Table 3-2. Compiler Options for Performance

Option	Description
-O3	Represents the highest level of optimization available. Various loop optimizations are performed, and various file–level characteristics are also used to improve performance.
-pm	Combines source files to perform program–level optimization by allowing the compiler visibility to the entire application source.
-oi<*size*>	Enables inlining of functions based on a maximum size. (Enabled with -O3.) Size here is determined internally by the optimizer and does not correspond to bytes or any other known standard unit. Use a -on*x* option to check sizes of individual functions.
-mb	Asserts to the compiler that all data is on–chip. This option is used to enable the compiler to generate dual–MAC. See section 1.4.2.2 for more details.
-op2	When used with -pm, this option allows the compiler to assume that the program being compiled does not contain any functions or variables called or modified from outside the current file. The compiler is free to remove any functions or variables that are unused in the current file.
-mn	Re–enables optimizations disabled when using -g option (symbolic debugging). Use this option when it is necessary to debug optimized code.
-v*device* *[:revision]*	Tells the compiler what physical hardware device and core revision you are compiling for. Correct use of this option minimizes the additional code generated by the compiler to work around any hardware ECN's.

3.2.4 Compiling for codesize

When compiling to minimize codesize we recommend using the options –O3, -pm, -op2, -mb, -oi0, and –ms. The compiler makes certain tradeoffs between performance and codesize. Compiling with –ms tells the compiler to favor lower codesize instead of higher performance when making those tradeoffs. Some of the changes you may see when compiling with –ms include:

- The beginning of each C function will not be aligned. When compiling without –ms the start of each function is aligned on a 4 byte boundary. This enables profiling to be more consistent. When you make changes to one function the cycle counts of other functions won't change simply from alignment differences. Compiling with –ms eliminates the codesize overhead (NOP's) of aligning functions.
- Loops are not unrolled. Loop unrolling increases codesize and thus is not performed under –ms.
- Instruction selection changes. When compiling with –ms the compiler may use different instructions for some expressions. For example, certain AMOV (or

mar) instructions require 1 more byte than the comparable execute phase MOV instruction.

- Less code hoisting out of loops. Hoisting loop invariant code out of loops can lead to increased code size. Compiling with –ms reduces the amount of hoisting performed by the compiler.
- Fewer predicated instructions. The compiler attempts to predicate some small blocks of code to eliminate control flow. Compiling with –ms reduces the size threshold for determining which instructions should be predicated to reduce codesize.

3.4.2.3 Memory Layout Considerations

To issue a dual-MAC instruction in a single cycle, the arrays pointed to by the various dual-MAC operands must be properly laid out in memory. Consider the following C code that can be used to generate a dual-MAC.

```
y0 += (long)x[j] * h[i];
y1 += (long)y[j] * h[i];
```

Recall that a C55x dual-MAC has three memory operands, one of them being shared between the two MAC's. In this example h[i] is the shared operand. For single cycle dual-MAC, the memory pointed to by the unshared operands (arrays x and y in our example) must be in DARAM. The shared operand (array h, typically the coefficient array) must reside in a separate memory bank (or block) from the other two operands (DARAM or SARAM is acceptable). See section 3.5.4 for more information on controlling the memory placement of C arrays.

3.4.5 Compiler support for circular addressing

The compiler can generate assembly code that utilizes the circular addressing hardware of the C55x. This feature is available with small memory model compilation only using version 2.30 and higher of the code generation tools. (Small memory model is the default mode of the compiler.) The optimizer must be run (i.e. you must use the –O<x> option) to enable generation of circular addressing code.

The compiler can transform certain array references inside loops into circular addressing code. Given an array ‘a’ of size ‘S’ and index variable ‘x’, the compiler will recognize ‘a’ as a circular buffer if the following conditions hold:

- All index expressions for ‘a’ contain only ‘x’ and constants. An index expressions must be of the form “bx + c” where ‘b’ and ‘c’ are constants.
- Increments of ‘x’ are always by positive constants.
- Increments of ‘x’ are always followed by “% S” (i.e. modulus the size of the buffer). The syntax “& T” is also acceptable where T = S – 1 and S is a power of two.
- ‘x’ is initialized with a value that is a compile-time constant.

