The objective of this presentation is to describe you the architectural changes of the new C66 DSP Core.

During this presentation, we are assuming that you're familiar with the C6000 in general.

Moreover, we will focus our attention on the differences between the C674x instruction set architecture and the new C66 ISA.

It is therefore recommended that you have a good understanding of the C674x before you start this presentation.

After a brief introduction, we will describe how the SIMD capabilities have been increased, how much fixed-point and floating-point performance can the new code deliver, and finally, we will walk through a concrete example showing how the new instructions can be used to optimize competition on intensive code.

So let's now move to the introduction section of this presentation.

TI has a long history of DSP cores based on the C6000 VLIW architecture.

Up to now, we developed two main families of C6000 DSPs, a floating-point family and a fixed-point family.

With the introduction of a new C66, we provide the best of a fixed-point and floating-point architecture.

We also significantly increase the performance with and architecture that can, for example, execute up to 32 to 16-bit MACs per cycle, and all this while we are able to maintain the full upward compatibility with the previous fixed and floating-point generations.

On the right hand side of this slide, you can observe a block diagram of the C66 CPU which includes two identical datapaths, the A datapath on the top and the B datapath at the bottom.

As for all previous versions of the C66, each datapath contains four functional units, the .L unit, the .S unit,
What we really changed on the C66 is the fact that we significantly increased the width of the SIMD instructions from 32-bit up to 64-bits for the .L and .S unit.

The width of the .M unit has been extended up to 128-bits and the .M unit can now accept 128-bit operands.

The size of the cross-path, which is used to exchange data between the A and B side has also been increased and extended to 64-bit.

In order to address the 128-bit containers, the notion of register quadruplets has been introduced, and we will go through that in a few slides.

And finally, we didn't perform any changes to the .D unit.

The .M unit has been significantly reworked to provide and increased performance, but also to optimize the architecture so that the unified datapath can be used for float and fixed-point operations.

Indeed, the C674x add dedicated datapaths for the float and fixed-point operations.

On the C66, the same logic is used for float and fixed-point multipliers.

Each .M unit of the C66 now contain 16 16-bit multipliers.

For reference on the C64x+, we only had four of these multipliers.

The 16-bit multipliers can be used to perform fixed-point operation or floating-point operations.

For example, the 16 multipliers can be used to perform four single-precision floating-point operations per cycle.

The fixed-point and floating-point performance has been quadrupled versus previous C674x architecture.

For example, we can now perform up to 32 fixed-points real multiplies per cycle or up to eight complex multiplies.
are cycle.

In floating-point, we can now perform eight single-precision multiplies per cycle, or four single-precision multiplies per cycle, or up to two double-precision multiplies per cycle.

Other improvements brought to the architecture regarding the floating-point capabilities include the possibility to have SIMD floating-point instructions and also some additional flexibility in the possible location of the instructions to the functional unit.

The performance has also been optimized by the addition of some new, specific instructions.

For example, some instructions had been added to the instruction set for the complex arithmetic and also to optimize the processing of the linear algebra.

On this table, we are comparing some performance matrix of the C64X+, the C674x, and C66 core.

On the first three rows, we can see that the multiply capability has been multiplied by four. The number of FLOP, or floating-point operations per cycle increased from 6 on the C674x up to 16 on the C66.

The load/store bandwidth did not change, but the vector processing capabilities, or what we call the SIMD capability, increased from 32-bit up to 128-bit for the .M unit on the C66 or up to 64-bit on the .L and .S unit on the C66.

You can find at the bottom of the slide the details of the floating-point operation precision calculation.

So now let's look a little bit more in detail at the SIMD changes we made on this new architecture.

On the previous C64x+ and C674x architectures, we are support 32-bit SIMD.
This means that in the 32-bit container, we can either have two packed 16-bit values, and in this case, the instruction mnemonic are typically terminated with 2 to indicate a two-way SIMD type of instruction.

Or we can also have 4 packed 8-bit values, and in that case, instruction mnemonic are terminated by a 4 to indicate a four-way type of instruction.

