ANALOG DEVICES **Engineer To Engineer Note EE-*147***

**Technical Notes on using Analog Devices' DSP components and development tools**
Phone: (800) ANALOG-D, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com, FTP: ftp.analog.com, WEB: www.analog.com/dsp

# Tuning C Source Code for the TigerSHARC® DSP Compiler

*DSP Tools Compiler Group, Analog Devices Inc., Revision 3, 26 Nov 2001*

This document provides some guidelines for obtaining the best code execution performance from the TigerSHARC® DSP family's C/C++ compiler using VisualDSP++™ release 2.0.

## Use the Optimizer

There is a vast difference in the performance of C code that has been compiled optimized and non-optimized. In some cases optimized code can run ten or twenty times faster. Optimization should always be attempted before measuring performance or shipping code as product. Note that the default setting is for non-optimized compilation, the non-optimized default being there to assist programmers in diagnosing problems with their initial coding.

The optimizer in the TigerSHARC DSP compiler is designed to generate efficiently executing code from C that has been written in a straightforward manner. The basic strategy for tuning a program is to present the algorithm in a way that gives the optimizer excellent visibility of the operations and data, hence the greatest freedom to safely manipulate the code. Note that future releases will enhance the optimizer, and expressing algorithms simply will provide the best path for reaping the benefits of such enhancements.

## Use the Statistical Profiler

Tuning source begins with an understanding of what areas of the application are the hot spots. Statistical profiling provided in VisualDSP++ is an excellent means for finding those hot spots. If the application is unfamiliar to you, compile it with diagnostics and run it unoptimized. This will give you results that connect directly to the C source. You will obtain a more accurate view of your application if you build a fully optimized application and obtain statistics that relate directly to the assembly code. The only problem may be in relating assembly lines to the original source. Do not strip out function names when linking. If you have the function names then you can scroll the assembly window to locate the hot spots. In very complicated code you can locate the exact source lines by counting the loops.

Note: The compiler optimizer may have moved code around.

## Data Types

The compiler directly supports six scalar data types.

| | |
|---|---|
| `int` | 32-bit signed integer |
| `unsigned` | 32-bit unsigned integer |
| `long long` | 64-bit signed integer |
| `unsigned long long` | 64-bit unsigned integer |
| `float` | 32-bit floating point |
| `long double` | 64-bit floating point |

The standard C types, `char, short` and `long`, in their signed and unsigned forms are also supported for compatibility, but they are all implemented as 32-bit integers. `double` is supported and in the default mode implemented as a 32-bit floating-point value.

`long double` arithmetic operations are implemented by library routines and consequently are far slower than operations on `float`. This data type should only be used where the algorithm requires the `long double's` greater range and precision, and performance is not at a premium.

Most operations on `long long` are supported directly by the hardware, but multiplication and division are not. This type is often useful for bit manipulation algorithms because of the number of bits that can be processed in each operation.

### Avoid Division in Loops

The hardware does not provide direct support for 32-bit integer and floating point division, so the division and modulus operations on `int` and `float` are also expensive. The compiler will convert an integer division by a power of two to a shift operation if it actually knows the value of the divisor. General rule: Do not divide inside a loop.

### Using 16-bit and 8-bit Data Types

Other data types supported by the hardware such as short vectors of 16-bit integers, 8-bit integers and 16-bit fixed-point complex are not directly supported by the compiler, but access to the instructions are available through intrinsic functions.

It should be noted that 16-bit operations are often more efficient than 32-bit ones. However, the compiler will not generate a 16-bit operation where you wrote a 32-bit one, even if it is obvious to you that the 16-bit operation would generate the same result. To get 16-bit operations you need to use an intrinsic function.

This code snippet shows scalar product written with 4x16 short vector intrinsics. It illustrates the recommended style for code written with intrinsic functions.

```
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);
}
```

**Appendix B** gives a comprehensive list of intrinsic functions recognized by the compiler.

### Indexed Arrays vs. Pointers

C allows you to program data accesses from an array in two ways: either by indexing off an invariant base pointer or by incrementing a pointer.