Example 0-1 shows a simple C function for which the compiler can make use of the C55x circular addressing hardware. The compiler detects a circular buffer with array ‘b’, index variable ‘x’ and size 16. The assembly generated by the compiler is shown in Example 0-2.

Example 0-1. A simple circular addressing example

```
void circ(int *a, int *b)
{
    int i, x = 0;

    for(i = 0; i < 16; i++) /* (1) start of circular buffer lifetime */
    {
        a[i] = b[x];
        x = (x + 3) % 16; /* or x = (x + 3) & 15 */
    }                       /* (2) end of circular buffer lifetime */
}
```

Example 0-2. A simple circular addressing example – algebraic assembly

```
_circ:

        BSA01 = @AR1_L
||      mmap()

        AR0 = #0  ; |6|
||      mar(AR3 = AR0) ; |2|

        bit(ST2, #ST2_AR0LC) = 1  ; circ mode (AR0) ; |6|
```

```
||      BK03 = #16

        BRC0 = #15
        localrepeat {
                                              ; loop starts
L1:
            *AR3+ = *AR0 ; |8|
            mar(AR0 + #3) ; circular mode ; |9|
        }                                     ; loop ends ; |10|
L2:
        bit(ST2, #ST2_AR0LC) = 0  ; circ mode (AR0)
        return
                                              ; return occurs
```

Example 0-2. A simple circular addressing example – mnemonic assembly

```
_circ:
        MOV mmap(AR1), BSA01

        MOV #0, AR0 ; |6|
||      AMOV AR0, AR3 ; |2|

        BSET ST2_AR0LC ; circ mode (AR0) ; |6|
||      MOV #16, BK03

        MOV #15, BRC0
        RPTBLOCAL L2-1
                                              ; loop starts
L1:
            MOV *AR0, *AR3+ ; |8|
            AADD #3, AR0 ; circular mode ; |9|
                                              ; loop ends ; |10|
L2:
        BCLR ST2_AR0LC ; circ mode (AR0)
        RET
                                              ; return occurs
```

Note the following restrictions on circular addressing:

- A negative update of 'x' (e.g. "x = (x – 1) % 10") is not allowed and would prohibit the compiler from generating circular addressing code. However, a negative update can be simulated via a large positive update. Subtracting one from a circular buffer of size ten would look like this "x = (x + 9) % 10".
- The circular buffer size must be a compile time constant. That is, the compiler must be able to determine what the size of the buffer is, or it will not generate circular addressing code.
- The compiler does not support circular addressing via the CDP register.
- The hardware has a limited number of registers that it can use for circular addressing. If the user writes code that requires more than the available number of circular registers, the compiler will attempt to generate efficient code to simulate circular addressing for some circular buffers.

- The compiler will not generate circular addressing code if a function call would be present within the lifetime of the circular buffer (e.g. No function calls are allowed between (1) and (2) in example 0-1.)
- Circular addressing is not available when compiling with the large memory model (-ml flag to cl55).

Some of these restrictions may be eased in future revisions of the compiler.

Previous versions of this document describe how to use user-defined macros CIRC_UPDATE and CIRC_REF to implement circular addressing. Example 0-3 shows the proper way to now define those macros and how they would be used to transform Example 0-2.

