# KeyStone Training

# KeyStone C66x CorePac Instruction Set Architecture

Rev1 Oct 2011

# Disclaimer

- This section describes differences between the TMS320C674x instruction set architecture and the TMS320C66x instruction set included in the KeyStone CorePac.
- Users of this training should already be familiar with the TMS320C674x CPU and Instruction Set Architecture.

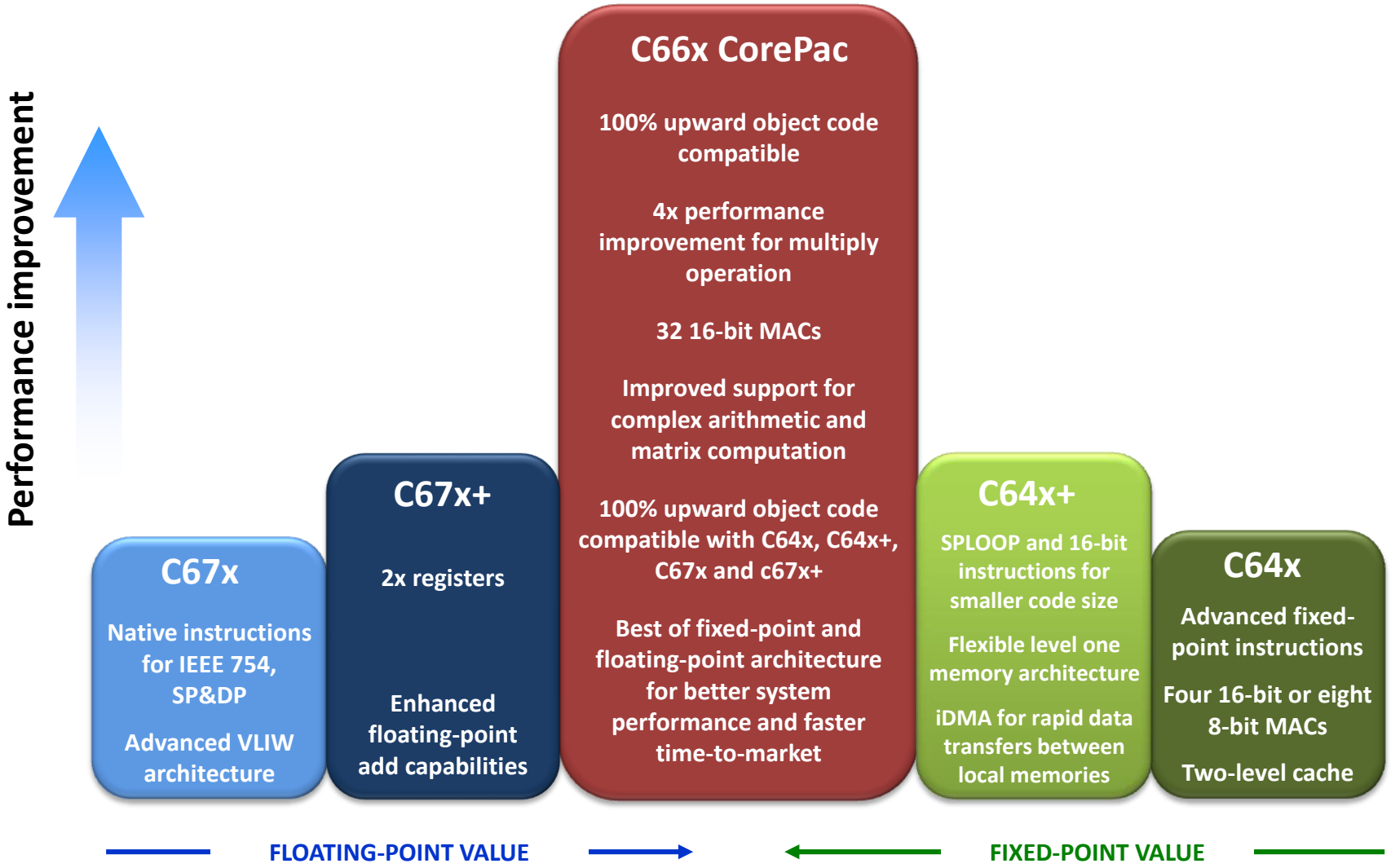
# Agenda

- Introduction
- Increased SIMD Capabilities
- C66x Floating-Point Capabilities
- Examples of New Instructions
- Matrix Multiply Example