These two versions of vector addition illustrate the two styles:

*Indexed Array:*

```
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];
}
```

*Pointers:*

```
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++;
}
```

The style should not make any difference to the generated code, but sometimes it does. Often one version of an algorithm will do better than the other but it is not always the same style that is better; the generated code is affected by the surrounding code, which is why there may be differences. The pointer style introduces additional variables that compete with the surrounding code for resources during the optimizer's analysis. Array accesses, on the other hand, must be transformed to pointers by the compiler and sometimes it does not do the job as well as you could do by hand.

The best strategy is to start with array notation. If this looks unsatisfactory try using pointers. Outside the important loops use the indexed style, as it is easier to understand.

### Use the –ipa Switch

To ensure the best performance the optimizer often needs to know things that can only be determined by looking outside the routine it is working on. In particular it helps to know the alignment and value of pointer parameters and the value of loop bounds.

The *–ipa* compiler switch enables inter-procedural analysis which makes this information available.

This may be switched on from the IDDE by checking the Interprocedural Optimization box in the Compile tab of the Project Options dialogue selected from the project menu.

When this switch is used the compiler may be called again from the link phase to recompile the program using additional information obtained during previous compilations.

Note: Because it only operates at link time, the effects of –ipa will not be seen if you compile with the –S switch. To see the assembler file put –save-temps in the Additional Options text box in the Compile tab of the Project Options dialogue and look at the .s file produced after your program has been built.

Much of the following advice assumes that the –ipa switch is being used.

### Initialize Constants Statically

Inter-procedural analysis will also identify variables that only have one value and replace them with constants that can enable better optimization. For this to happen a variable must have only a single value throughout the program. If the variable is statically initialized to zero, as all global variables are by default, and assigned to at one other point in the program the analysis sees two values and will not consider it to have a constant value.

*Bad: (IPA cannot see val is a constant)*

```
#include <stdio.h>

static int val;

void init() {
  val = 3;
}

void func() {
  printf("val %d",val);
}

int main() {
  init();
  func();
}
```

*Good: (IPA knows val is 3)*

```
#include <stdio.h>

static int val = 3;

void init() {
}

void func() {
  printf("val %d",val);
}

int main() {
  init();
  func();
}
```

### Loop Guidelines

**Appendix A** gives an overview of how the optimizer transforms a loop to generate highly efficient code. It describes the "loop unrolling" optimization technique.

#### *Do not unroll loops yourself*

Not only does loop unrolling make the program harder to read but it also prevents optimization. The compiler must be able to unroll the loop itself in order to use both compute blocks automatically.

In this example the first version of the loop runs two and a half times faster than the second, in cases where inter-procedural analysis can

determine that the initial values of `a`, `b`, and `c` are quad-word aligned and `n` is a multiple of four.

*Good: (Compiler unrolls and uses both compute blocks)*

```
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: (Compiler leaves on a single compute block)*

```
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;
  }
}
```

### *Avoid loop carried dependencies*

A loop-carried dependency is where computations in a given iteration of a loop cannot be completed without knowing the values calculated in earlier iterations. When a loop has such dependencies the compiler cannot overlap loop iterations.

Some dependencies are caused by scalar variables that are used before they are defined in a single iteration.

*Bad: (scalar dependency)*

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

An optimizer can reorder iterations in the presence of the class of scalar dependencies known as reductions. These are loops that reduce a vector of values to a scalar using an associative and commutative operator. The most common example is multiply and accumulate.

*Good: (a reduction*)

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

In the first case, the scalar dependency is the subtraction operation: the variable `x` changes in a manner that will give different results if the iterations are performed out of order. In contrast, in the second case, the properties of the addition operator mean that the compiler can perform the operations in any order, and still get the same result.

### *Do not rotate loops by hand*

Loops in DSP code are often "rotated" by hand, attempting to do loads and stores from earlier and future iterations at the same time as computation from the current iteration. This technique introduces loop-carried dependencies that prevent the compiler from rearranging the code effectively. It is better to give the compiler a "normalized" version, and leave the rotating to the compiler.

*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;
}
```

By rotating the loop, the scalar variables `ta` and `tb` have been added, and they have introduced loop-carried dependencies which prevent the compiler from issuing iterations in parallel. The optimizer is capable of doing this kind of loop rotation itself.

