Code Optimization and Advanced C

Chapter 6

Introduction

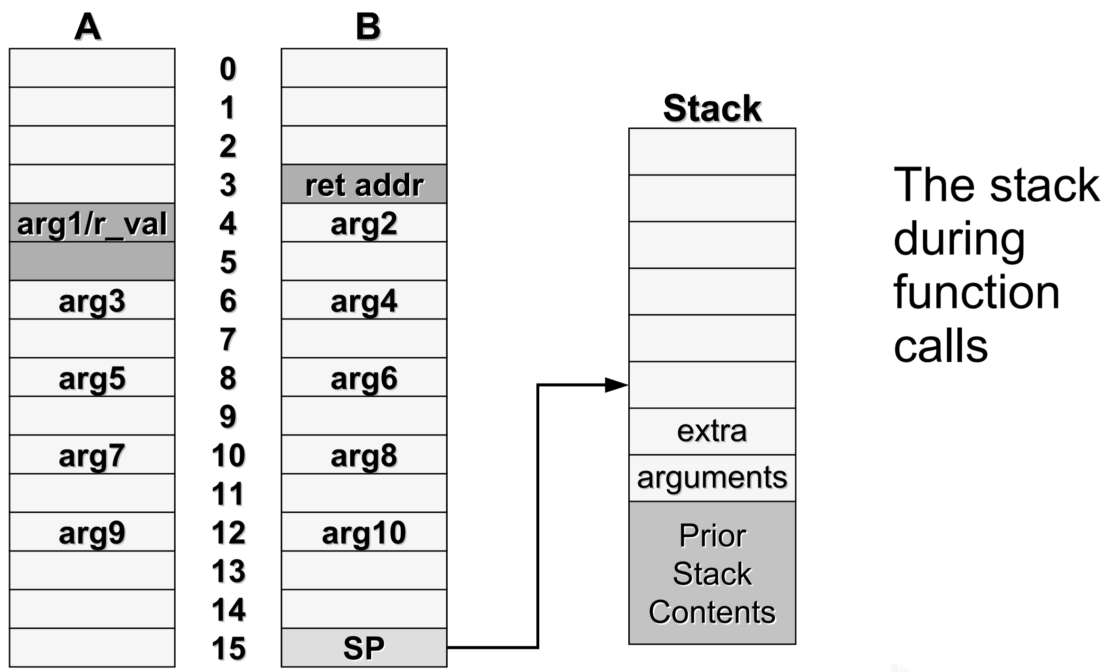
Some advanced issues in C programming on the C6x platform. Two broad categories included here are the C runtime environment and optimizing C code.

C Runtime Environment

Stack Pointer

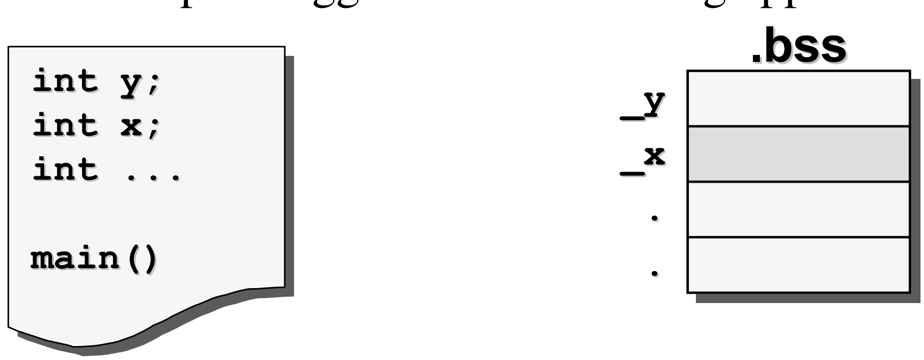
- The stack pointer (SP) points to the first empty location
- SP is double-word aligned before each function
- It is created by the compiler's init routine (`boot.c`)
- The length is defined by `-stack` in the linker option
- The stack length is not validated at runtime

- During function calls, C or assembly, the stack is used as follows:



