为Tiger SHARC DSP编译器调整C源程序

DSP工具编译器开发组， Analog Deviec Inc,版本3， 2001年11月26

在使用Visual DSP++™2.0版本的TigerSHARC® DSP系列的C/C++编译器时，本文档为如何获得最优的程序执行效率提供了一些指导。

使用优化器

C代码执行过程中，优化和非优化编译存在很大的差异。在某些情况下，执行优化代码要快10到20倍。在性能测试或在程序代码作为产品出售之前，都要进行程序优化。应注意的是，编译器的默认设置是不使用优化，这样，没有优化的程序代码就有助于程序员诊断原始代码存在的问题。

Tiger SHARC DSP编译器中的优化器可以将直接编写的C语言程序转化为执行效率很高的代码。调整程序的基本方法就是利用优化器对操作和数据的可视化来表示算法，这样安全的程序代码移植就有很大的灵活性。应当注意的是，未来的版本将增强优化器功能，更简单的算法将是进一步提升增强的益处的最佳途径。

使用统计性能分析器

调整源程序必须首先明确应用程序中的哪些部分是热点，VISUAL DSP++提供的统计性能分析器是找到这些热点的有效工具。如果你不熟悉应用程序，应该进行有编译诊断的编译，并运行非优化的程序，这样就能得到和C源程序直接相联系的结果。通过完全优化方式编译应用程序，且获得与汇编代码直接相关的统计特性，这样，就可以得到更加准确的结论。可能存在的唯一问题是如何将汇编行语句和原始代码关联起来。在链接过程中，不要去掉函数名，如果存在函数名，就可以滑动汇编窗口来定位这些热点。在非常复杂的程序代码中，可对循环进行计数，准确的定位源程序行。

注意：编译优化器可能已经移动了程序代码。

数据类型

编译器直接支持6种数据类型：

int	32位带符号整数
unsigned	32位无符号整数
long long	64位带符号整数
unsigned long long	64位无符号整数
float	32位浮点数
long double	64位浮点数

标准的C数据类型：char，short和long 型,无论是有符号和无符号格式都支持，但它们都是作为32位整型进行处理，double型数在缺省模式下是当作32位浮点数运算。

Long double 型算术操作通过库函数实现，因此要比float运算慢的多，只有当算法要求数据有足够大的范围和精度时才使用这种数据类型。

除了乘除法外，Long long 型的大多数数据操作基本上都由硬件直接支持。由于位数可以在每次操作中处理，所以Long long 型对于位操作算法通常是很有用的。

在循环中避免使用除法

硬件不能直接支持32位整数和浮点数的除法运算，所以整数和浮点数的除法和取模运算的代价都比较高。如果编译器已知除数的值，则编

译器将整型转换为2的幂次移位操作。一般规则是：在循环中不要使用除法。

使用16位和8位数据类型

硬件支持的其余数据类型，如16位整型短向量，8位整型短向量和16位定点复数，编译器都不直接支持，但通过内部函数，对指令的访问也是可行的。

应当注意的是，16位操作的执行效率要高于32位操作。然而，即使用16位数可明显得到相同的运行结果，编译器也不会在写成32位操作数的地方生成16位操作数。要生成16位的操作数，必须使用内部函数。

以下节选的程序段说明了以4*16短矢量书写的向量点积，为用内部函数书写程序代码提供了一种常用的推荐格式。

```
typedef long long int4x16;
#define add(x,y) __builtin_add_4x16(x,y)
#define mult(x,y) __builtin_mult_i4x16(x,y)
#define sum(x) __builtin_sum_4x16(x)

int sp4x16(int4x16 a[], int4x16 b[], int n)
{
   int i;
   int4x16 sum4 = 0;
   for (i = 0; i < n/4; ++i)
      sum4 = add(sum4, mult(a[i], b[i]));
   return sum(sum4);
}
```

附录B全面列出了编译器支持的所有内部函数。

索引阵列与指针

C语言有两种方法允许编程访问阵列数据：按固定的基指针索引或通过梯增指针。

下面两个向量相加程序说明了这两种方式：

索引阵列:

```
void va_ind(int a[], int b[], int out[],
int n) {
   int i;
   for (i = 0; i < n; ++i)
      out[i] = a[i] + b[i];
}
```

