Blackfin **O**nline **L**earning & **D**evelopment

Presentation Title: Blackfin Processor Core Architecture Overview

Presenter: George Kadziolka

Chapter 1: INTRODUCTION

Subchapter 1a: Module Overview

Hello, this presentation is entitled The Blackfin Core Architecture Overview. My name is George Kadziolka. I'm the President of Kaztek Systems. I've been personally providing the training on the various Analog Devices architectures since 2000.

This module will introduce the Blackfin family which includes the Blackfin processors, tools and other development support that's available. Then we're going to get into the Blackfin Core Architecture and tell you a little bit about what's going on under the hood.

The Blackfin family will be the first stop, we're going to talk about the Blackfin processor, some of the origins, the tools that are available, then we'll get into the Blackfin core, talk about the arithmetic operation, data fetching, program flow control with the sequencer. We'll get into the Blackfin bus architecture and hierarchal memory structure. We'll talk about caching when we get to the memory management section.

Additional Blackfin core features will also be touched on including DMA, Power Management as well as some of the hardware debug support that's available.

Subchapter 1b: Blackfin Family Overview

The Blackfin family consists of a broad range of Blackfin processors to suit a number of different applications. There's also a wide variety of software development tools available, Analog Devices for instance provides VisualDSP++ and supports a uCLinux development environment. There's also a number of tools to evaluate hardware, for instance in the form of EZ-KITs, as well as a number of debug tools that are available to debug your application on your target platform

In addition to the support that Analog Devices provides, there's extensive third party support through the third party collaborative where you can find additional development tools. There's a number of vendors providing those. A variety of operating systems that can run on the Blackfin, including uCLinux. There's a number of companies providing network stacks, TCP/IP stacks, CAN stacks, a number of vendors providing hardware building blocks, CPU modules as well as just a vast variety of software solutions.

All Blackfin processors combine extensive DSP capability with high end micro controller functions, all in the same core. This allows you to develop efficient media applications while only using a single tool

chain. All Blackfin processors are based on the same core architecture so what this means is once you understand how one Blackfin processor works, you could easily migrate to other variants. The code is also compatible from one processor to another.

Processors vary in clock speed, the amount of on-chip memory, the mix of peripherals, power consumption, package size, and of course, price, so there's a very large selection of Blackfin variants to let you choose which Blackfin processor is most suitable for your particular application.

What we're looking at here is a variety of the peripherals that are offered within the Blackfin family, such as the EBIU or the micro processor style interface. We also have a PPI synchronous parallel interface typically used for video applications, serial ports or SPORTs, typically used for audio interfaces. GPIO pins are there, timer, UART, Ethernet capabilities built into several of the Blackfin variants as well as CAN or Controller Area Network and, USB. For a complete list of what the mixes are you're encouraged to go look at the Blackfin selection guide online at Analog's website.

Subchapter 1C: Embedded Media Applications

In a typical embedded media application the traditional model consists of having a micro controller providing operations such as control, networking, and a real time clock, usually it's a byte addressable part as well. Then you might have several blocks to perform signal processing functions, which could be audio compressions, video compression, beam forming or what have you. Then you typically have some sort of ASIC, FPGA or some other logic to provide interface to SDRAM or other peripherals. All these functions are combined into the Blackfin so now with a single core you can perform all these operations.

What does the Blackfin architecture mean to the developer? Well, by combining a very capable 32 bit controller with dual 16 bit DSP capabilities all in the same core, you can develop very efficient media applications such as multimedia over IP, digital cameras, telematics, software radio just to name a few examples. From a development perspective the single core means that you only have one tool chain to worry about so you don't have a separate controller, separate DSP, you have everything done on one core. Your embedded application which consists of both control and signal processing code is going to be dealt with the same compiler, so the result is very dense control code and high performance DSP code.

From a controller perspective the Blackfin has L1 memory space that can be used for stack and heap, so this means single cycle pushes and pops. There's dedicated stack and frame pointers, byte addressability is a feature, so we could be dealing with 8 bit, 16 bit, or 32 bit data but they're all residing at some byte address, as well as simple bit level manipulation.

From a DSP perspective the Blackfin has fast computational capability. But of course, that isn't very useful unless you can get data moved in and out efficiently as well. Unconstrained data flow is another key feature. A lot of the DSP operations are sums of products, so the intermediate sums require high dynamic range in order to store them, so extended precision accumulators is also another key feature. Efficient sequencing, being able to deal with interrupts efficiently, looped code, branch operations are dealt with efficiently in the Blackfin, as well as efficient IO processing. Things such as DMA offer an efficient way of communicating with the outside world. The DSP aspect of the Blackfin core is optimized to perform operations such as FFTs, as well as convolutions.

What we're looking at here is an example of a Blackfin processor, this is one of the members of the 54X family, but we're really trying to illustrate here is that the core, which consists of the Blackfin processor, internal memory, is common to all of the Blackfin variants. Once you learn how one Blackfin processor works you can easily migrate to others, as we mentioned earlier.