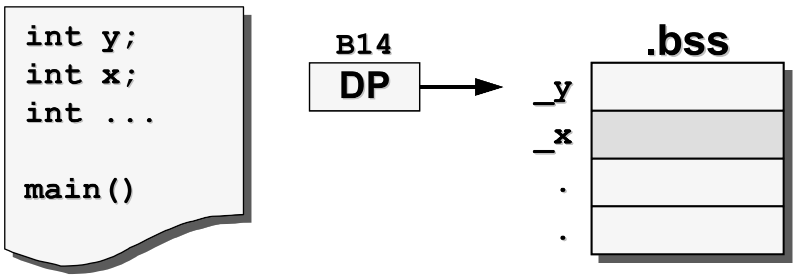
Global Pointer

- To access global variables in assembly (linear assembly), we have in the past suggested the following approach:




```
mvkl     _x,  A0   ;loads ptr w/ lower 16-bits of address
mvkh     _x,  A0   ;loads ptr w/ upper 16-bits of address
ldw.D2   *A0, A7   ;loads register A7 from mem loc "_x"
```

- This takes three cycles, but here is a faster way using the global data pointer (DP)
- First set up a pointer to the global (.bss) section




```
DP   .equ     B14
```

- With this pointer we can access global variables in one cycle

```
DP   .equ     B14
          ___________
          ___________
          ___________

          ldw.D2    *+DP(_x), A7
```

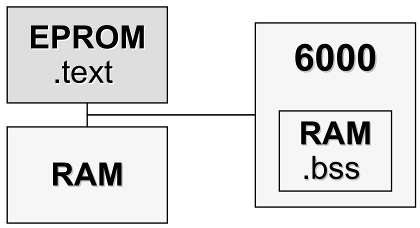
  – The DP points to the top of all global/static variables
  – Global access's, use the DP with the var in ( )
  – This special addressing features applies to both B14 (DP) and B15 (SP)
  – The DP is created by the compiler's init routine (`boot.c`)

Memory Management

- In Chapter 3 memory management was briefly discussed
- Here we noted code and global data was organized as follows:

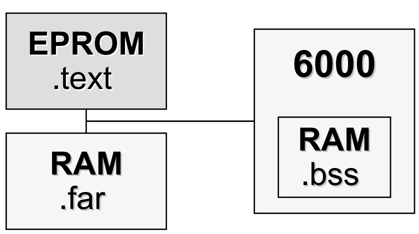
.text	Code
.bss	Global and Static Variables

- Physically these sections may be placed as follows:



- We may want global and static variables in two locations, e.g., on-chip and off-chip

.text	Code
.bss	Global and Static Variables
.far	Global and Static Variables



- `far` can be linked away from `.bss`

- Examples of using the `far` keyword

```
far short m;
short far n;
far short x_buf[1024];
```

- Custom named data sections, as defined in the memory map, can be created and used as follows:

```
#pragma DATA_SECTION (x, "myVar");
#pragma DATA_SECTION (y, "myVar");
int x[32];
short y;
```

- Code sections can also be created

```
#pragma CODE_SECTION(dotp, "myCode");
int dotp(a, x)
```

Syntax	Uses
`int dotp(a, x)`	`B _dotp`

`far int dotp(a, x)`	`MVKL _dotp, reg`
	`MVKH _dotp, reg`
	`B reg`

Use of Volatile

- The `volatile` key word is used to prevent the optimizer from unknowingly removing variables
  - A variable may be altered by some system means not available to the program itself
  - The compiler might think the variable is a constant
  - The `volatile` key word prevents this sort of optimization

Optimizing C Code

Memory Aliases

C Intrinsic Functions

- Intrinsic functions allow access to specific C6x hardware with the convenience of C, e.g., a saturated add can be greatly simplified

```
int sadd(int a, int b)
{
  int result;

  result = a + b;

  if (((a ^ b) & 0x80000000) == 0)
  {
    if ((result ^ a) & 0x80000000)
    {
      result = (a < 0) ? 0x80000000 : 0x7fffffff;
    }
  }
  return (result);
}
```