# Introduction

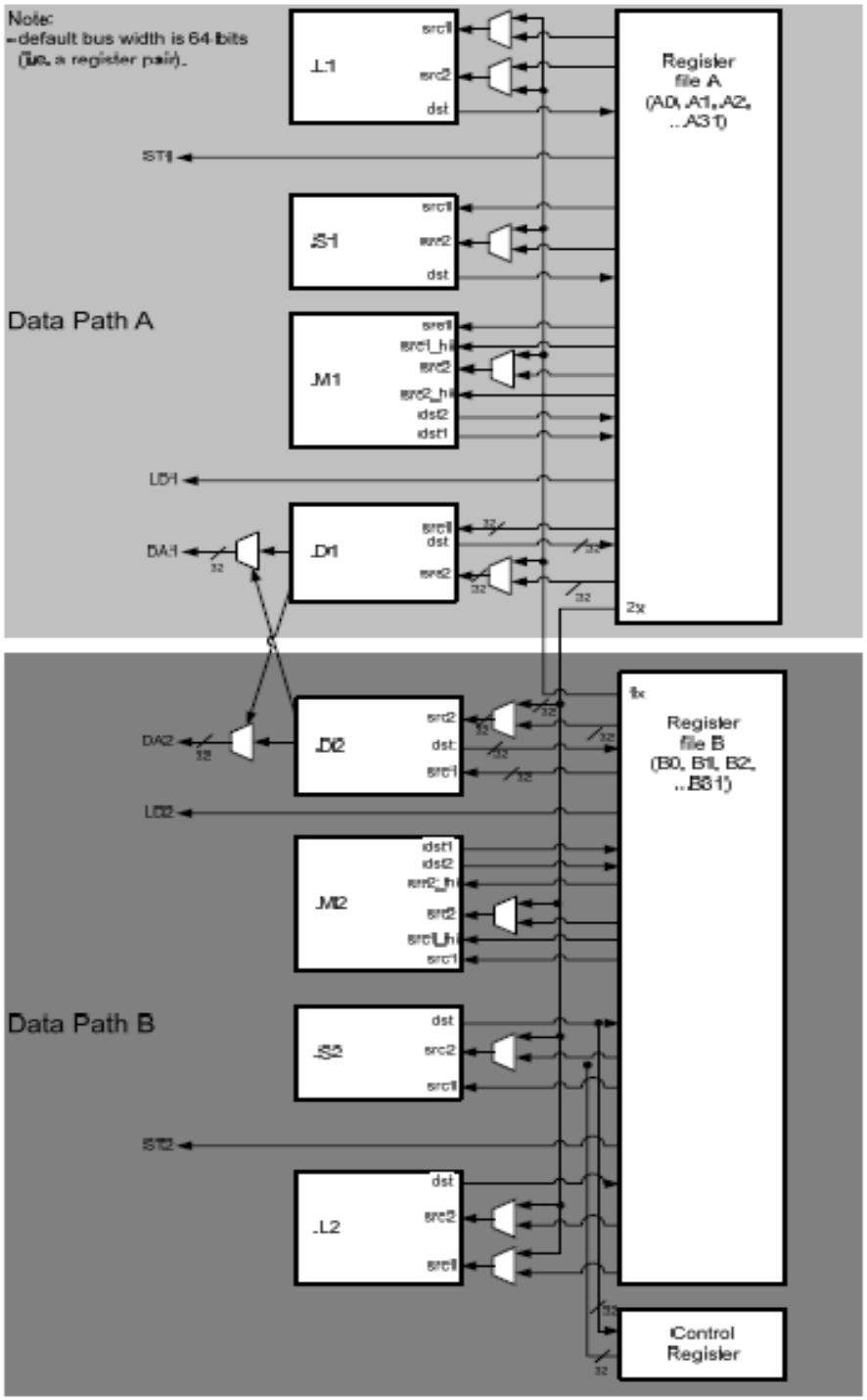
- Introduction
- Increased SIMD Capabilities
- C66x Floating-Point Capabilities
- Examples of New Instructions
- Matrix Multiply Example

# Enhanced DSP Core



# CPU Modifications

- Datapaths of the .L and .S units have been increased from 32-bit to 64-bit.
- Datapaths of the .M units have been increased from 64-bit to 128-bit.
- The cross-path between the register files has been increased from 32-bit to 64-bit.
- Register file quadruplets are used to create 128-bit values.
- No changes in D datapath.