指针：

```
void va_ptr(int a[], int b[], int out[],
int n) {
   int i, *pout = p, *pa = a, *pb = b;
   for (i = 0; i < n; ++i)
      *pout++ = *pa++ + *pb++;
}
```

以上两种方式对于生成的程序代码通常没有任何差别，但是有时会有差别。通常，某个版本的算法可能比其他版本的算法好，但是它也不总是最好。生成的程序代码还受到周围程序代码的影响，这就是为什么可能存在差别的原因。指针方式将引入额外的变量，在优化器分析时，会与周围的程序代码竞争资源。另一方面，编译器必须将阵列访问转换为指针方式，有时这种方式不能实现某些人为的干预。

最好先使用数组方式的策略，如果不满意的话，才就尝试使用指针方式。在重要的循环之外要使用索引方式，这也比较容易理解。

使用_ipa选项

为了确保最优的性能，优化器需要知道它要分析的子程序外部确定的一些特性。特别有助于确知对齐方式、指针参数值和循环计数值。

编译器的—ipa可选项使能内部程序分析，从而得到以上信息。

在集成开发调试环境（IDDE）界面下，选择菜单命令Project，打开 Project options 对话框，点击Compile 按钮，选中interprocedural Optimization选项，即可使用—ipa选项。

在链接时，使用该选项可能重新调用编译器，

使用先前编译获得信息，重新编译原程序。

注意：只有在链接时才进行这种操作，当使用—S 选项时，就看不到－IPA选项的影响。为了看到汇编文件，在工程选项对话框中，将不着－save—temps写入到附加选项文本框中，编译程序后，就可以观察产生的 .S文件。

下面的所有建议都是假设使用了--ipa选项。

静态初始化常量

内部程序分析也可以识别那些只有一个值的变量，并用常数来代替，这会有利于优化。要做到这一点，在整个程序中此变量必须只有单一值。同所有全局变量默认初始化为零一样，若一变量被静态初始化为零，同时在程序的其他地方赋值了，分析会发现两个值，则不会认为该变量是一个常量。

Bad: (IPA 不认为val是常量)

```
#include <stdio.h>
static int val;
void init() {
   val = 3;
}
void func() {
   printf("val %d",val);
}
int main() {
   init();
   func();
}
```

Good: (IPA已知val为3)

```
#include <stdio.h>
static int val = 3;
void init() {
   printf("val %d",val);
}
void func() {
}
int main() {
   init();
   func();
}
```

循环向导

附录A概述了优化器如何将一个循环转化为运行效率高的代码，并介绍了“循环展开”技术。

不要自己展开循环

循环的展开不仅使程序难于阅读，而且不利于优化。为了自动使用两个计算块，编译器必须自己能展开循环。

在下面的例子中，第一个版本的循环程序运行速度是第二个的3.5倍。该情况下，内部程序分析可检测到a，b，c的初始值为四字排列，且n 是四的倍数。

Good: (编译器展开循环，使用两个计算块)

```
void va1(int a[], int b[], int c[], int n)
{
   int i;
   for (i = 0; i < n; ++i) {
      c[i] = b[i] + a[i];
   }
}
```

Bad: (编译器只使用一个计算块)

```
void va2(int a[], int b[], int c[], int n)
{
   int xa, xb, xc, ya, yb, yc;
   int i;
   for (i = 0; i < n; i+=2) {
      xb = b[i]; yb = b[i+1];
      xa = a[i]; ya = a[i+1];
      xc = xa + xb; yc = ya + yb;
      c[i] = xc; c[i+1] = yc;
   }
}
```