With Intrinsics

```
result = _sadd(a,b);
```

- We can think of intrinsic functions as a specialized function library written by TI
- A source for more information on using intrinsics is the TI C62x/67x *Programmers Guide*, SPRU 198G
- C intrinsic function are discussed in Chapter 3 Example 3.1 and Chapter 8 Section 8.2.2 and the examples of Section 8.4

- A simple dot product using intrinsics

```
//DotpN.c Multiplies two arrays, each array with N numbers

int dotp(short *a, short *b, int ncount); //function prototype
int dotpi(const short *a, const short *b, int ncount); //function
prototype

#include <stdio.h>              //for printf
#include "dotpN.h"     //data file of numbers
short x[count] = {x_array};    //declaration of 1st array
short y[count] = {y_array};     //declaration of 2nd array

main()
{
  int result = 0;    //result sum of products

  result = dotp(x,y,count);//call dotp function
  printf("result = %d (decimal) \n", result); //print result
  result = dotpi(x,y,count);//call dotp function
  printf("result2 = %d (decimal) \n", result); //print result
}

int dotp(short *a, short *b, int ncount) //dot product function
{
  int sum = 0;                  //init sum
  int i;

  for (i = 0; i < ncount; i++)
  {
     sum += a[i] * b[i];       //sum of products
  }
  return(sum);                  //return sum as result
}

//intrinsics dot product function
int dotpi(const short *a, const short *b, int ncount) {

  int sum = 0;                  //init sum
  int suml=0, sumh=0;
  int i;
```

```
  const int *i_a = (const int *)a;
  const int *i_b = (const int *)b;

  for (i = 0; i < (ncount >> 1); i++)
  {
     suml = suml + _mpy(i_a[i],i_b[i]);
     sumh = sumh + _mpyh(i_a[i],i_b[i]);
  }
  sum = suml + sumh;
  return(sum);                  //return sum as result
}
```

- The `_mpy` multiples the lower 16-bits and the `_mpyh` multiplies the high 16-bits
- The results are combined at the end, since using only one sum would inhibit parallelism and creates dependencies for the optimizer
- The include file `DotpN.h` contains defines for the data and the length *N*, denoted `count` in the program
- Examples were run on the DSK using different values for count and no optimization versus `-O3` optimization

Table 6.1: Cycle count comparisons for C intrinsics sum of products.

Code & opt level	$N = 16$	$N = 64$	C_{16}/MAC	C_{64}/MAC
Std. loop/no opt.	580	2068	36.25	32.31
Std. loop/-O3	74	107	4.63	1.67
Word wide/no opt.	507	1707	31.69	26.67
Word wide/-O3	70	94	4.38	1.47
Actual MAC	16	64	na	na

Intrinsics Reference Guide

C Compiler Intrinsic	Assembly Instruction	Description	Device
int **_abs(**int *src2***);** int**_labs(**long *src2***);**	**ABS**	Returns the saturated absolute value of src2.	
int **_abs2 (**int src**);**	**ABS2**	Calculates the absolute value for each 16–bit value.	'C64x
int **_add2(**int *src1*, int *src2***);**	**ADD2**	Adds the upper and lower halves of src1 to the upper and lower halves of src2 and returns the result. Any overflow from the lower half add will not affect the upper half add.	
int **_add4 (**int *src1*, int *src2***);**	**ADD4**	Performs 2s–complement addition to pairs of packed 8–bit numbers.	'C64x
int **_avg2 (**int *src1*, *int src2***);**	**AVG2**	Calculates the average for each pair of signed 16–bit values.	'C64x
unsigned **_avgu4(**unsigned, unsigned**);**	**AVGU4**	Calculates the average for each pair of unsigned 8–bit values.	'C64x
unsigned **_bitc4 (**unsigned src**);**	**BITC4**	For each of the 8–bit quantities in src, the number of 1 bits is written to the corresponding position in the return value.	'C64x
unsigned **_bitr (**unsigned src**);**	**BITR**	Reverses the order of the bits.	'C64x
uint **_clr(**uint *src2,* uint *csta*, uint *cstb***);**	**CLR**	Clears the specified field in src2. The beginning and ending bits of the field to be cleared are specified by csta and cstb, respectively.	
unsigned **_clrr(**uint *src1*, int *src2***);**	**CLR**	Clears the specified field in src2. The beginning and ending bits of the field to be cleared are specified by the lower 10 bits of the source register.	
int **_cmpeq2 (**int *src1*, int *src2*);	**CMPEQ2**	Performs equality comparisons on each pair of 16–bit values. Equality results are packed into the two least–significant bits of the return value.	'C64x
int **_cmpeq4 (**int *src1*, int *src2***);**	**CMPEQ4**	Performs equality comparisons on each pair of 8–bit values. Equality results are packed into the four least–significant bits of the return value.	'C64x
int **_cmpgt2 (**int *src1*, int *src2***);**	**CMPGT2**	Compares each pair of signed 16–bit values. Results are packed into the two least–significant bits of the return value.	'C64x
unsigned **_cmpgtu4 (**unsigned *src1*, unsigned *src2***);**	**CMPGTU4**	Compares each pair of unsigned 8–bit values. Results are packed into the four least–significant bits of the return value.	'C64x