# Core Evolution – Unified Architecture

- **Increased Performance**
- **Fixed/Floating Unification**

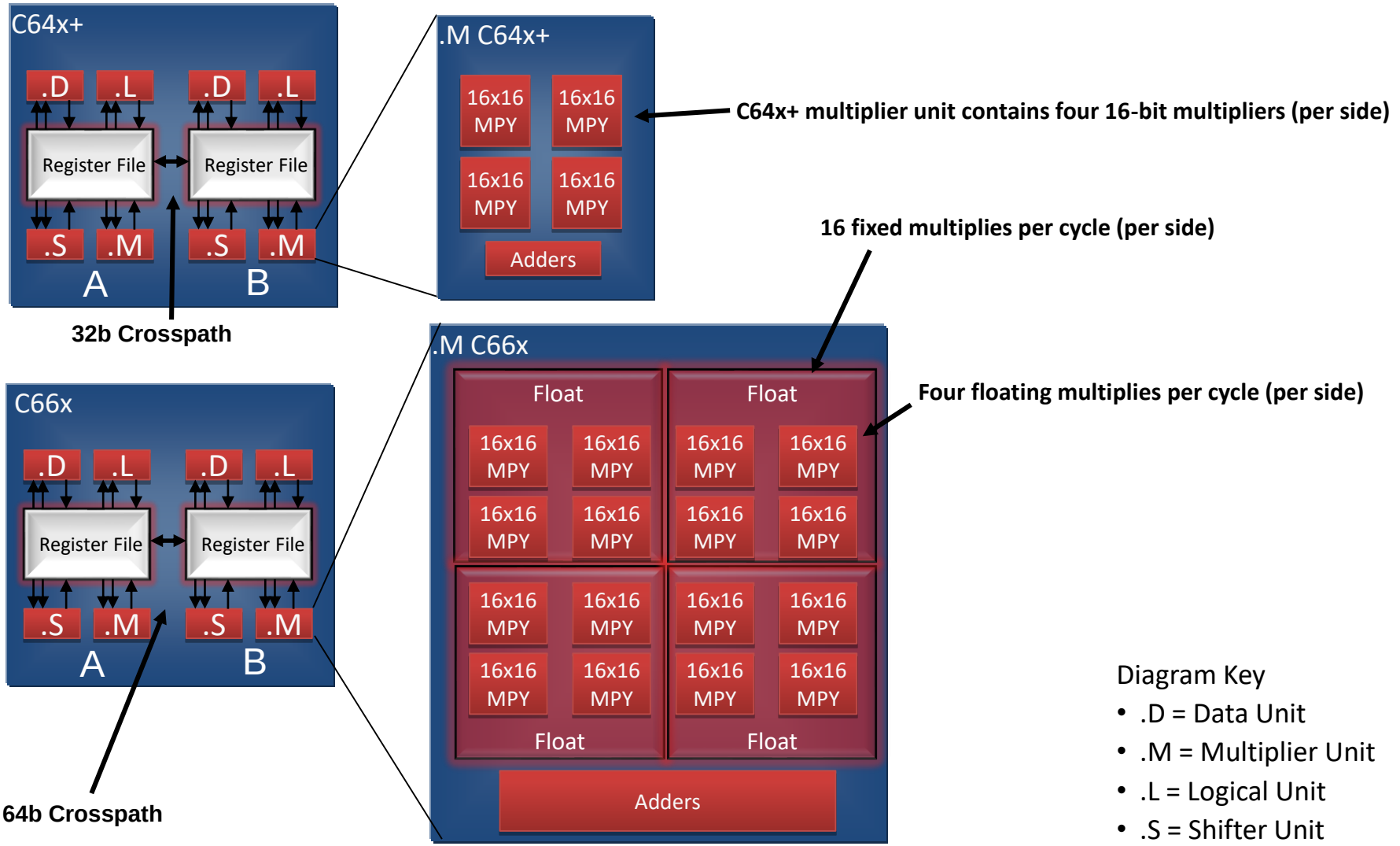



Diagram Key

- .D = Data Unit
- .M = Multiplier Unit
- .L = Logical Unit
- .S = Shifter Unit

# Increased Performance

- Floating-point and fixed-point performance is significantly increased.
  - 4x increase in the number of MACs
- Fixed-point core performance:
  - 32 (16x16-bit) multiplies per cycle.
  - Eight complex MACs per cycle
- Floating-point core performance:
  - Eight single-precision multiplies per cycle
  - Four single-precision MACs per cycle
  - Two double-precision MACs per cycle
  - SIMD (Single Instruction Multiple Data) support
  - Additional resource flexibility (e.g., the INT to/from SP conversion operations can now be executed on .L and .S units).
- Optimized for complex arithmetic and linear algebra (matrix processing)
  - L1 and L2 processing is highly dominated by complex arithmetic and linear algebra (matrix processing).

# Performance Improvement Overview

| | C64x+ | C674x | C66x |
|---|---|---|---|
| Fixed point 16x16 MACs per cycle | 8 | 8 | 32 |
| Fixed point 32x32 MACs per cycle | 2 | 2 | 8 |
| Floating point single-precision MACs per cycle | n/a | 2 | 8 |
| Arithmetic floating-point operations per cycle | n/a | 6 [1] | 16 [2] |
| Load/store width | 2 x 64-bit | 2 x 64-bit | 2 x 64-bit |
| Vector size<br>(SIMD capability) | 32-bit<br>(2 x 16-bit, 4 x 8-bit) | 32-bit<br>(2 x 16-bit, 4 x 8-bit) | 128-bit [3]<br>(4 x 32-bit, 4 x 16-bit, 4 x 8-bit) |

[1] One operation per .L, .S, .M units for each side (A and B)
[2] Two-way SIMD on .L and .S units (e.g., 8 SP operations for A and B) and 4 SP multiply on one .M unit (e.g., 8 SP operations for A and B).
[3] 128-bit SIMD for the M unit. 64-bit SIMD for the .L and .S units.

# Increased SIMD Capabilities

- Introduction
- Increased SIMD Capabilities
- C66x Floating-Point Capabilities
- Examples of New Instructions
- Matrix Multiply Example

# SIMD Instructions

- C64x and C674x support 32-bit SIMD:
  - 2 x 16-bit
    - Syntax: <instruction_name>**2** .<unit> <operand>
    - Example: MPY2
  - 4 x 8-bit
    - Syntax: <instruction_name>**4** .<unit> <operand>
    - Example: AVGU4
- C66x improves SIMD support:
  - Two-way SIMD version of existing instruction:
    - Syntax: **D**<instruction_name> .<unit> <operand>
    - Example: DMPY2, DADD

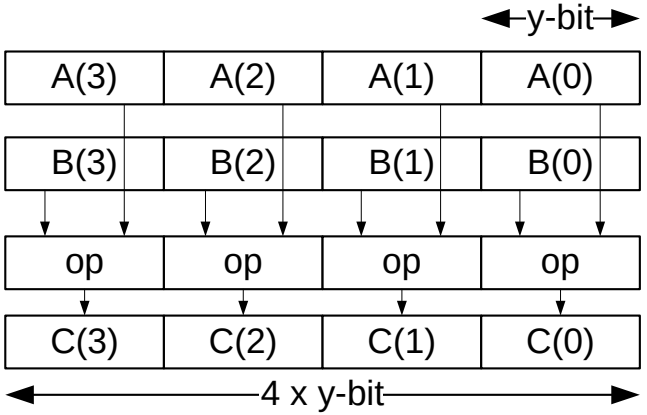
# SIMD Data Types

C66x supports various SIMD data types:

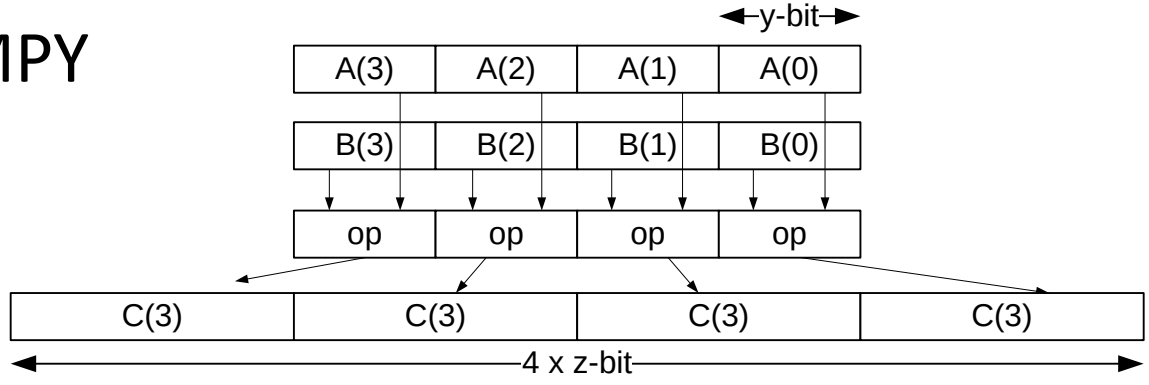
- 2 x 16-bit
  - Two-way SIMD operations for 16-bit elements
    - Example: ADD2
- 2 x 32-bit
  - Two-way SIMD operations for 32-bit elements
    - Example: DSUB
  - Two-way SIMD operations for complex (16-bit I / 16-bit Q) elements
    - Example: DCMPY
  - Two-way SIMD operations for single-precision floating elements
    - Example: DMPYSP
- 4 x 16-bit
  - Four-way SIMD operations for 16-bit elements
    - Example: DMAX2, DADD2
- 4 x 32-bit
  - Four-way SIMD operations for 32-bit elements
    - Example: QSMPY32R1
  - Four-way SIMD operations for complex (16-bit I / 16-bit Q) elements
    - Example: CMATMPY
  - Four-way SIMD operations for single-precision floating elements
    - Example: QMPYSP
- 8 x 8-bit
  - Eight-way SIMD operations for 8-bit elements
    - Example: DMINU4

