
__attribute__((aligned(8))) float A[100];
__attribute__((aligned(8))) float B[100];
__attribute__((aligned(8))) float C[100];
void matrix_mpy(void)
{
```

```
int32_T i;
int32_T i_0;
int32_T i_1;
for (i_0 =0; i_0 <10; i_0++)
{
    for (i =0; i <10; i++)
    {
        int32_T C_tmp, tmp;
        C[10* i_0 + i] =0.0f;
        for (i_1 =0; i_1 <10; i_1++)
        {
            C[10* i_0 + i] +=A[10* i_1 + i] *B[10* i_0 + i_1];
        }
    }
}
```

```
void matrix_mpy_f32_4by4(float (*restrict Ma_f32) [4], float (*restrict Mb_f32)[4], float
(*restrict Mc_f32)[4])
{
int32_t i,j,k;
// Use __builtin_assume_aligned to inform the compiler about alignment Ma_f32 =
__builtin_assume_aligned(Ma_f32, 8);
Mb_f32 = __builtin_assume_aligned(Mb_f32, 8);
Mc_f32 = __builtin_assume_aligned(Mc_f32, 8);
..
}
```

备注

目前，编译器无法根据对象的物理地址（即仅基于位置属性）区分对齐。

备注

这种“矢量化”（即将较小数据类型的访问合并到较大数据类型）也可以使用 8 位和 16 位数据进行，编译器的未来版本已计划纳入该功能。

2.3.6 自动代码生成工具

使用 Mathworks Embedded Coder 等自动代码生成工具进行性能优化是一个重点领域。如需详情，请单击此处。

根据配置设置，生成的代码可能包含双精度浮点运算，而当其在仅支持单精度浮点运算的硬件上运行时，可能会导致性能大幅下降。建议用户在生成的代码中查找以下内容：

- C 代码中任何意外的双精度浮点常量。这些常量可能是未带后缀“f”的浮点数。它们有时也具有特定名称，例如“DBL_EPSILON”。
- 编译器生成的汇编代码中出现的双精度运算。这些运算可能有特定的名称，例如“CALLD @__extendsfdf2”（双精度乘法）或“CALLD @__muldf3”（双精度乘法）。

2.3.7 准确剖析代码

用户可以使用不同的剖析技术对其应用程序代码进行基准测试。启用优化后，编译器可以重新排序操作并执行内联等。这有时会使用户难以准确了解进行剖析的内容，以及进行剖析的内容是否确实是用户想要剖析的内容。

“__builtin_instrumentation_label()”是一个非常有用的标签，可以在需要进行剖析的代码前后使用（请参阅下面的代码块）。但是，当启用优化时，它并不是完美的代码屏障。除了将代码块轮廓化为自己的函数、将代码块标记为非内联以禁用内联以及调用该函数之外，C29 编译器没有真正的代码移动障碍。

```
// Example 1
__builtin_instrumentation_label("profiling_start");
function1();
__builtin_instrumentation_label("profiling_stop");

// Example 2
__builtin_instrumentation_label("profiling_start");
// code being profiled
..
..
__builtin_instrumentation_label("profiling_stop");
```

2.4 应用代码优化

本节讨论会影响应用性能的应用代码及其配置。

2.4.1 SDK 优化库

使用 F29x SDK 中提供的优化库和源代码。这些代码包含许多标准控制、DSP 和数学运算的理想实现。其中一些实现（FFT、FIR）是以汇编语言编写的。

许多 RTS 库函数是周期密集型函数，因为它们会包含所有极端情况。当做出某些假设时（例如，没有 NaN 或无限值作为操作数或浮点运算结果），可以将这些函数替换为利用特定 C29 指令的更简单、更优化的函数。例如：asinf()、acosf()、atan2f()、ceilf()、cosf()、divf()、expf()、floorf()、fmodf()、roundf()、sinf()、truncf()。F29x-SDK 中提供了这些实现的示例，并已通过 -ffast-math 编译器选项启用。

使用 AUTOSAR 的汽车应用利用代码生成工具生成的数学库，其中包含浮点和定点库，以及用于定点到浮点和浮点到定点转换的函数。可以利用 C29 指令以高效的方式执行这些库。

2.4.2 使用库优化代码尺寸

应用可能包含预编译库，但可能不会使用这些库中的所有函数。要确保链接器排除各库和应用代码中未使用的函数，请确保两者均使用以下编译器选项构建：