避免带有相关条件的循环

带有相关条件的循环是不知上一次循环所计算出的值，不能完成本次给定循环间隔的计算。当循环中有这样的相关条件时，编译器就不能载入新的循环间隔值。

在单个间隔中，某些相关条件是由标量产生的，因为是在定义它们之前就使用了。

Bad:(标量相关条件)

```
For (i=0;i<n;++i)
   X=a[i]-x
```

优化器可以在某些标量相关条件类，如重新载入值，出现之前重新排列循环的顺序。这是针对那些可使用综合交互操作，将数值的矢量简化为标量的循环，最基本的例子是乘和累加。

Good:(重新载入)

```
For (i=0;i<n;++i)
   X=x+a[i]*b[i]
```

在第一个例子中，标量相关条件是减法操作：如果不按顺序进行循环，变量x就会得到不同的值。作为对比，在例二中使用加法操作，无论编译器以何种顺序进行循环都将得到相同的结果。

不要手动方式返回的循环

为了在上次或下次循环中同时进行数据的载入和保存操作，就像同时在本次循环中计算一样，DSP程序中的循环一般都是通过手动方式轮回的。这种技术会引入循环相关条件，会防碍编译器有效的重新安排程序代码。所以最好给编译器一个标准版本，让编译器完成循环的返回。

Bad: (rotated)

```
float ss(float *a, float *b, int n) {
   float ta, tb , sum = 0.0f;
   int i = 0;
   ta = a[i]; tb = b[i];
   for (i = 1; i < n; i++) {
      sum += ta + tb;
      ta = a[i]; tb = b[i];
   }
   sum += ta + tb;
   return sum;
}
```

通过循环的返回，已经实现了了变量ta 和tb的相加，这两个变量又引入了循环相关条件，使编译器不能产生等长的间隔。优化器本身就能自动的使这种类型的循环轮回。

Good:

```
float ss(float *a, float *b, int n) {
   float sum = 0.0f;
      int i;
   for (i = 0; i < n; i++) {
      sum += a[i] + b[i];
   }
return sum;
}
```

避免在循环中写数组

在写数组元素时也会产生其他相关条件，在这样的循环中，从数组a中载入数据时，优化器不能辨别a的值是来自上一次循环中定义的值还是在下一次循环中覆盖其值。

Bad: (数组相关条件)

```
for (i = 0; i < n; ++i)
   a[i] = b[i] * a[c[i]];
```

当每次循环的地址是一个表达式，并且以一个固定的值变化，此时优化器可以解析该地址访问。这就是“还原变量”。

Good: (还原变量)

```
for (i = 0; i < n; ++i)
   a[i+4] = b[i] * a[i];
```

避免别名

下面的循环看起来不含有任何的相关条件:

```
void fn(int a[], int b[], int n) {
   for (i = 0; i < n; ++i)
      a[i] = b[i];
}
```

但是a和b都是参数，虽然它们都用[]符号声

明，但实际上二者都是指针，并指向同一个数组。这样，同一个数据就可以通过两个不同的指针访问，这就称这两个参数相互别名。

如果使用了--ipa选项，编译器就可以观察fn的调用位置，并可以确定是否存在指向同一数组的指针。

即使用了--ipa选项，也很容易产生明显的别名，内部程序分析是通过将指针和一组变量联系起来进行工作的，而这些指针可能指向程序中的同一点。为了简化分析,没有考虑控制程序流。如果发现两个指针指向的数组交叉，就认为这两个指针都指向了这两个数组的联合。

如在两个地方调用了上述函数fn，使用其全局数组作为其参数，内部程序分析将产生如下结果：

```
fn(glob1, glob2, N);   数组没有交叉，
fn(glob1, glob2, N);   a和b不同名(good)

fn(glob1, glob2, N);   数组没有交叉，
fn(glob3, glob4, N);   a和b不同名(good)

fn(glob1, glob2, N); 数组交叉，
fn(glob3, glob1, N); a和b可能同名(bad)
```

在第三种情况下，当检测是否存在别名时，内部程序分析一次将所有的调用联合考虑，而不是单独考虑。如果单独考虑，内部程序分析必须考虑程序流控制，大量的重新配置也将使得编译时间很长。