# SIMD Operations (1/2)

- Same precision
  - Examples:
    - MAX2
    - DADD2
    - DCMPYR1

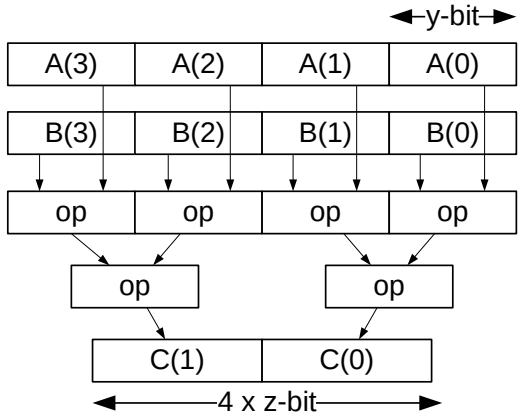



- Increased/narrowed precision
  - Example: DCMPY

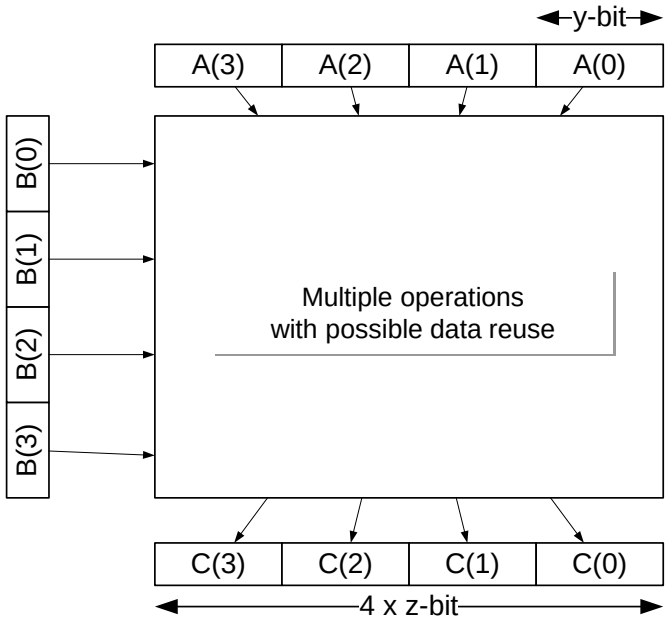


# SIMD Operations (2/2)

- Reduction
  - Example:
    - DCMPY, DDOTP4H




- Complex instructions
  - Example:
    - DDOTPxx, CMATMPY



# Registers and Data Types

- Introduction
- Increased SIMD Capabilities
- Registers and Data Types
- C66x Floating-Point Capabilities
- Examples of New Instructions
- Matrix Multiply Example

# Registers

| Register File | |
|---|---|
| A | B |
| A1:A0 | B1:B0 |
| A3:A2 | B3:B2 |
| A5:A4 | B5:B4 |
| A7:A6 | B7:B6 |
| A9:A8 | B9:B8 |
| A11:A10 | B11:B10 |
| A13:A12 | B13:B12 |
| A15:A14 | B15:B14 |
| A17:A16 | B17:B16 |
| A19:A18 | B19:B18 |
| A21:A20 | B21:B20 |
| A23:A22 | B23:B22 |
| A25:A24 | B25:B24 |
| A27:A26 | B27:B26 |
| A29:A28 | B29:B28 |
| A31:A30 | B31:B30 |

| Register File | |
|---|---|
| A | B |
| A3:A2:A1:A0 | B3:B2:B1:B0 |
| A7:A6:A5:A4 | B7:B6:B5:B4 |
| A11:A10:A9:A8 | B11:B10:B9:B8 |
| A15:A14:A13:A12 | B15:B14:B13:B12 |
| A19:A18:A17:A16 | B19:B18:B17:B16 |
| A23:A22:A21:A20 | B23:B22:B21:B20 |
| A27:A26:A25:A24 | B27:B26:B25:B24 |
| A31:A30:A29:A28 | B31:B30:B29:B28 |

- C66x provides a total of 64 32-bit registers, which are organized in two general purpose register files (A and B) of 32 registers each.
- Registers can be accessed as follows:
  - Registers (32-bit)
  - Register pairs (64-bit)
  - Register quads (128-bit)
- C66x provides explicit aliased views.

# The *__x128_t* Container Type (1/2)

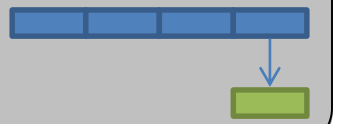
- To manipulate 128-bit vectors, a new data type has been created in the C compiler: *__x128_t.*
- C compiler defines some intrinsic to create 128-bit vectors and to extract elements from a 128-bit vector.