On the C66, we increased the SIMD capabilities, and we introduced two-way SIMD version of some existing instruction.

For example, we now have a double-add instruction which is a two-way SIMD, 32-bit addition.

Or we also have a double MPY2, DMPY2, instruction which is, in fact, a four-way SIMD 16-bit multiply instruction.

This slide describes most of the possible SIMD data types supported by the C66 with some instructions example.

We have seen previously that the C66 supports two-way SIMD for 16-bit value as the previous C 674x core.

We now also support two-way SIMD for 32-bit fixed and floating point value.

For example, DSUB is a two-way SIMD substrate 32-bit element, DCMPY performs two complex multiplications, and DMPYSP is a two-way SIMD version of the floating-point multiply instruction for single-precision.

Similarly, we support four-way SIMD for 16-bit element and four-way SIMD for 32-bit element.

For example, QMPY32R1 performed four 32.-bit fixed-point multiply with rounding in parallel, and QMPYSP is the four-way SIMD version of a floating-point multiply instruction.

That is so say that four single-precision multiplies are executed in parallel by this instruction.

We also support SIMD instruction for 8-bit value, and in that case, we can now support up to eight-way SIMD for 8-bit data.
For example, the instruction DMINU4 is an eight-way minimum value for the unsigned 8-bit value.

We can classify the SIMD operations.

The first class contains all instructions that are providing exactly the same number of elements and the element width is maintained.

For example, these are typically the instruction of this type are DMAX2, DADD2, DCMPYR1.

The second class contains all the instructions that maintain the number of elements, but the width of each element is increased.

Example of this class of instruction contains the DCMPY, which is a two-way SIMD complex multiply, which is taking input element of 16-bits, and which provides output element 32-bits.

Some SIMD instructions will provide results which contain a smaller number of elements than what was provided at the input.

This is typically the case when an instruction is performed being reduction.

That is to say, after performing operations SIMD lane by SIMD lane, the instructions performs operations between SIMD lane.

For example, all the DOT product or DOTP type of instructions fall int that category.

Finally, the last class of SIMD instruction is when the functional unit performs multiple operations with some data reutilization.

This is typically the case of a matrix complex multiply, CMATMPY.

This instruction will basically reuse the matrix rules to be multiplied with multiple matrix columns.

Let’s now look at the changes that have been made to the register file organization.

As on the C674x, the C66 provides a total of 64 32-bit registers organized as described on the table at
On the C674x, we are able to address individual registers or to address register pairs as we want to be able to manipulate 64-bit quantities.

On the C66 and with the addition of the 128-bit SIMD capabilities, we now want to be able to manipulate 128-bit quantities, and we can now access registers quads.

For example, on the top left, the intrinsics _get32_128 can be used to extract a 32-bit element from a 128-bit container.

On the bottom left, the intrinsics _hi128 can be used to extract the high 64-bits of the 128-bit container.

On the top right, the intrinsics _llto128 can be used to construct a 128-bit quantity from two 64-bit elements.

And at the bottom right, the intrinsics _ito128 can be used to construct a 128-bit vector from four 32-bit elements.

For more detailed information about the new intrinsics, you can refer to the related version of the C6000 Optimizing Compiler User's Guide.

As mentioned previously, the C66 now provided the possibility to execute SIMD operations for floating-point numbers.

Therefore, we have certified a new C data type for a 64-bit vector containing two single-precision floating-point number.

As previously, the C compiler also includes some intrinsics to extract single-precision numbers from 64-bit vector, and also to construct a 64-bit vector from two single-precision floating-point values.

So let's now look at the C66 floating-point capabilities.

Let's see why floating-point is important and how floating-point can be used to ease the development and increase the performance of advanced algorithms.
The natural approach to design a new algorithm is to describe it in float-point, and typically DSP software engineers initially
developed the function in floating-point format.

As floating-point DSPs were more expensive than fixed-point DSP, and also less powerful in terms of cycle and operating clock frequency, DSP software teams were spending a significant amount of time and resources to convert the floating-point algorithm into a fixed-point implementation.