四字排列数据

为了充分高效的使用硬件资源，必须为计算单元提供数据。在许多算法中，运算数据访问的平衡就是通过载入128位的数据，保持硬件不间断的工作。

硬件结构要求对存储单元的访问自然对齐。因此，64位的访问必须是偶地址，而对128位的访问必须是四字对齐的地址。所以，为了产生运行效率最高的代码，通常需要确保数据是四字排列的。

编译器可以帮助建立数组数据的对齐，堆栈帧是保持四字排列的。不论是何种数据类型，顶层的数组都分配到了四字排列的地址。

如果编写的程序只将数组的第一个元素作为参数，同时编写的循环将输入数组处理为一个元素，并从元素0开始，则内部程序分析就能建立符合四字访问的对齐方式式。

在内循环中处理多维数组的某行时，也会产生类似情况。为了确保每行都是从四字边界开始，可以插入空数据实现这一点。例如，增加不必要的列而使行的长度是四的倍数。

当某循环存在一个独立的没有对齐的指针时，编译器会使用硬件数据对齐缓冲器来访问此指针所指向的128位数据。

如果指向对齐数据的指针作为参数传递给某个函数，就应该使用 Intrinsic_builtin_aligned命令。优化器首先安全的逼近指针，并保证在函数的开始，部分或所有的数据指针都是四字排列的地址。程序运行过程中，若存在没有排齐的地址，必然会产生一个很难解决的问题。

```
float ss(float *a, float *b, int n) {
   float sum;
   int i;
__builtin_aligned(a,4);
__builtin_aligned(b,4);
   < loop >
}
```

编译器可以查看文本值，-ipa可以将该文本传到该文本可以存放的地方。当存在不确定的值时，编译器会在循环中产生向量和非向量结构，并在实时运行中决定使用哪种结构

避免在循环中向下访问数组，此时向量化的软

件可能将它作为向上访问数组。

尽可能多的在内循环中处理数据

大部分程序运行过程中，内循环花费主要时间，所以优化器主要是优化内循环。若想使循环体的执行速度更快，优化程序要在循环体前或者循环体后降低程序的执行速度，这也认为是一种好的交换。所以应确保算法在内循环中消耗大部分时间，否则优化器实际上将使其执行速度更慢。

一种有用的技术就是循环切换。如果有循环嵌套，若内循环次数较少而外循环次数较多，则有必要重新编写循环程序，使外循运行次数较少。

"inline"限定词

在内循环中应避免函数调用，但如果必须调用，而且函数体较小，则考虑使用inline 限定词。此时会成行的编译函数体，这样不仅会减少函数调用和返回的时间，还会使得优化器可以更可视化的查看函数代码。这样做的代价就是将增加程序代码的长度。

注意延迟

当不能执行当前指令时，直到先前的指令退出了流水线，所有的流水机制都会引入等待周期。TigerSHARC DSP 在查找数据表时，将等待四个周期。A[B[I]]将比所期望的延迟还要多用四个周期。

在循环中避免使用条件指令

如果循环中含有条件转移指令，当条件判决的结论不同于编译器先前的估计时，就会有很长的等待开销。在许多情况下，编译器能够将if_else和？：结构转化为线性预测指令，但是如果在if或else模块中有复杂的计算，就会产生条件跳转指令。

将数组存贮在不同的空间——PM 限定词

pm类型限定词将数据存放在备用的存储单元中，以实现同时双数据访问。在同一个指令行内，Tiger SHARC DSP支持同时对两个存储区的操作。但是只有当两个地址是不同的内存空间时，才可以在一个周期内完成。如果同一指令行中的两个数据访问是对相同的存储器块操作，就必须等待。

下面是点积的例子：

```
for (i = 0; i < n; i++) {
   sum += a[i] + b[i];
}
```

因为每次循环都要从数组a和b中载入数据，所以确保这两个数组存放在不同的存储器块中很有好处。

要实现这一点，可以使用如下的静态数组声明：

pm int a[N]

或者限定的指针：

pm int *a。

缺省或标准模式是dm存储器空间。

用移位操作替代除法

除法运算需要函数调用，其代价相对较高。取模操作（%）也是除法的一种。当除数是2的幂时，编译器会使用执行速度更快的移位操作来代替除法。若被除数是无符号数整数，可直接用单一的移位指令取代，若为有符号数整数，则应增加额外的周期。此时可以考虑使用无符号数模型。