# The __*x128_t* Container Type (2/2)

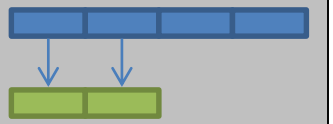
- Example:

Extraction

_get32_128(src, 0)

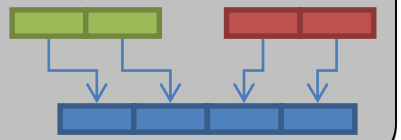


_hi128(src)

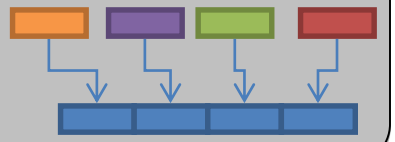


Creation

_llto128(src1,src2)



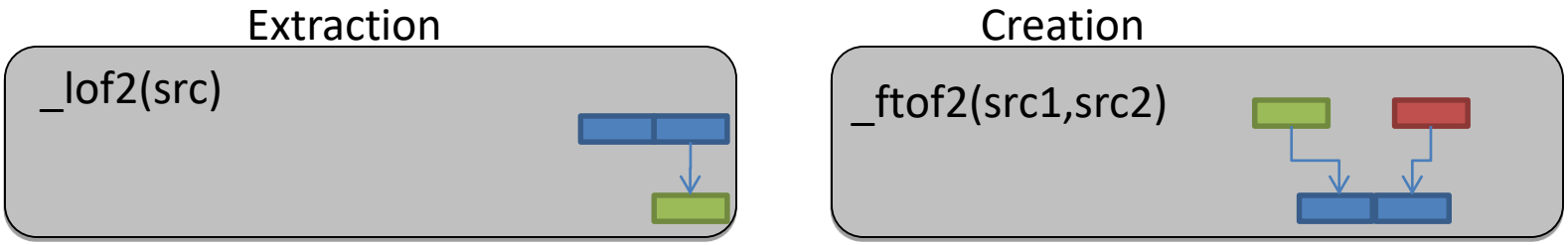
_ito128(src1,src2,src3, src4)



- Refer to the TMS320C6000 Optimizing Compiler User Guide for a complete list of available intrinsics to create 128-bit vectors and extract elements from a 128-bit vector.

# The *__float2_t* Container Type

- C66x ISA supports floating-point SIMD operations.
- *__float2_t* is a container type to store two single precision floats.
- On previous architectures (C67x, C674x) , the *double* data type was used as a container for SIMD float numbers. While all old instructions can still use the *double* data type, all new C66x instructions will have to use the new data type: *__float2_t.*
- The C compiler defines some intrinsic to create vectors of floating-point elements and to extract floating-point elements from a floating-point vector.



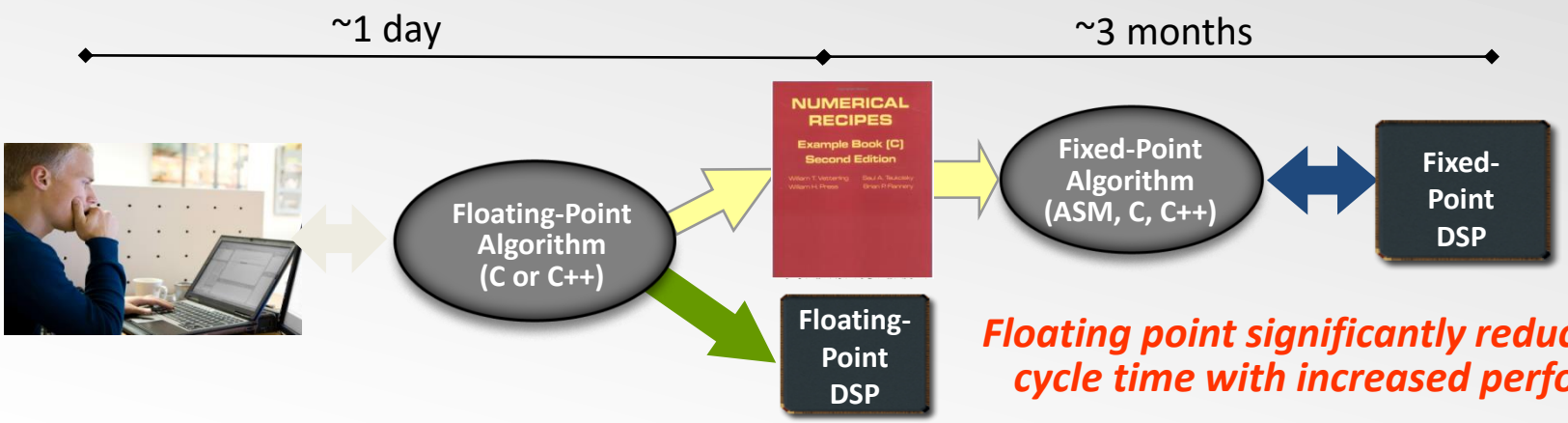
# C66x Floating Point Capabilities

- Introduction
- Increased SIMD Capabilities
- Register
- C66x Floating-Point Capabilities
- Examples of New Instructions
- Matrix Multiply Example

# Support for Floating Point in C66x

Floating point enables efficient MIMO processing and LTE scheduling:

- C66x core supports floating point at full clock speed resulting in 20 GFlops per core @ 1.2GHz.
- Floating point enables rapid algorithm prototyping and quick SW redesigns, thus there is no need for normalization and scaling.
- Use Case: LTE MMSE MIMO receiver kernel with matrix inversion
  - Performs up to 5x faster than fixed-point implementation
  - Significantly reduces development and debug cycle time




***Floating point significantly reduces design cycle time with increased performance***

# C66x Floating-Point Compatibility

- C66x is 100% object code compatible with C674x.
- A new version of each basic floating-point instruction has been implemented.

| | C674x | | C66x | |
|---|---|---|---|---|
| | Delay Slots | Functional Unit Latency | Delay Slot | Functional Unit Latency |
| MPYSP | 3 | 1 | 3 | 1 |
| ADDSP / SUBSP | 3 | 1 | 2 | 1 |
| MPYDP | 9 | 4 | 3 | 1 |
| ADDDP/SUBDP | 6 | 2 | 2 | 1 |

- The C compiler automatically selects the new C66x instructions.
- When writing in hand-coded assembly, the Fast version has to be specifically used.