Intrinsics Table (continued)

unsigned **_deal (**unsigned src**);**	**DEAL**	The odd and even bits of src are extracted into two separate 16–bit values.	'C64x
int **_dotp2 (**int *src1*, int *src2***);** long **_ldotp2 (**int *src1*, int *src2***);**	**DOTP2** **LDOTP2**	The product of signed 16–bit values in src1 is added to the product of signed 16–bit values in src2.	'C64x
int **_dotpn2 (**int *src1*, int *src2***);**	**DOTPN2**	The product of signed 16–bit values in src2 is subtracted from the product of signed 16–bit values in src1.	'C64x
int **_dotpnrsu2 (**int *src1*, unsigned *src2***);**	**DOTPNR-SU2**	The product of unsigned 16–bit values in src2 is subtracted from the product of signed 16–bit values in src1. 2^{15} is added and the result is sign shifted right by 16.	'C64x
int **_dotprsu2 (**int *src1*, unsigned *src2***);**	**DOTPR-SU2**	Adds the result of the product of the first signed pair and the unsigned second pair of 16–bit values. 2^{15} is added and the result is sign shifted right by 16.	'C64x
unsigned **_dotpu4 (**unsigned *src1*, unsigned *src2***);** int **_dotpsu4 (**int *src1*, unsigned *src2***);**	**DOTPU4** **DOTPSU4**	For each pair of 8–bit values in src1 and src2, the 8–bit value from src1 is multiplied with the 8–bit value from src2. The four products are summed together.	'C64x
int**_dpint(**double**)**;	**DPINT**	Converts 64-bit double to 32-bit signed integer, using the rounding mode set by the CSR register.	'C67x
int **_ext(**uint *src2,* uint *csta*, int *cstb***);**	**EXT**	Extracts the specified field in src2, sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right; csta and cstb are the shift left and shift right amounts, respectively.	
int **_extr(**int *src2*, int *src1***);**	**EXT**	Extracts the specified field in src2, sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right; csta and cstb are the shift left and shift right amounts, respectively.	
uint **_extu(**uint *src2,* uint *csta*, uint *cstb***);**	**EXTU**	Extracts the specified field in src2, zero-extended to 32 bits. The extract is performed by a shift left followed by a unsigned shift right; csta and cstb are the shift left and shift right amounts, respectively.	
uint **_extur(**uint *src2*, int *src1***);**	**EXTU**	Extracts the specified field in src2, zero-extended to 32 bits. The extract is performed by a shift left followed by a unsigned shift right; csta and cstb are the shift left and shift right amounts, respectively.	