```
-ffunction-sections
```

然后将每个函数放置在特定代码段中，如 .text.<function_name>，否则将每个函数放置在 .text 中。

备注

使用 __attribute__((section)) 将代码放入相应段中。-ffunction-sections 不会影响具有 section 属性的对象。在这种情况下，如果用户将多个函数分组到同一个代码段中，即使只使用其中一个函数，也会将映射到该部分的所有函数都链接到最终的可执行文件中。

2.4.3 C29 特别指令

C29 支持许多可用于优化特定函数类型的指令。下面列出了主要示例：

- SVGEN — 空间矢量生成可通过 QUADF 指令进行优化，使用内联函数“float __builtin_c29_quadf32(unsigned int * tdm_w_uip0, float * rw_fp1, float * rw_fp2)”进行利用。

  备注

  已在 F29x Motor Control SDK 中计划了 SVGEN（包括可应用于三级逆变器的 SVGEN）的优化实现。
- CRC — 循环冗余校验实现可通过 CRC 指令进行优化。F29x SDK 中提供了代码示例。还提供了一个内联函数“unsigned int __builtin_c29_i32_crc(unsigned int ui0, unsigned int ui1, unsigned int ui2, unsigned int ui3)”。
- 限制（饱和）运算 — 可使用 MINMAXF 指令进行优化。
  - 如果使用三元运算符编写 C 代码，且 min 与 max 是常量，并使用 -O3 和 -ffast-math 选项进行编译，则编译器会生成 MINMAXF 指令和一个出色的实现。此实现速度最快，因为它包含两条并行执行的 MV 指令，后跟 MINMAXF 指令。如果 min 和 max 是常量，但仅使用了 -O3，则不会生成 MINMAXF，而是生成 CMPF 和 SELECT 指令对。如果 min 和 max 并非常量，并且使用了 -O3 和 -ffast-math 选项，则编译器会生成从内存读取的 LD 指令，接着生成 CMPF、SELECT 和 MAXF 指令。

```
float saturation(float in)
{
   float out;
   out = (in > max)? max:((in < min)? min:in);
   return out;
}
```

- 使用 if.else 条件和下面的任一实现，行为与上述完全相同。

```
float saturation(float in)
{
   float out;
   if(in > max)
   {
      out = max;
   } else if(in < min)
   {
      out = min;
   } else {
      out = in;
   }
   return out;
}
```

```
float saturation(float in)
{
   float out = in;
   if(in > max)
   {
      out = max;
   } else if(in < min)
   {
      out = min;
   }
   return out;
}
```

```
float saturation(float in)
{
   float out = in;
   if(in > max)
   {
      out = max;
   }
   if(in < min)
   {
      out = min;
   }
   return out;
}
```

- 死区操作——当 min 和 max 为常量、启用 -O3 选项，且 C 代码采用如下代码块所示的三元运算符或 if..else 语句编写时，编译器会生成最高效的代码。

```
float deadzone(float in)
{
   float out;
   out = (in>1.0f)?(in-1.0f):((in>-1.0f)?0.0f:(in+1.0f));
   return out;
}
```

```
float deadzone(float in)
{
    float out;
    if(in >1.0f)
    {
        out = in-1.0f;
    } else if(in >-1.0f){
        out =0.0f;
    } else {
        out = in+1.0f;
    }
    return out;
}
```

2.4.4 C29 并行性

- C29 编译器可以利用 C29 架构的并行性，并行执行多条指令，尤其在依次执行多个独立运算的情况下。例如，下面的代码块演示了依次发生的两个相同 PID 运算。如果 DCL_runPID 在头文件中声明为静态函数，则编译器可以执行内联，然后并行执行两个 PID 运算。

备注

但是，为了通过并行操作提高性能，可能还需要将存储器对象放置在不同的 RAM 块中，以避免在同时访问与独立执行（例如 PID）实例相关的对象时出现存储器停顿。

```
float run_dualPID(DCL_PID *restrict p1, DCL_PID *restrict p2,float32_t rk1, float32_t yk1,
float32_t lk1,float32_t rk2, float32_t yk2, float32_t lk2)
{
   float x = DCL_runPID_C3(p1, rk1, yk1, lk1);
   float y = DCL_runPID_C3(p2, rk2, yk2, lk2);
   return x+y;
}
```

- 二进制 LUT 搜索 — 二进制查找表搜索在电机控制应用中很常见，并且可以通过将条件循环更改为固定迭代循环来进行优化。F29-SDK 在 examples/rtlibs/fastmath/binary_lut_search 中提供了一个示例。

2.4.5 首选 32 位变量和写入

ECC 位包含 32 位数据，因此小于 32 位的 RAM 写入大小，存储器包装会执行读取-修改-写入操作来补入新值，并重新计算整个 32 位字的 ECC。当发生小于 32 位的多次写入时，会导致停滞。包括 ARM CPU 在内的大多数 CPU 都是如此。