  FADDSP / FSUBSP / FMPYSP / FADDSP / FSUBSP / FMPYSP

# C66x Floating Point

- C66x ISA includes a complex arithmetic multiply instruction, CMPYSP.
  - CMPYSP computes the four partial products of the complex multiply.
  - To complete a full complex multiply in floating point, the following code has to be executed:

```
CMPYSP .M1 A7:A6, A5:A4, A3:A2:A1:A0 ; partial products
DADDSP .M1 A3:A2, A1:A0, A31:A30     ; Add the partial products.
```

# Examples of New Instructions

- Introduction
- Increased SIMD Capabilities
- C66x Floating-Point Capabilities
- Examples of New Instructions
- Matrix Multiply Example

# New Instructions on .M Unit

| C/C++ Compiler Intrinsic | Assembly Instruction | Description |
|---|---|---|
| __x128_t _dcmpy(long long *src1*, long long *src2*); | DCMPY | Two-way SIMD complex multiply operations on two sets of packed numbers. |
| __x128_t _dccmpy(long long *src1*, long long *src2*); | DCCMPY | Two-way SIMD complex multiply operations on two sets of packed numbers with complex conjugate of src2. |
| long long _dcmpy(long long *src1*, long long *src2*); | DCMPYR1 | Two-way SIMD complex multiply operations on two sets of packed numbers with rounding. |
| long long _dccmpy(long long *src1*, long long *src2*); | DCCMPYR1 | Two-way SIMD complex multiply operations on two sets of packed numbers with rounding and complex conjugate of src2. |
| __x128_t _cmatmpy(long long *src1*, __x128_t*src2*); | CMATMPY | Multiply a 1x2 vector by one 2x2 complex matrix, producing two 32-bit complex numbers. |
| __x128_t _ccmatmpy(long long *src1*, __x128_t *src2*); | CCMATMPY | Multiply the conjugate of a 1x2 vector by one 2x2 complex matrix, producing two 32-bit complex numbers. |
| long long _cmatmpyr1(long long *src1*, __x128_t*src2*); | CMATMPYR1 | Multiply a 1x2 vector by one 2x2 complex matrix, producing two 32-bit complex numbers with rounding |
| long long _ccmatmpyr1(long long *src1*, __x128_t *src2*); | CCMATMPYR1 | Multiply the conjugate of a 1x2 vector by one 2x2 complex matrix, producing two 32-bit complex numbers with rounding. |
| __x128_t _dmpy2 (long long *src1*, long long *src2*); | DMPY2 | Four-way SIMD multiply, packed signed 16-bit |
| __x128_t _dsmpy2 (long long *src1*, long long *src2*); | DSMPY2 | Four-way SIMD multiply signed by signed with left shift and saturation, packed signed 16-bit |

# New Instructions on .M Unit

| C/C++ Compiler Intrinsic | Assembly Instruction | Description |
|---|---|---|
| long long _dxpnd2 (unsigned *src*); | DXPND2 | Expands bits to packed 16-bit masks |
| long long _ccmpy32r1 (long long *src1*, long long *src2*); | CCMPY32R1 | 32-bit complex conjugate multiply of Q31 numbers with Rounding |
| __x128_t _qmpysp (__x128_t *src1*, __x128_t *src2*); | QMPYSP | Four-way SIMD 32-bit single precision multiply producing four 32-bit single precision results. |
| __x128_t _qmpy32 (__x128_t *src1*, __x128_t *src2*); | QMPY32 | Four-way SIMD multiply of signed 32-bit values producing four 32-bit results. (Four-way _mpy32). |
| __x128_t _qsmpy32r1 (__x128_t *src1*, __x128_t *src2*); | QSMPY32R1 | 4-way SIMD fractional 32-bit by 32-bit multiply where each result value is shifted right by 31 bits and rounded. This normalizes the result to lie within -1 and 1 in a Q31 fractional number system. |

# New Instructions on .L Unit

| C/C++ Compiler Intrinsic | Assembly Instruction | Description |
|---|---|---|
| long long _dshr(long long *src1*, unsigned *src2*); | DSHR | Shift-right of two signed 32-bit values by a single value in the src2 argument. |
| long long _dshru(long long *src1*, unsigned *src2*); | DSHRU | Shift-right of two unsigned 32-bit values by a single value in the src2 argument. |
| long long _dshl(long long *src1*, unsigned *src2*); | DSHL | Shift-left of two signed 32-bit values by a single value in the src2 argument. |
| long long _dshr2(long long *src1*, unsigned *src2*); | DSHR2 | Shift-right of four signed 16-bit values by a single value in the src2 argument (two way _shr2(), four way SHR). |
| long long _dshru2(long long *src1*, unsigned *src2*); | DSHRU2 | Shift-right of four unsigned 16-bit values by a single value in the src2 argument (two way _shru2(), four way SHRU). |
| unsigned _shl2(unsigned *src1*, unsigned *src2*); | SHL2 | Shift-left of two signed 16-bit values by a single value in the src2 argument. |
| long long _dshl2(long long *src1*, unsigned *src2*); | DSHL2 | Shift-left of four signed 16-bit values by a single value in the src2 argument (two way _shl2(), four way SHL). |
| unsigned _dcmpgt2(long long *src1*, long long *src2*); | DCMPGT2 | Four-way SIMD comparison of signed 16-bit values. Results are packed into the four least significant bits of the return value. |
| unsigned _dcmpeq2(long long *src1*, long long *src2*); | DCMPEQ2 | Four-way SIMD comparison of signed 16-bit values. Results are packed into the four least significant bits of the return value. |
| void _mfence(); | MFENCE | Stall CPU while memory system is busy. |

# New Instructions on .L/.S Unit

| C/C++ Compiler Intrinsic | Assembly Instruction | Description |
|---|---|---|
| Double _daddsp(double src1, double src2); | DADDSP | Two-way SIMD addition of 32-bit single precision numbers. |
| Double _dsubsp(double src1, double src2); | DSUBSP | Two-way SIMD subtraction of 32-bit single precision numbers. |
| __float2_t _dintsp(long long src); | DINTSP | Converts two 32-bit signed integers to two single-precision float point values. |
| long long _dspint (__float2_t *src*); | DSPINT | Converts two packed single-precision floating point values to two signed 32-bit values. |