*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;
}
```

### *Avoid array writes in loops*

Other dependencies can be caused by writes to array elements. In this loop the optimizer will not be able to tell whether the load from `a` reads a value defined on a previous iteration or one that will be overwritten in a subsequent iteration.

*Bad: (array dependency)*

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

The optimizer can resolve access patterns where the addresses are expressions that vary by a fixed amount on each iteration. These are known as "induction variables".

*Good: (induction variables)*

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

### *Avoid aliases*

It may seem that a loop that looks like this does not contain any loop carried dependencies,

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

but `a` and `b` are both parameters, and although they are declared with `[]` they are in fact pointers which may point to the same array. When the same data may be reachable through two pointers we say they may alias each other.

If the *-ipa* switch is enabled the compiler will be able to look at the call sites of `fn` and possibly determine whether they ever do point at the same array.

Even with the *-ipa* switch it is quite easy to create apparent aliases. The inter-procedural analysis works by associating pointers with sets of variables they may refer to at some point in the program. To simplify the analysis no account is taken of the control flow, and if the sets for two pointers are found to intersect then both pointers are assumed to point to the union of the two sets.

If `fn` above were to be called in two places with the global arrays as arguments then the inter-procedural analysis will have the following results:

```
fn(glob1, glob2, N);   Sets do not intersect: a and b
fn(glob1, glob2, N);   are not aliases (good)
```

```
fn(glob1, glob2, N);   Sets do not intersect: a and b
fn(glob3, glob4, N);   are not aliases (good)
```

```
fn(glob1, glob2, N);   Sets intersect: a and b
fn(glob3, glob1, N);   may be aliases (bad)
```

The third case shows that IPA considers the union of all calls, at once, rather than considering each call individually, when determining whether there is a risk of aliasing. If each call were considered individually, IPA would have to take flow control into account, and the number of permutations would make compilation time impracticably long.

### *Quad-word align your data*

To make most efficient use of the hardware the compute units must be kept fed with data. In many algorithms the balance of data accesses to computations is such that to keep the hardware busy data must be fetched with 128-bit loads.

The hardware architecture requires that references to memory be naturally aligned. Thus 64-bit references must be at even address locations, and 128-bit at quad-aligned addresses. So for the

most efficient code to be generated you often have to ensure you data is quad-word aligned.

The compiler helps establish the alignment of array data. The stack frames are kept quad-word aligned. Top-level arrays are allocated at quad-word aligned addresses, regardless of their data types.

If you write programs that only pass the address of the first element of an array as a parameter, and write loops that process input arrays an element at a time, starting at element zero, then inter-procedural analysis should be able to establish that the alignment is suitable for quad-word accesses.

Another situation occurs where the inner loop processes a single row of a multi-dimensional array. Consider making sure each row begins on a quad-word boundary, possibly inserting dummy data to do so, i.e. adding additional columns so that the row length is a multiple of 4.

When a loop has a single non-aligned pointer, the compiler can make use of the hardware data alignment buffer to access 128-bits of data from that pointer.

If pointers to aligned data are passed as parameters to a function then make use of the intrinsic __builtin_aligned. Optimizers take a safety first approach with pointers and it is good practice to assure the optimizer locally at the start of a function that some or all of the data pointers are guaranteed to be quad aligned addresses. The corollary of course is that if you make the assurance and the program passes a non-aligned address then you will have an obscure bug!

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

  < loop >
}
```

A literal value can be seen by the compiler and *–ipa* will propagate literals where it can. Where ambiguity persists the compiler will plant a vector and a non-vector form of the loop, deciding between them at run time.

Avoid descending access to arrays in loops where possible as the vectorizing software currently looks for ascending cases only.

### *Do as much work as possible in the inner loop*

The optimizer focuses on the inner loops because this is where most programs spend the majority of their time. It is considered a good trade-off for an optimization to slow down the code before and after a loop if it is going to make the loop body run faster, so make sure that your algorithm also spends most of its time in the inner loop, otherwise it may actually be made to run slower by optimization.