附录A：优化器的工作原理

我们通过浮点标量点积的计算来说明优化器的工作原理

```
float sp(float *a, float *b, int n) {
    int i;
    float sum=0;
    for (i=0; i<n; i++) {
    sum+=a[i]*b[i];
    }
    return sum;
}
```

经过代码的产生和常用的标量优化后，编译器生成如下形式的循环：

```
.P1L3:
    xR0 = [J0 += 1];;
    xR2 = [J1 += 1];;
    xfR4 = R0 * R2;;
    xfR5 = R5 + R4;;
    K0 = K0 - 1;;
    if nkle, jump .P1L3;;
```

循环退出测试已经移到了循环的低端，当循环计数器计数到0时，将重新载入新值。计算所得的和存放在寄存器xR5中。J0和J1寄存器保存初始化参数A和B的指针，并在每次循环中加一。

为了同时使用两个计算块，优化器展开循环到两个计算块中，两个循环同时并行运行。

```
.P1L3:
    yxR0 = l[J0 += 2];;
    yxR2 = l[J1 += 2];;
    xyfR4 = R0 * R2;;
    xyR5 = R5 + R4;;
    K0 = K0 - 2;;
    if nkle, jump .P1L3;;
```

此时计算所得的值存放在xR5和 yR5中，循环结束后，二者相加以得到最终的结果。 为了循环中使用长字载入，并得到同样的效率，编译器必须知道J0 和J1的偶数初始值。也应当注意除非编译器知道原始循环执行了偶数次，才能在循环外插入奇数次循环执行。

如果优化器可以确认J0 和 J1初始化为四字排列，它将展开循环，从而更好的利用TigerSHARC DSP的存储器带宽。

```
.P1L3:
    yxR1:0 = q[J0 += 4];;
    yxR3:2 = q[J1 += 4];;
    xyfR4 = R0 * R2;;
    xyfR6 = R1 * R3;;
    xyR5 = R5 + R4;;
    xyR7 = R7 + R6;;
```

```
    K0 = K0 - 4;;
    if nkle, jump .P1L3;;
```

此时，计算所得的值存放在 xR5, yR5, xR7 和yR7中。

最后，优化器软件将循环进入指令流水，展开并重复每次循环，从而使功能单元得到最高效的应用。如果编译器已经知道循环至少以20倍速执行，且循环次数是8的倍数，就会产生如下的代码：

```
.P1L3:
    yxR1:0 = q[J0+=4]; K0 = K0 ?4;;
    yxR3:2 = q[J1+=4];;
    yxR9:8 = q[J0+=4]; K0 = K0 ?4;;
    xyfR4 = R0 * R2; yxR11:10 = q[J1+=4];;
    xyfR6 = R1 * R3; yxR1:0 = q[J0+=4]; K0 = K0 ?4;;
    xyfR5 = R5 + R4; xyfR12 = R8 * R10; yxR3:2 = q[J1+=4];;

.P1L28:
    xyfR7 = R7 + R6; xyfR14 = R9 * R11; yxR9:8 = q[J0+=4]; K0 = K0 ?4;;
    xyfR5 = R5 + R12; xyfR4 = R0 * R2; yxR11:10 = q[J1+=4];;
    xyfR7 = R7 + R14; xyfR6 = R1 * R3; yxR1:0 = q[J0+=4]; K0 = K0 ?4;
    if nkle, jump .P1L28; xyfR5 = R5 + R4; xyfR12 = R8 * R10; yxR3:2 = q[J1+=4];;

    xyfR7 = R7 + R6; xyfR14 = R9 * R11;;
    xyfR5 = R5 + R12; xyfR4 = R0 * R2;;
    xyfR7 = R7 + R14; xyfR6 = R1 * R3;;
    xyfR5 = R5 + R4;;
    xyfR7 = R7 + R6;;
```