Intrinsics Table (continued)

uint **_ftoi(**float**);**		Reinterprets the bits in the float as an unsigned integer. (Ex: _ftoi(1.0) == 1065353216U)	’C67x
int **_gmpy4 (**int *src1*, int *src2***);**	**GMPY4**	Performs the galois field multiply on 4 values in src1 with 4 parallel values in src2. The 4 products are packed into the return value.	’C64x
uint **_hi(**double**);**		Returns the high 32 bits of a double as an integer.	’C67x, ’C64x
double **_itod(**uint, uint**);**		Creates a new double register pair from two unsigned integers.	’C67x, ’C64x
float **_itof(**uint**);**		Reinterprets the bits in the unsigned integer as a float. (Ex: _itof(0x3f800000) == 1.0)	’C67x
double & **_memd8(**void * ptr**);**	**LDNDW/ STNDW**	Allows unaligned loads and stores of 8 bytes to memory.	’C64x
int & **_mem4(**void * ptr**);**	**LDNW/ STNW**	Allows unaligned loads and stores of 4 bytes to memory.	’C64x
long **_ldotp2 (**int *src1*, int *src2***);** int **_dotp2 (**int *src1*, int *src2***);**	**LDOTP2** **DOTP2**	The product of signed 16–bit values in src1 is added to the product of signed 16–bit values in src2.	’C64x
uint **_lmbd(**uint s*rc1,* uint *src2***);**	**LMBD**	Searches for a leftmost 1 or 0 of src2 determined by the LSB of src1. Returns the number of bits up to the bit change.	
uint **_lo(**double**);**		Returns the low (even) register of a double register pair as an integer.	’C67x, ’C64x
int **_max2 (**int *src1*, int *src2***);** int **_min2 (**int *src1*, int *src2***);** unsigned **_maxu4 (**unsigned *src1*, unsigned *src2***);** unsigned **_minu4 (**unsigned *src1*, unsigned *src2***);**	**MAX2** **MIN2** **MAXU4** **MINU4**	Places the larger/smaller of each pair of values in the corresponding position in the return value. Values can be 16–bit signed or 8–bit unsigned.	’C64x
double **_mpy2 (**int *src1*, int *src2***);**	**MPY2**	Returns the products of the lower and higher 16–bit values in src1 and src2.	’C64x
double **_mpyhi (**int *src1*, int *src2***);** double **_mpyli (**int *src1*, int *src2***);**	**MPYHI** **MPYLI**	Produces a 16 by 32 multiply. The result is placed into the lower 48 bits of the returned double. Can use the upper or lower 16 bits of src1.	’C64x
int **_mpyhir (**int *src1*, int *src2***);** int**_mpylir (**int *src1*, int *src2***);**	**MPYHIR** **MPYLIR**	Produces a signed 16 by 32 multiply. The result is shifted right by 15 bits. Can use the upper or lower 16 bits of src1.	’C64x

Intrinsics Table (continued)

double **_mpysu4 (**int *src1*, unsigned *src2***);** double **_mpyu4 (**unsigned *src1*, unsigned *src2***);**	**MPYSU4** **MPYU4**	For each 8–bit quantity in src1 and src2, performs an 8–bit by 8–bit multiply. The four 16–bit results are packed into a double. The results can be signed or unsigned.	’C64x
int **_mpy(**int *src1,* int *src2***);** int **_mpyus(**uint *src1,* int *src2***);** int **_mpysu(**int *src1,* uint *src2***);** uint **_mpyu(**uint *src1,* uint *src2***);**	**MPY** **MPYUS** **MPYSU** **MPYU**	Multiplies the 16 LSBs of src1 by the 16 LSBs of src2 and returns the result. Values can be signed or unsigned.	
int **_mpyh(**int *src1,* int *src2***);** int **_mpyhus(**uint *src1,* int *src2***);** int **_mpyhsu(**int *src1,* uint *src2***);** uint **_mpyhu(**uint *src1,* uint *src2***);**	**MPYH** **MPYHUS** **MPYHSU** **MPYHU**	Multiplies the 16 MSBs of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.	
int **_mpyhl(**int *src1,* int *src2***);** int **_mpyhuls(**uint *src1,* int *src2***);** int **_mpyhslu(**int *src1,* uint *src2***);** uint **_mpyhlu(**uint *src1,* uint *src2***);**	**MPYHL** **MPYHULS** **MPYHSLU** **MPYHLU**	Multiplies the 16 MSBs of src1 by the 16 LSBs of src2 and returns the result. Values can be signed or unsigned.	
int **_mpylh(**int *src1,* int *src2***);** int **_mpyluhs(**uint *src1,* int *src2***);** int **_mpylshu(**int *src1,* uint *src2***);** uint **_mpylhu(**uint *src1,* uint *src2***);**	**MPYLH** **MPYLUHS** **MPYLSHU** **MPYLHU**	Multiplies the 16 LSBs of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.	
int **_mvd (**int src**);**	**MVD**	Moves the data from the src to the return value over 4 cycles using the multipler pipeline.	’C64x
void **_nassert(**int**);**		Generates no code. Tells the optimizer that the expression declared with the assert function is true. This gives a hint to the compiler as to what optimizations might be valid (trip count information for software pipelined loops and about using word-wide optimizations).	
uint **_norm(**int *src2***);** uint **_lnorm(**long *src2***);**	**NORM**	Returns the number of bits up to the first nonredundant sign bit of src2.	
unsigned **_pack2 (**unsigned *src1*, unsigned *src2***);** unsigned **_packh2 (**unsigned *src1*, unsigned *src2***);**	**PACK2** **PACKH2**	The lower/upper half–words of src1 and src2 are placed in the return value.	’C64x
unsigned **_packh4 (**unsigned *src1*, unsigned *src2***);** unsigned **_packl4 (**unsigned *src1*, unsigned *src2***);**	**PACKH4** **PACKL4**	Packs alternate bytes into return value. Can pack high or low bytes.	’C64x