Example 0-3. Using CIRC_REF and CIRC_UPDATE

```
#define CIRC_UPDATE(var,inc,size) (var) = ((var) + (inc)) % (size);
#define CIRC_REF(var,size) (var)

void circ(int *a, int *b)
{
   int i, x = 0;


   for(i = 0; i < 16 ; i++)
   {
      a[i] = b[CIRC_REF(x,16)];
      CIRC_UPDATE(x,3,16)
   }
}
```

Table 3-7. C Coding Methods for Generating Efficient C55x Assembly Code

Operation	Recommended C Code Idiom

16bit * 16bit => 32bit (multiply)	`int a,b;`
	`long c;`
	`c = (long)a * b;`

Q15 * Q15 => Q31 (multiply)
Fractional mode (no saturation)
	`int a,b;`
	`long c;`
	`c = ((long)a * b) << 1;`

Q15 * Q15 => Q15 (multiply) Fractional mode with saturation	`int a,b,c;` `c = _smpy(a,b);`

Q15 * Q15 => Q31 (multiply)
Fractional mode with saturation
	`int a,b;`
	`long c;`
	`c = _lsmpy(a,b);`

32bit + 16bit * 16bit => 32 bit (MAC)	`int a,b;`
	`long c;`
	`c = c + ((long)a * b));`

Q31 + Q15 * Q15 => Q31 (MAC)
Fractional mode (no saturation)
	`int a,b;`
	`long c;`
	`c = c + ((long)a * b) << 1;`

Q31 + Q15 * Q15 => Q31 (MAC) Fractional mode with saturation	`int a,b;` `long c;`

	`c = _smac(c,a,b);`

32bit – 16bit * 16bit => 32 bit (MAS)	`int a,b;`
	`long c;`
	`c = c – ((long)a * b));`

Q31 – Q15 * Q15 => Q31 (MAS)
Fractional mode (no saturation)
	`int a,b;`
	`long c;`
	`c = c – ((long)a * b) << 1;`

Q31 – Q15 * Q15 => Q31 (MAS)
Fractional mode with saturation
	`int a,b;`
	`long c;`
	`c = _smas(c,a,b);`

16bit +/- 16bit => 16bit
32bit +/- 32bit => 32bit
40bit +/- 40bit => 40bit (addition or subtraction)
	`<int, long, long long> a,b,c;`
	`c = a + b;`
	`/* or */`
	`c = a – b;`

16bit + 16bit => 16bit (addition) with saturation	`int a,b,c;` `c = _sadd(a,b);`
32bit + 32bit => 32bit (addition) with saturation	`long a,b,c;` `c = _lsadd(a,b);`
40bit + 40bit => 40bit (addition) with saturation	`long long a,b,c;` `c = _llsadd(a,b);`
16bit – 16bit => 16bit (subtraction) with saturation	`int a,b,c;` `c = _ssub(a,b);`
32bit – 32bit => 32bit (subtraction) with saturation	`long a,b,c;` `c = _lssub(a,b);`
40bit – 40bit => 40bit (subtraction) with saturation	`long long a,b,c;` `c = _llssub(a,b);`

Max and min	`<int, long, long long> a,b;`
	`a = a > b ? a : b; /* max */`
	`a = a < b ? a : b; /* min */`

Bidirectional left shift (i.e. shift left if shift value is positive, shift right if shift value is negative).	`<int, long, long long> a;`
	`int b;`
	`a = b > 0 ? a << b : a >> -b;`

|16bit| => 16bit
|32bit| => 32bit
|40bit| => 40bit (absolute value)
	`<int, long, long long> a,b;`
	`b = abs(a);  /* or */`
	`b = labs(a); /* or */`
	`b = llabs(a);`

|16bit| => 16bit
|32bit| => 32bit
|40bit| => 40bit (absolute value)
with saturation
	`<int, long, long long> a,b;`
	`b = _abss(a);  /* or */`
	`b = _labss(a); /* or */`
	`b = _llabss(a);`

round(Q31) = > Q15 (rounding towards infinity)
with saturation
	`long a;`
	`int b;`
	`b = _rnd(a)>>16;`

Q39 => Q31 (format change)	`long long a;`
	`long b;`
	`b = a >> 8;`

Q30 = > Q31 (format change)
with saturation
	`long a;`
	`long b;`
	`b = _lsshl(a,1);`

40bit => 32bit both Q31 (size change)	`long long a;`
	`long b;`
	`b = a;`