附录 B:内部函数

以下是编译器可以识别的内部函数：

int __builtin_add_sat(int, int);	Rs = Rm + Rm (S);
int __builtin_abs(int);	Rs = ABS Rm;
int __builtin_addbitrev(int, int);	Js = Jm + Jn (BR);
int __builtin_avg(int, int);	Rs = (Rm + Rn) / 2;
int __builtin_clip(int, int);	Rs = CLIP Rm by Rn;
int __builtin_count_ones(int);	Rs = ONES Rm;
int __builtin_fext(int, int);	Rs = FEXT Rm by Rn (SE);
int __builtin_frmul(int, int);	Rs = Rm * Rm;
int __builtin_frmul_sat(int, int);	Rs = Rm * Rm (S);
int __builtin_max(int, int);	Rs = MAX (Rm, Rn);
int __builtin_min(int, int);	Rs = MIN (Rm, Rn);
int __builtin_neg_sat(int);	Rs = - Rm;
int __builtin_sub_sat(int, int);	Rs = Rm - Rm (S);
float __builtin_conv_RtoF(int);	FRs = FLOAT Rm by -31;
float __builtin_copysignf(float, float);	FRs = Rm COPYSIGN Rn;
float __builtin_fabsf(float);	FRs = ABS Rm;
float__builtin_favgf(float,float);	FRs=(Rm+Rn)/2;
float__builtin_fclipf(float,float);	FRs=CLIPRmbyRn;
float__builtin_fmaxf(float,float);	FRs=MAX(Rm,Rn);
float__builtin_fminf(float,float);	FRs=MIN(Rm,Rn);

int__builtin_conv_FtoR(float);	Rs=FIXFRmby31;
longlong__builtin_llabs(longlong);	LRsd=ABSRmd;
longlong__builtin_llavg(longlong,longlong);	LRsd=(Rmd+Rnd)/2;
longlong__builtin_llclip(longlong,longlong);	LRsd=CLIPRmdbyRnd;
int__builtin_llcount_ones(longlong);	Rs=ONESRmd;
longlong__builtin_llmax(longlong,longlong)	LRsd=MAX(Rmd,Rnd);
longlong__builtin_llmin(longlong,longlong);	LRsd=MIN(Rmd,Rnd);
int__builtin_abs_2x16(int);	SRs=ABSRm;
int__builtin_add_2x16(int,int);	SRs=Rm+Rn;
int__builtin_add_2x16_sat(int,int);	SRs=Rm+Rn(S);
int__builtin_clip_2x16(int,int);	SRs=CLIPRmbyRn;
int__builtin_cmult_fr2x16(int,int);	MRa+=Rm**Rn(C);
int__builtin_cmult_i2x16(int,int);	MRa+=Rm**Rn(IC);
int__builtin_max_2x16(int,int);	SRs=MAX(Rm,Rn);
int__builtin_min_2x16(int,int);	SRs=MIN(Rm,Rn);
int__builtin_mult_fr2x16(int,int);	SRsd=Rmd*Rnd;
int__builtin_mult_i2x16(int,int);	SRsd=Rmd*Rnd(I);
int__builtin_neg_2x16(int);	SRs=-Rm;
int__builtin_sub_2x16(int,int);	SRs=Rm-Rn;
int__builtin_sub_2x16_sat(int,int);	SRs=Rm-Rn(S);
int__builtin_sum_2x16(int);	Rs=SUMRm;
int__builtin_compact_to_fr2x16(longlong);	SRs=COMPACTRmd;
longlong__builtin_expand_fr2x16(int);	Rsd=EXPANDSRm;
int__builtin_compact_to_i2x16(longlong);	SRs=COMPACTRmd(I);
longlong__builtin_expand_i2x16(int);	Rsd=EXPANDSRm(I);
longlong__builtin_merge_2x16(int,int);	SRsd=MERGERm,Rn;
longlong__builtin_abs_4x16(longlong);	SRsd=ABSRmd;
longlong__builtin_add_4x16(longlong,longlong);	SRsd=Rmd+Rnd;
longlong__builtin_add_4x16_sat(longlong,longlong);	SRsd=Rmd+Rnd(S);
longlong__builtin_clip_4x16(longlong,longlong);	SRsd=CLIPRmdbyRnd;
longlong__builtin_max_4x16(longlong,longlong);	SRsd=MAX(Rmd,Rnd);
longlong__builtin_min_4x16(longlong,longlong);	SRsd=MIN(Rmd,Rnd);
longlong__builtin_mult_fr4x16(longlong,longlong);	SRsd=Rmd*Rnd;
longlong__builtin_mult_fr4x16_sat(longlong,longlong);	SRsd=Rmd*Rnd(S);
longlong__builtin_mult_i4x16(longlong,longlong);	SRsd=Rmd*Rnd(I);
longlong__builtin_neg_4x16(longlong);	SRsd=-Rmd;
longlong__builtin_sub_4x16(longlong,longlong);	SRsd=Rmd-Rnd;
longlong__builtin_sub_4x16_sat(longlong,longlong);	SRsd=Rmd–Rnd(S);
int__builtin_sum_4x16(longlong);	Rs=SUMSRmd
int__builtin_abs_4x8(int);	SRs=ABSRm;
int__builtin_add_4x8(int,int);	SRs=Rm+Rn;
int__builtin_add_4x8_sat(int,int);	SRs=Rm+Rn(S);
int__builtin_clip_4x8(int,int);	SRs=CLIPRmbyRn;
int__builtin_max_4x8(int,int);	SRs=MAX(Rm,Rn);
int__builtin_min_4x8(int,int);	SRs=MIN(Rm,Rn);
int__builtin_sub_4x8(int,int);	SRs=Rm-Rn;
int__builtin_sub_4x8_sat(int,int);	SRs=Rm–Rn(S);
int__builtin_sum_4x8(int);	Rs=SUMRm;
longlong__builtin_merge_4x8(int,int);	SRsd=MERGERm,Rn;
longlong__builtin_abs_8x8(longlong);	SRsd=ABSRmd;
longlong__builtin_add_8x8(longlong,longlong);	SRsd=Rmd+Rnd;
longlong__builtin_add_8x8_sat(longlong,longlong);	SRsd=Rmd+Rnd(S);