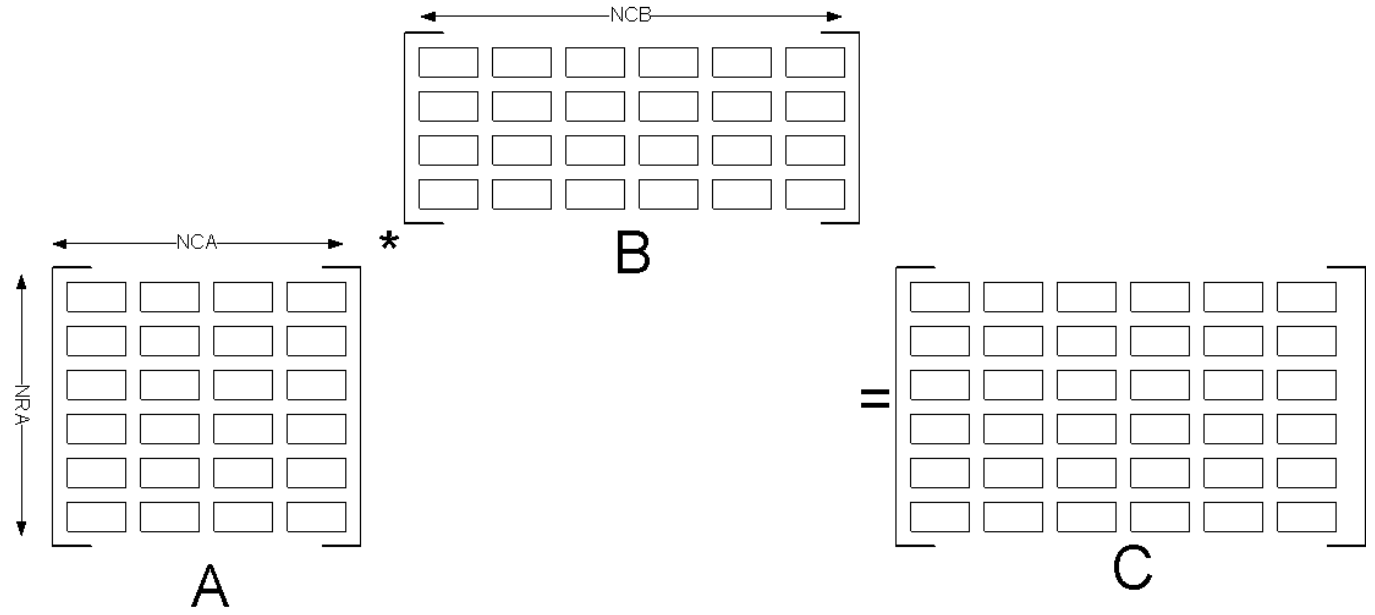
# Other New Instructions

- For an exhaustive list of the C66x instructions, please refer to the Instruction Descriptions in the TMS320C66x DSP CPU and Instruction Set.
- For an exhaustive list of the new C66x instructions and their associated C intrinsics, please refer to the *Vector-in-Scalar Support C/C++ Compiler v7.2 Intrinsics* table in the TMS320C6000 Optimizing Compiler User Guide.

# Matrix Multiply Example

- Introduction
- Increased SIMD Capabilities
- C66x Floating-Point Capabilities
- Examples of New Instructions
- Matrix Multiply Example

# Matrix Multiply



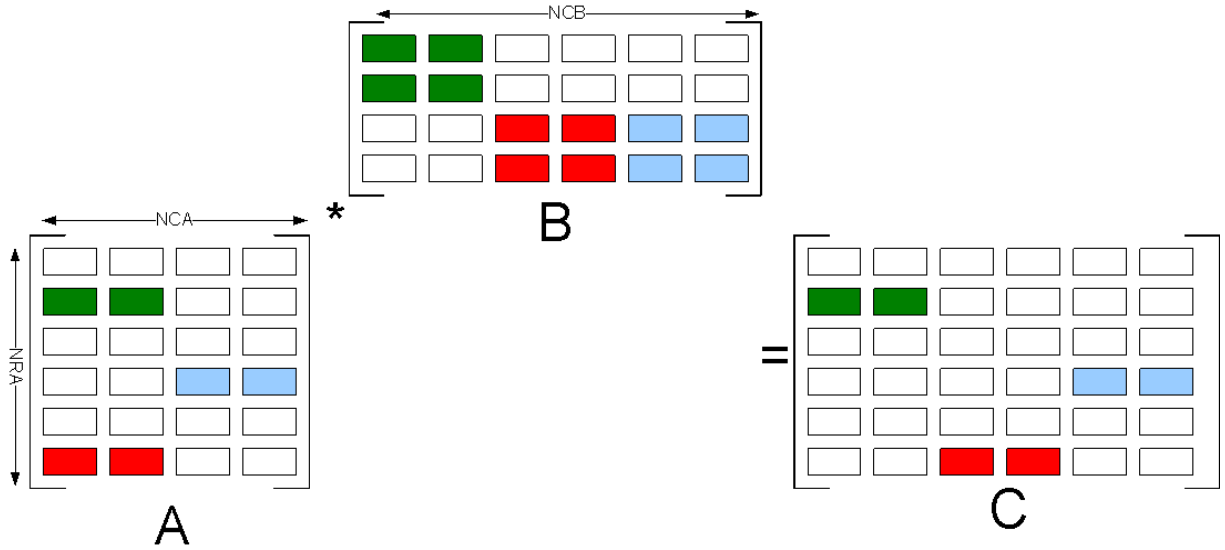

```
for (i=0; i<NRA; i++)
{
  for(j=0; j<NCB; j++)
  {
    sum_16.real = 0;
    sum_16.imag = 0;
    for (k=0; k<NCA; k++)
    {
      sum_16.real +=  ((a[i][k].real * b[k][j].real) - (a[i][k].imag * b[k][j].imag) + 0x00004000)>>15;
      sum_16.imag += ((a[i][k].real * b[k][j].imag) + (a[i][k].imag * b[k][j].real) + 0x00004000)>>15;
    }
    c_ref[i][j].real = sum_16.real;
    c_ref[i][j].imag = sum_16.imag;
  }
}
```

# Matrix Multiply

- CMATMPY instruction performs the basic operation:

$$\begin{bmatrix} C_{11} & C_{12} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

- Multiple CMATMPY instructions can be used to compute larger matrices.