Intrinsics Table (continued)

unsigned **_packhl2 (**unsigned *src1*, unsigned *src2***);** unsigned **_packlh2 (**unsigned *src1*, unsigned *src2***);**	**PACKHL2** **PACKLH2**	The upper/lower half–word of src1 is placed in the upper half–word the return value. The lower/upper half–word of src2 is placed in the lower half–word the return value.	’C64x
double **_rcpdp(**double**)**;	**RCPDP**	Computes the approximate 64-bit double reciprocal.	’C67x
float **_rcpsp(**float**)**;	**RCPSP**	Computes the approximate 64-bit double reciprocal.	’C67x
unsigned **_rotl (**unsigned *src1*, unsigned *src2***);**	**ROTL**	Rotates src2 to the left by the amount in src1.	’C64x
double **_rsqrdp(**double *src***)**;	**RSQRDP**	Computes the approximate 64-bit double reciprocal square root.	’C67x
float **_rsqrsp(**float *src***)**;	**RSQRSP**	Computes the approximate 32-bit float reciprocal square root.	’C67x
int **_sadd(**int *src1,* int *src2***);** long **_lsadd(**int *src1,* long *src2***):**	**SADD**	Adds src1 to src2 and saturates the result. Returns the result.	
unsigned **_saddu4 (**unsigned *src1*, unsigned *src2***);**	**SADDU4**	Performs saturated addition between pairs of 8-bit unsigned values in src1 and src2.	’C64x
int **_sadd2 (**int *src1*, int *src2***);** int **_saddus2 (**unsigned *src1*, int *src2***);**	**SADD2** **SADDUS2**	Performs saturated addition between pairs of 16–bit values in src1 and src2. Src1 values can be signed or unsigned.	’C64x
int **_sat(**long *src2***);**	**SAT**	Converts a 40-bit value to an 32-bit value and saturates if necessary.	
uint **_set(**uint *src2,* uint *csta*, uint *cstb***);**	**SET**	Sets the specified field in src2 to all 1s and returns the src2 value. The beginning and ending bits of the field to be set are specified by csta and cstb, respectively.	
unsigned **_setr(**unsigned, int**);**	**SET**	Sets the specified field in src2 to all 1s and returns the src2 value. The beginning and ending bits of the field to be set are specified by the lower ten bits of the source register.	
unsigned **_shfl (**unsigned src**);**	**SHFL**	The lower 16 bits of src are placed in the even bit positions, and the upper 16 bits of src are placed in the odd bit positions.	’C64x
unsigned **_shlmb (**unsigned *src1*, unsigned *src2***);** unsigned **_shrmb (**unsigned *src1*, unsigned *src2***);**	**SHLMB** **SHRMB**	Shifts src2 left/right by one byte, and the most/least significant byte of src1 is merged into the least/most significant byte position.	’C64x