While the floating-point implementation could be developed in the order of days, let’s say, the conversion from float fixed could typically take several weeks or months.

This is no more the case the C66 since we can now use both fixed and floating-point instructions in very high-performance DSP.

MIMO receivers are particularly important in the LTE physical layer receive processing chain.

MIMI receiver kernels rely on matrix calculations and matrix inversion in particular.

In this example of MIMO receiver, we have been able to significantly improve the performance of the algorithm typically by a factor of 5x, and this has been done with a significantly reduced development cycle time as well.

As we've seen on the previous slides, we have significantly increased the floating-point processing capability on the C66 versus the C674x by supporting SIMD floating-point instructions and also by adding some new floating-point instructions.

But also another improvement that we made is that we significantly increased the performance of whole existing C674x instruction.

For example, all floating-point instruction can now be fully pipelined.

This is to say that we can basically issue one floating-point instruction every single cycle.

Also, the number of cycles which is required by an instruction to generate a result, and that we commonly call
the number of delay slots, has been reduced.

For example, a double-precision multiply can now be completed after only three delay slots while it was requiring nine cycles to complete on the C674x architecture.

Also, the user or the compiler can now generate a double-precision multiply every single cycle.

To maintain a full object code compatibility, the fast version of whole previously defined instructions have been added to VISA, and the compiler will naturally and automatically select the new C66 instruction as the C66 switch is turned on.

In addition to the fast instructions and to the SIMD processing capability, some new instruction had been added to the floating-point instruction set.

For example, we've added a CMPYSP instruction to compute the multiplication of two complex, single-precision floating-point numbers.

However, it is important to note that unlike its fixed-point counterparts, the floating-point complex multiply instruction only computes the four partial products.

So if we multiply the complex number A plus GB times C plus GD, that means that this instruction will only compute the product AC minus BD, AD, and BC.

Therefore it is required that in addition to the CMPYSP instruction, we also compute an SIMD floating-point addition to handle the sum of a partial product, as it can be seen on this slide.

Let's quickly go through some examples of new instruction for each unit.

Let's look now at some examples of new instructions for the multiplier unit.

On the first row, we have the instruction DCMPY, which is two-way SIMD complex multiply operations that will operate on
two sets of packed numbers.

So typically, this instruction will take two 64-bit container in inputs for source 1 and source 2.

In each 64-bits, you have two complex numbers.

Each complex number is quantified on 16-bits or 16-bit I, 16-bit Q for each complex number, and this instruction will typically return 128-bit vector containing four 32-bits result.

A little bit below in the table, we also have the CMATMPY instruction, which is the instruction which is taking a 1 by 2 vector and comparing that value to a 2 by 2 complex matrix.

This institution will typically take the vector on 64-bits or in source 1, and take the 2 by 2 matrix in the 128-bit container in source 2.

This instruction will return the 128-bit vector that will contain basically four results. We can also notice that for most of the multiply instruction, we generally have the complex conjugate version.

So it is to say, we have a multiplication by a complex conjugate.

So this is, for example, the DCCMPY instruction, and we also have a flavor where we can perform the multiplication, and then perform a round and shift of its output to reduce the representation at the output of the instruction.

So that's, for example, all the instruction that you see with a suffix of AR1.

So DCMPYAR1 is the equivalent of DCMPY, but the results are rounded and shifted before being returned in the destination.

On this slide, we have some examples for four-way SIMD instructions.

For example, the QMPYSP is an instruction that will perform four single-precision multiply that will produce four single-precision results.

So typically, this instruction will take one of the two 128-bit inputs.

So each 128-bit vector will contain four single-precision numbers, and it will return a 128-bit vector containing four single-precision results.
We also have a fixed-point version which is a four-way SIMD multiply, and for that one, we have the round

and shift flavor, which is the QSMPY32R1.

From the .L unit, we have also added the support for SIMD on 64-bits.

For example, all the

shift operations are now SIMD.

So we have DSHR for two-way SIMD shift-right.

The same thing for the left side.

And we also have the DSHR2, which is a four-way right-shift for 16-bit quantities.