# Matrix Multiply

- C66x C + intrinsic code:
  - Use of the __x128_t type
  - Use of some conversion intrinsic
  - Use of _cmatmpyr1() intrinsic

```
for (k=0; k<nca; k+=2)              {
  a_1_0   = *ptrA0++;          a_j1_j0 = *ptrA1++;
  a_3_2   = *ptrA2++;          a_j3_j2 = *ptrA3++;

  b_1_0   = _amem8((void *) &b[i + k*ncb]);
  b_3_2   = _amem8((void *) &b[i + k*ncb + 2]);
  b_k1_k0 = _amem8((void *) &b[i + (k+1)*ncb]);
  b_k3_k2 = _amem8((void *) &b[i + (k+1)*ncb + 2]);

  __x128_t b_k1_k0_b_1_0 = _llto128(b_k1_k0, b_1_0);
  __x128_t b_k3_k2_b_3_2 = _llto128(b_k3_k2, b_3_2);

  sum0 = _dsadd2(_cmatmpyr1(a_1_0,     b_k1_k0_b_1_0), sum0);
  sum1 = _dsadd2(_cmatmpyr1(a_j1_j0 ,  b_k1_k0_b_1_0), sum1);
  sum2 = _dsadd2(_cmatmpyr1(a_3_2,     b_k1_k0_b_1_0), sum2);
  sum3 = _dsadd2(_cmatmpyr1(a_j3_j2,   b_k1_k0_b_1_0), sum3);
  sum4 = _dsadd2(_cmatmpyr1(a_1_0,     b_k3_k2_b_3_2), sum4);
  sum5 = _dsadd2(_cmatmpyr1(a_j1_j0,   b_k3_k2_b_3_2), sum5);
  sum6 = _dsadd2(_cmatmpyr1(a_3_2,     b_k3_k2_b_3_2), sum6);
  sum7 = _dsadd2(_cmatmpyr1(a_j3_j2,   b_k3_k2_b_3_2), sum7);

  } /* End of loop on Nca */
```

# Matrix Multiply C66x Implementation Description

- C66x C + intrinsic code:

```
for (k=0; k<nca; k+=2)                    {
  a_1_0   = *ptrA0++;          a_j1_j0 = *ptrA1++;
  a_3_2   = *ptrA2++;          a_j3_j2 = *ptrA3++;

  b_1_0   = _amem8((void *) &b[i + k*ncb]);
  b_3_2   = _amem8((void *) &b[i + k*ncb + 2]);
  b_k1_k0 = _amem8((void *) &b[i + (k+1)*ncb]);
  b_k3_k2 = _amem8((void *) &b[i + (k+1)*ncb + 2]);

  __x128_t b_k1_k0_b_1_0 = _llto128(b_k1_k0, b_1_0);
  __x128_t b_k3_k2_b_3_2 = _llto128(b_k3_k2, b_3_2);

  sum0 = _dsadd2(_cmatmpyr1(a_1_0,     b_k1_k0_b_1_0), sum0);
  sum1 = _dsadd2(_cmatmpyr1(a_j1_j0 ,  b_k1_k0_b_1_0), sum1);
  sum2 = _dsadd2(_cmatmpyr1(a_3_2,     b_k1_k0_b_1_0), sum2);
  sum3 = _dsadd2(_cmatmpyr1(a_j3_j2,   b_k1_k0_b_1_0), sum3);
  sum4 = _dsadd2(_cmatmpyr1(a_1_0,     b_k3_k2_b_3_2), sum4);
  sum5 = _dsadd2(_cmatmpyr1(a_j1_j0,   b_k3_k2_b_3_2), sum5);
  sum6 = _dsadd2(_cmatmpyr1(a_3_2,     b_k3_k2_b_3_2), sum6);
  sum7 = _dsadd2(_cmatmpyr1(a_j3_j2,   b_k3_k2_b_3_2), sum7);

  } /* End of loop on Nca */
```

Most inner loop unrolled

Construct a 128-bit vector from two 64-bit

128-bit vector data type

Four-way SIMD saturated addition

Matrix multiply operation with rounding

# Matrix Multiply C66x Resources Utilization

C compiler software pipelining feedback:

- The TI C66x C compiler optimizes this loop in four cycles.
- Perfect balance in the CPU resources utilization:
  - Two 64-bit loads per cycle
  - Two CMATMPY per cycle
    - i.e., 32 16-bit x 16-bit multiplies per cycle
  - Eight saturated additions per cycle.

Additional examples are described in the application report, [Optimizing Loops on the C66x DSP](#).

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 132
;*      Loop opening brace source line   : 133
;*      Loop closing brace source line   : 196
;*      Known Minimum Trip Count         : 16
;*      Known Maximum Trip Count         : 16
;*      Known Max Trip Count Factor      : 16
;*      Loop Carried Dependency Bound(^) : 1
;*      Unpartitioned Resource Bound     : 4
;*      Partitioned Resource Bound(*)    : 4
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     3        1
;*      .D units                     4*       4*
;*      .M units                     4*       4*
;*      .X cross paths               3        2
;*      .T address paths             4*       4*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           5        5      (.L or .S unit)
;*      Addition ops (.LSD)          0        0      (.L or .S or .D unit)
;*      Bound(.L .S .LS)             4*       3
;*      Bound(.L .S .D .LS .LSD)     4*       4*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 4  Schedule found with 5 iterations in parallel
;*
```

# For More Information

- For more information, refer to the C66x DSP CPU and Instruction Set Reference Guide.
- For a list of intrinsics, refer to the TMS320C6000 Optimizing Compiler User Guide.
- For questions regarding topics covered in this training, visit the C66x support forums at the TI E2E Community website.