A useful technique is loop switching. If you have nested loops where the outer loop runs many times and the inner loop runs a small number of times, it may be possible to rewrite the loops so that the fewer number of iterations is for the outer loop.

### *The "inline" qualifier*

Try and avoid function calls in inner loops, but if you must and the function is small consider using the inline qualifier. This will cause the body of the function to be compiled inline and will not only save the cost of the function call and return code, but also give the optimizer greater visibility into the function code. The cost is that code size will expand.

### Be Aware of Latencies

All pipelined machines will introduce stall cycles when you cannot execute the current instruction until a prior instruction has exited the pipeline.

TigerSHARC DSP stalls for four cycles on a table lookup. A[B[I]] takes four cycles more than you may expect.

### *Avoid conditional code in loops*

If a loop contains conditional code, there may be a large stall penalty if the decision has to branch against the compiler's prediction. In many cases, the compiler will be able to convert if-else and ?: constructs into linear predicated instructions, but complex calculations in the if or else block will cause conditional jump code to be generated.

### *Try to put arrays into different spaces – the PM qualifier*

The `pm` type qualifier places data in alternate memory for dual simultaneous access. TigerSHARC DSP can support two memory operations on a single instruction line. However, this will only complete in one cycle if the two addresses are situated in different memory spaces – if both access the same block there will be a stall.

Take as an example the dot product:

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

Since on every cycle we load from arrays `a` and `b`, it may be useful to ensure that these arrays are located in different blocks.

You can ensure this by using either static array declarations such as:

```
pm int a[N];
```

or qualified pointers such as:

```
pm int * a;
```

The default or normal mode is `dm`.

### Replacing Division with Shifts

Division requires a function call and is relatively expensive. The modulus (%) operator is also a form of division. Try and use division by powers of two that can be replaced by the compiler with much faster shift operations. The division of an unsigned integer by a power of two can be replaced by a single shift operation. The division of a signed integer by a power of two requires additional cycles. Consider if a cast to unsigned could be applied.

## Appendix A: How the Optimizer Works

We will use the following floating-point scalar product to show how the optimizer works:

```
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;
}
```

After code generation and conventional scalar optimizations, the compiler will have generated a loop that looks something like this:

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

The loop exit test has been moved to the bottom and the loop counter rewritten to count down to zero. Sum is being accumulated in xR5. J0 and J1 hold pointers that are initialized with the parameters A and B and incremented on each iteration.

In order to use both compute blocks, the optimizer unrolls the loop to run two iterations in parallel.

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

Sum is now being accumulated in xR5 and yR5, which must be added together after the loop to produce the final result. In order to the use long word loads needed for the loop to be as efficient as this, the compiler has to know that J0 and J1 have initial values that are even. Note also that unless the compiler knows that original loop was executed an even number of times a conditionally executed odd iteration must be inserted outside the loop.

If the optimizer can verify that J0 and J1 are initially quad word aligned then it will unroll the loop to make better use of the TigerSHARC DSP's memory bandwidth.

```
.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;;
```

Now Sum is being calculated in xR5, yR5, xR7 and yR7.

Finally the optimizer software pipelines the loop, unrolling and overlapping iterations to obtain the highest possible use of functional units. The following code would be generated if it were known that the loop executed at least twenty times and the loop count was a multiple of eight.

```
.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;;
```