longlong__builtin_clip_8x8(longlong,longlong);	SRsd=CLIPRmdbyRnd;
longlong__builtin_max_8x8(longlong,longlong);	SRsd=MAX(Rmd,Rnd);
longlong__builtin_min_8x8(longlong,longlong);	SRsd=MIN(Rmd,Rnd);
longlong__builtin_sub_8x8(longlong,longlong);	SRsd=Rmd-Rnd;
longlong__builtin_sub_8x8_sat(longlong,longlong);	SRsd=Rmd–Rnd(S);
int__builtin_sum_8x8(longlong);	Rs=SUMSRmd;
longlong__builtin_compose_64(inthi,intlo);	composea64-bitdatum
unsignedlonglong__builtin_compose_64u(unsigned,unsigned);	
longdouble__builtin_f_compose_64(float,float);	
int__builtin_high_32(longlong);	extractmostsignificantword
unsigned__builtin_high_32u(unsignedlonglong);	
float__builtin_f_high_32(longdouble);	
int__builtin_low_32(longlong);	extractleastsignificantword
unsigned__builtin_low_32u(unsignedlonglong);	
float__builtin_f_low_32(longdouble);	
_builtin_quad_builtin_compose_128(longlong,longlong);	composea128-bitdatum
int__builtin_high_64(__builtin_quad);	extractmostsignificantwords
int__builtin_low_64(__builtin_quad);	extractleastsignificantwords
int__builtin_sysreg_read(int);	readsystemregisters
longlong__builtin_sysreg_read2(int);	
__builtin_quad__builtin_sysreg_read4(int);	
void__builtin_sysreg_read(int);	writetosystemregisters
void__builtin_sysreg_read2(longlong);	
void__builtin_sysreg_read4(__builtin_quad);	
void*__builtin_alloca_aligned(int,int);	allocatedataonthestack
void__builtin_assert(int);ignored	