Intrinsics Table (continued)

int **_shr2 (**int *src1*, unsigned *src2***);** unsigned **_shru2 (**unsigned *src1*, unsigned *src2***);**	**SHR2** **SHRU2**	For each 16-bit quantity in src2, the quantity is arithmetically or logically shifted right by src1 number of bits. src2 can contain signed or unsigned values.	'C64x
int **_smpy(**int *src1*, int *sr2***);** int **_smpyh(**int *src1*, int *sr2***);** int **_smpyhl(**int *src1*, int *sr2***);** int **_smpylh(**int *src1*, int *sr2***);**	**SMPY** **SMPYH** **SMPYHL** **SMPYLH**	Multiplies src1 by src2, left shifts the result by one, and returns the result. If the result is 0x8000 0000, saturates the result to 0x7FFF FFFF.	
double **_smpy2 (**int *src1*, int *src2***);**	**SMPY2**	Performs 16-bit multiplication between pairs of signed packed 16-bit values, with an additional 1 bit left–shift and saturate into a double result.	'C64x
int **_spack2 (**int *src1*, int *src2***);**	**SPACK2**	Two signed 32–bit values are saturated to 16–bit values and packed into the return value.	'C64x
unsigned **_spacku4 (**int *src1*, int *src2***);**	**SPACKU4**	Four signed 16–bit values are saturated to 8–bit values and packed into the return value.	'C64x
int **_spint(**float**)**;	**SPINT**	Converts 32-bit float to 32-bit signed integer, using the rounding mode set by the CSR register.	'C67x
int **_sshvl (**int *src1*, int *src2***);** int **_sshvr (**int *src1*, int *src2***);**	**SSHVL** **SSHVR**	Shifts src2 to the left/right of src1 bits. Saturates the result if the shifted value is greater than MAX_INT or less than MIN_INT	'C64x
uint **_sshl(**uint *src2*, uint *src1***);**	**SSHL**	Shifts src2 left by the contents of src1, saturates the result to 32 bits, and returns the result.	
int **_ssub(**int *src1,* int *src2***);** long **_lssub(**int *src1,* long *src2***):**	**SSUB**	Subtracts src2 from src1, saturates the result size, and returns the result.	
uint **_subc(**uint *src1,* uint *src2***);**	**SUBC**	Conditional subtract divide step.	
int **_sub2(**int *src1*, int *src2***);**	**SUB2**	Subtracts the upper and lower halves of src2 from the upper and lower halves of src1, and returns the result. Any borrowing from the lower half subtract does not affect the upper half subtract.	
int **_sub4 (**int *src1*, int *src2***);**	**SUB4**	Performs 2s–complement subtraction between pairs of packed 8–bit values.	'C64x
int **_subabs4 (**int *src1*, int *src2***);**	**SUBABS4**	Calculates the absolute value of the differences for each pair of packed 8–bit values.	'C64x
unsigned **_swap4 (**unsigned src**);**	**SWAP4**	Exchanges pairs of bytes (an endian swap) within each 16–bit value.	'C64x

Intrinsics Table (continued)

unsigned **_unpkhu4 (**unsigned src**);**	**UNPKHU4**	Unpacks the two high unsigned 8–bit values into unsigned packed 16–bit values.	’C64x
unsigned **_unpklu4 (**unsigned src**);**	**UNPKLU4**	Unpacks the two low unsigned 8–bit values into unsigned packed 16–bit values.	’C64x
unsigned **_xpnd2 (**unsigned src**);**	**XPND2**	Bits 1 and 0 of src are replicated to the upper and lower halfwords of the result, respectively.	’C64x
unsigned **_xpnd4 (**unsigned src**);**	**XPND4**	Bits 3 through 0 are replicated to bytes 3 through 0 of the result.	’C64x

Note: Instructions not specified with a device apply to all ’C6000 devices.