## Appendix B: Intrinsic Functions

## These are the intrinsic functions recognized by the compiler:

```
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 = CLIP Rm by Rn;
float __builtin_fmaxf(float, float);                      FRs = MAX (Rm, Rn);
float __builtin_fminf(float, float);                      FRs = MIN (Rm, Rn);
int __builtin_conv_FtoR(float);                           Rs = FIX FRm by 31;

long long __builtin_llabs(long long);                     LRsd = ABS Rmd;
long long __builtin_llavg(long long, long long);          LRsd = (Rmd + Rnd) / 2;
long long __builtin_llclip(long long, long long);         LRsd = CLIP Rmd by Rnd;
int       __builtin_llcount_ones(long long);              Rs = ONES Rmd;
long long __builtin_llmax(long long, long long);          LRsd = MAX (Rmd, Rnd);
long long __builtin_llmin(long long, long long);          LRsd = MIN (Rmd, Rnd);
```

```
int __builtin_abs_2x16(int);                              SRs = ABS Rm;
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 = CLIP Rm by Rn;
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 = SUM Rm;

int __builtin_compact_to_fr2x16(long long);               SRs = COMPACT Rmd;
long long __builtin_expand_fr2x16(int);                   Rsd = EXPAND SRm;

int __builtin_compact_to_i2x16(long long);                SRs = COMPACT Rmd (I);
long long __builtin_expand_i2x16(int);                    Rsd = EXPAND SRm (I);
long long __builtin_merge_2x16(int, int);                 SRsd = MERGE Rm, Rn;

long long __builtin_abs_4x16(long long);                  SRsd = ABS Rmd;
long long __builtin_add_4x16(long long, long long);       SRsd = Rmd + Rnd;
long long __builtin_add_4x16_sat(long long, long long);   SRsd = Rmd + Rnd (S);
long long __builtin_clip_4x16(long long, long long);      SRsd = CLIP Rmd by Rnd;
long long __builtin_max_4x16(long long, long long);       SRsd = MAX (Rmd, Rnd);
long long __builtin_min_4x16(long long, long long);       SRsd = MIN (Rmd, Rnd);
long long __builtin_mult_fr4x16(long long, long long);    SRsd = Rmd * Rnd;
long long __builtin_mult_fr4x16_sat(long long, long long); SRsd = Rmd * Rnd (S);
long long __builtin_mult_i4x16(long long, long long);     SRsd = Rmd * Rnd (I);
long long __builtin_neg_4x16(long long);                  SRsd = - Rmd;
long long __builtin_sub_4x16(long long, long long);       SRsd = Rmd - Rnd;
long long __builtin_sub_4x16_sat(long long, long long);   SRsd = Rmd – Rnd (S);
int __builtin_sum_4x16(long long);                        Rs = SUM SRmd;

int __builtin_abs_4x8(int);                               SRs = ABS Rm;
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 = CLIP Rm by Rn;
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 = SUM Rm;

long long __builtin_merge_4x8(int, int);                    SRsd = MERGE Rm, Rn;
long long __builtin_abs_8x8(long long);                     SRsd = ABS Rmd;
long long __builtin_add_8x8(long long, long long);          SRsd = Rmd + Rnd;
long long __builtin_add_8x8_sat(long long, long long);      SRsd = Rmd + Rnd (S);
long long __builtin_clip_8x8(long long, long long);         SRsd = CLIP Rmd by Rnd;
long long __builtin_max_8x8(long long, long long);          SRsd = MAX (Rmd, Rnd);
long long __builtin_min_8x8(long long, long long);          SRsd = MIN (Rmd, Rnd);
long long __builtin_sub_8x8(long long, long long);          SRsd = Rmd - Rnd;
long long __builtin_sub_8x8_sat(long long, long long);      SRsd = Rmd – Rnd (S);
int __builtin_sum_8x8(long long);                           Rs = SUM SRmd;

long long __builtin_compose_64(int hi, int lo);             compose a 64-bit datum
unsigned long long __builtin_compose_64u(unsigned, unsigned);
long double __builtin_f_compose_64(float, float);

int __builtin_high_32(long long);                           extract most significant word
unsigned __builtin_high_32u(unsigned long long);
float __builtin_f_high_32(long double);

int __builtin_low_32(long long);                            extract least significant word
unsigned __builtin_low_32u(unsigned long long);
float __builtin_f_low_32(long double);

__builtin_quad __builtin_compose_128(long long, long long); compose a 128-bit datum
int __builtin_high_64(__builtin_quad);                      extract most significant words
int __builtin_low_64(__builtin_quad);                       extract least significant words

int __builtin_sysreg_read(int);                             read system registers
long long __builtin_sysreg_read2(int);
__builtin_quad __builtin_sysreg_read4(int);

void __builtin_sysreg_read(int);                            write to system registers
void __builtin_sysreg_read2(long long);
void __builtin_sysreg_read4(__builtin_quad);

void * __builtin_alloca_aligned(int, int);                  allocate data on the stack
void __builtin_assert(int);                                 ignored
```