So typically, this instruction will take care 64-bits in inputs in source 1.

And the shift amount will be indicated in source 2, and we will return a 64-bit quantity containing four 16-bit

values that have been shifted by source 2.

We have also added the SIMD support for the compare instructions.

So we have DCMPGT, for example two

DCMP equal to.

This is some four-way SIMD comparison instructions and on the underneath

this is also where is located the MFENCE instruction, which is a particular instruction used to make sure that all

the memory transactions that have been issued by the core have been completed.

So this is an instruction that will be especially useful in the multi-core programming type of an environment or we

have multiple cores that are accessing a shared resource.

In terms of instructions that can be executed on both .L or .S, here again, we have added some SIMD

capability for the floating-point.

As an example here, you can see the Double ADDSP or Double sub-SP, which are respectively the addition and the
subtraction of two single-precision numbers together.

And in terms of data format conversion, we have also increased the throughput of the int to float and float to int conversion by adding some SIMD instruction for this type of operation.

So DINTSP, for example, will convert two 32-bit signed integers into some single-precision floating-point value in parallels.

So that significantly increases the throughput of conversion from float to fixed and fixed to float.

And we've seen in several examples that this is particularly useful as you want to use both fixed and float in different stages in your algorithm.

So we went through some examples of the instructions for on each unit, but for an exhaustive list of all the new C66 instructions, it would be good to refer to the instruction set description in the C66 DSP CPU and Instruction Set Guide.

Also, an important document to consult is the C6000 Optimizing Compiler User's Guide that will contain an exhaustive list of all the new C66 instructions and their associated C intrinsics.

So it's very useful as you want to program the C66 to refer to these documents so you will be able to find exactly the syntax of all intrinsics.

Let's now look at a concrete example where we would write the optimized code for a complex matrix multiplication.

So in this example, we will multiply the matrix A by the matrix B, resulting in the matrix C. Each element of these matrices are complex numbers.

And at the bottom of the slide, you can see the DNCC
code that we would typically write for such an operation with the free nested loop, the first loop, and the
time of row for the matrix A, the second loop for the number of columns for the matrix B, and
finally, the most inner loop, which is spanning across a number of columns
for the matrix A. On the C66, we define a new instruction, the CMATMPY, which basically
performs the medication of
a vector 1 by 2 by a matrix 2 by 2.
And this basic instruction can be used, in fact, in order to compute larger matrices, and as it is shown
on this diagram here, we can see that by using multiple instances of CMATMPY, we can, in fact, compute all
the elements of the matrix C. In order to optimize this small function from the matrix multiplication, we are going
to use C66 intrinsics.
And typically, in this example, we will see how we can use the _x128_t type, how we can use some
conversion intrinsics to build some 128-bit containers, and also how we can use the CMATMPY1 intrinsics.
So let's look a little bit more into the code itself.
As you can see, we've unrolled the most inner loop where we are basically possessing two columns at a time.
We can also see that we're using some intrinsics to construct 128-bits container from 264-bit value where we solve at
the beginning of a line here, you can see that we have defined the 128-bit vector data type.
We can also observe the usage of the CMATMPYR1 instruction, which is, in fact, the CMATMPY instruction with some rounding
capabilities, and also the DSADD2, intrinsics, which is a four-way SIMD saturated addition.
The C66 C compiler provides the capability to generate some feedback about the various optimization he made, and especially for
each loop, we are getting some information about the resource utilization and why the compiler has been able to software pipeline or not the loop.

So if we look at the report generated for the previously described loop, we can see that in this particular case, we achieve a preferred balance in terms of CPU resource utilization.

As you can see, for the .D unit, we have four cycles consumed on each side, and we also have four cycles consumed on each side for the .M unit.

So we have a preferred balance between the load/store capability of the architecture and the multiply capability of the architecture.

Other examples also describing to an application report which is focused on the C66 particularities, and so here on this slide, you also have a link to this application report which is available on our web.

So that's now the end of this training.

Here you have some links that are useful if you're looking for some more information about the C66 ISA.

And again, we would like to thank you for following this C66 KeyStone ISA training.