```
Example: 5 writes take 13 cycles
ST.16 *(ADDR1)(A4+#0x1a),#0x1
ST.16 *(ADDR1)(A4+#0x14),#0x303
ST.8 *(ADDR1)(A4+#0x1e),#0x0
ST.8 *(ADDR1)(A4+#0x16),#0x4
ST.16 *(ADDR1)(A4+#0x1c),#0x0
```

备注

应用程序代码必须尽量减少小于 32 位的写入，在一般情况下，尽可能使用 32 位变量。

使用 32 位变量有时还可避免编译器添加额外指令来对 16 位值进行符号扩展。以下示例显示了一个编译器用于将 16 位值符号扩展到 32 位值的附加指令。

```
Example:
int16_t mashup_16(int16_t in_a, int16_t in_b)
{
int16_t tmp1, tmp2, tmp3, tmp4;
   tmp1 = in_a + in_b;
   tmp2 = in_a - in_b;
   tmp3 = in_b - in_a;
   tmp3 = tmp1>>(tmp3 &0x7);
   tmp4 = tmp2<<(tmp1 &0x7);
   return (tmp3 ^ tmp4);
}
Generated code:
20103420 <mashup_16>:
20103420:  33dd 0004              MV    A4,D0
20103424:  33dd 0025              MV    A5,D1
20103428:  3204 18a4              SUB    A6,A5,A4,#0x0
2010342c:  b2e7 b200 3386 0007 20a4 0007
                                  MV.S16    A7,#0x7
                                ||    ADD    A8,A5,A4,#0x0
                                ||    AND.U16    A6,#0x7
20103438:  33d2 1d07              AND    A7,A8,A7
2010343c:  b3e4 3204 0108 1085
                                  SEXT.16    A8,A8
                                ||    SUB    A4,A4,A5,#0x0
20103444:  33d8 1087              LSL    A4,A4,A7
20103448:  b3d5 7a09 1506         ASR    A5,A8,A6
                                ||    RETD
```

```
2010344e:  33e6 10a4            XOR    A4,A5,A4
20103452:  33e4 0084            SEXT.16    A4,A4
20103456:  33e0 0004            MV     D0,A4
```

备注

由于所有 CPU 寄存器都是 32 位的，对寄存器的运算也是 32 位的，因此使用 32 位数据变量（对于时间关键型代码）一般会提高代码性能。

2.4.6 编码风格及其对性能的影响

开发人员编写 C 代码的方式会对性能产生影响。本节说明可能发生这种情况的具体示例场景。

- 使用循环时，性能可能因循环计数器是固定值还是变量而有所不同。如果是固定值，则编译器完全了解循环，并可确定最大限度发挥性能的策略：无论这是否意味着展开循环、对循环进行软件流水线处理等。例如，对于矩阵乘法，如果在循环中指定矩阵行和列大小，而不是将其作为函数参数传入，则性能将得到显著提升。
- 在某些情况下，将多个独立循环合并为一个循环可以提升性能。下面的第一个代码块会生成次优代码，第二个代码块的优化程度更高。

```
uint8_T Bit_Manipulation_Test_Case(void)
{
uint32_T result;
uint32_T i;
uint8_T valid;
result = 0u;
valid = TC_OK;
i = 0u;
/* Or Test Case */
for(i=0; i<BIT_MANIPULATION_ARRAY_SIZE; i++)
{
    result = (Swc1_Bit_Manipulation.Operand_A[i] | Swc1_Bit_Manipulation.Operand_B[i]);
    if(result != Swc1_Bit_Manipulation.Result_Or[i])
    {
        valid = TC_NOK;
    }
}
/* And Test Case */
for(i=0; i<BIT_MANIPULATION_ARRAY_SIZE; i++)
{
    result = (Swc1_Bit_Manipulation.Operand_A[i] & Swc1_Bit_Manipulation.Operand_B[i]);
    if(result != Swc1_Bit_Manipulation.Result_And[i])
    {
        valid = TC_NOK;
    }
}
/* Xor Test Case */
for(i=0; i<BIT_MANIPULATION_ARRAY_SIZE; i++) {
    result = (Swc1_Bit_Manipulation.Operand_A[i] ^ Swc1_Bit_Manipulation.Operand_B[i]);
    if(result != Swc1_Bit_Manipulation.Result_Xor[i]) {
        valid = TC_NOK;
    }
}
    return valid;
}
```

```
uint8_T Bit_Manipulation_Test_Case(void)
{
uint32_T result_or,result_and,result_xor;
uint32_T i;
uint8_T valid;
result_or = 0u;
result_and = 0u;
result_xor = 0u;
valid = TC_OK;
i = 0u;
/* Or, And, Xor Test Case */
for(i=0; i<BIT_MANIPULATION_ARRAY_SIZE; i++)
{
    result_or = (Swc1_Bit_Manipulation.Operand_A[i] | Swc1_Bit_Manipulation.Operand_B[i]);
    if(result_or != Swc1_Bit_Manipulation.Result_Or[i])
    {
        valid = TC_NOK;
```

```
    }
    result_and = (Swc1_Bit_Manipulation.Operand_A[i] & Swc1_Bit_Manipulation.Operand_B[i]);
    if(result_and != Swc1_Bit_Manipulation.Result_And[i])
    {
        valid = TC_NOK;
    }
    result_xor = (Swc1_Bit_Manipulation.Operand_A[i] ^ Swc1_Bit_Manipulation.Operand_B[i]);
    if(result_xor != Swc1_Bit_Manipulation.Result_Xor[i])
    {
        valid = TC_NOK;
    }
}
return valid;
}
```

- 此外，如果条件语句涉及从存储器加载或访问全局变量，则在可能的情况下将它们预加载到局部变量中可能会有所帮助。这样可以更多地使用 C29 CPU 上的更宽寄存器集。它还可防止从存储器加载值并立即对其执行条件检查时出现流水线停顿。下面的第一个代码块会生成次优代码，第二个代码块的优化程度更高。

```
// Variables are globals
if(xx ==FALSE)
{
    A = b * c + d;
    E = f * c + d;
    if(dd > high)
    {
        D = high;
    } elseif (dd < low) {
        if(kk == RUN)
        {
            D = low;
        } else {
            D = dd;
        }
    } else {
        D=dd;
    }
}
```

```
// Local copies of globals
float b_temp=b, c_temp=c, d_temp=d, f_temp=f, high_temp=high, low_temp=low, dd_temp=dd, kk_temp=kk,
D_temp=D, g_temp=g, h_temp=h;
if(xx==FALSE)
{
    A = b_temp * c_temp + d_temp;
    E = f_temp * c_temp + d_temp;
    if(dd_temp > high_temp)
    {
        D_temp = high_temp;
    } elseif (dd_temp < low_temp) {
        if(kk_temp == RUN)
        {
            D_temp = low_temp;
        } else {
            D_temp = dd_temp;
        }
    } else {
        D_temp=dd_temp;
    }
}
```

3 参考资料

1. 德州仪器 (TI)，*C29x-CPU 用户指南*
2. 德州仪器 (TI)，*F29H85x 和 F29P58x 实时微控制器数据表*
3. 德州仪器 (TI)，*F29H85x 和 F29P58x 实时微控制器技术参考手册*
4. 德州仪器 (TI)，*将应用软件迁移到 C29 CPU 用户指南*
5. 德州仪器 (TI)，*使用 C29x SSU 实现运行时安全和信息安全保护*
6. 德州仪器 (TI)，*TI C29x Clang 编辑器工具用户指南*
7. 德州仪器 (TI)，*从 TMS320F2837x、TMS320F2838x、TMS320F28P65x 迁移到 TMS320F29H85x*

4 修订历史记录

注：以前版本的页码可能与当前版本的页码不同

Changes from APRIL 30, 2025 to NOVEMBER 30, 2025 (from Revision A (April 2025) to Revision B (November 2025)) **Page**

重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、与某特定用途的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他安全、安保法规或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。对于因您对这些资源的使用而对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，您将全额赔偿，TI 对此概不负责。

TI 提供的产品受 TI 销售条款)、TI 通用质量指南 或 ti.com 上其他适用条款或 TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。 除非德州仪器 (TI) 明确将某产品指定为定制产品或客户特定产品，否则其产品均为按确定价格收入目录的标准通用器件。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

最后更新日期：2025 年 10 月