Chapter 2: The Blackfin Core
Subchapter 2a: Core Overview

Let's get into the Blackfin core. The core itself consists of several units, there's an arithmetic unit, which allows us to perform SIMD operations, where basically with a single instruction we can operate on multiple data streams. The Blackfin is also a load store architecture which means that the data that we're going to be working with needs to be read from memory and then we calculate the results and store the results back into memory.

There's also an addressing unit which supports dual data fetch, so in a single cycle we can fetch two pieces of information, typically data and filter coefficients. We also have a sequencer which is dealing with program flow on the Blackfin, and we just want to mention that there's several register files, for instance for data as well as for addressing, and we'll talk about those as we go through this section.

Subchapter 2b: Arithmetic Unit

The first stop is the arithmetic unit. The arithmetic unit of course performs the arithmetic operations in the Blackfin. It has a dual 40 bit ALU, which performs 16, 32 or 40 bit operations, arithmetic and logical. We also have a pair of 16 by 16 bit multipliers, so we can do up to a pair of 16 bit multiplies at the same time and when combined with the 40 bit ALU accumulators we can do dual MACS in the same cycle. There's also a barrel shifter which is used to perform shifts, rotates and other bit operations.

The data registers are broken up into sections , what we have here is the register file which consists of eight 32 bit registers from R0 to R7. They can be used to hold 32 bit values. Of course when we're doing DSP operations, a lot of our data is in 16 bit form so they can hold pairs of 16 bit values as well. The R0.H, R0.L for instance correspond to the upper and lower halves of the R0 register which could hold two 16 bit values. We also have a pair of 40 bit accumulators, A0 and A1, which are typically used with multiply accumulate operations to provide extended precision storage for the intermediate products.

We're looking at several ALU instructions, in this case 16 bit ALU operations. We're also taking a look at the algebraic syntax for the assembly language. As we can see the syntax is quite intuitive it makes it very easy to understand what the instruction is doing. In the first example we have a single 16 bit operation, R6.H = R3.H + R2.L:, so this is just the notation of the assembly language. It's very straight forward, what we're saying is we're going to take the upper half of R3 added to the lower half of R2 and place the result into the upper of R6.

Next example is a dual 16 bit operation. R6=R2+|-R3. What we're saying is we're going to read R2 and R3 as a pair of 32 bit registers, but that operator in between the +|- basically says it's a dual 16 bit operation. We're going to be taking the lower half of R3 subtracting it from the lower half of R2 with the result going to the lower half of R6. In addition we're going to take the upper half of R3 add it to the upper half of R2 and place the result in the upper half of R6. This effectively is a single cycle operation.

We also have an example of a quad 16 bit operation where what we're really doing is just combining two 16 bit operations in the same instruction. There's a coma here in between these two to indicate that we're trying to do those four operations in the same cycle. This is done when we want the sum and difference

between the pairs of 16 bit values in our two operands. Notice we have R0 and R1 on both sides of the coma in this example.

Of course we can also do 32 bit arithmetic operations as we see here. R6=R2+R3;. Similar example to what we saw previously, the difference is we only have a single operator so that tells the assembler that we're doing a single 32 bit operation. In other words R2 and R3 contain 32 bit values not pairs of 16 bit values. We can also do a dual 32 bit operation where we take the sum and difference between, in this case, R1 and R2. These also are effectively single cycle operations.

We mentioned about the multipliers, in this particular example we have a pair of MAC operations going on at the same time, so A1-=R2.H*R3.H, A0+=R2.L*R3.L; Again two separate multiply accumulate operations, R2 and R3 are the 32 bit registers containing the input data and we can mix and match which half registers we use as our input operands. Again this is effectively a single cycle operation, in addition this could be combined with up to a dual data fetch.

We also have a barrel shifter, the barrel shifter enables shifting, rotating any number of bits within a half register, 32 or 40 bit register, all in a single cycle. It's also used to perform individual bit operations on a 32 bit register in that register file, for instance we can individually set, clear, or toggle a bit without effecting any of the other bits in the register. We can also test to see if a particular bit has been cleared.

There's also instructions to allow field extract and field deposit. With this you specify the starting position of the field, how many bits long you're interested in, and you can pull that field out from a register and return in the lower bits of another register. Used for bit stream manipulation.

The Blackfin also has four 8 bit video ALUs, and the purpose of these devices is to allow you to perform up to four 8 bit operations in a single cycle. 8 bit because a lot of video operations typically deal with 8 bit values. Some examples here we can do a quad 8 bit add or subtract where we have four bites for each of the operands just add them or subtract them. We can do a quad 8 bit average, where we can average pairs of pixels or groups of four pixels. There's also an SAA instruction, Subtract, Absolute, Accumulate. What it does is it allows you to take two sets of four pixels, take the absolute value of the difference between them and accumulate the results in the accumulators. In fact four 16 bit accumulators are used. This is used for motion estimation algorithms.

All of these quad 8 bit ALU instructions still effectively take one cycle to execute, so they're quite efficient. What we're showing below is just the typical set up. We need our four byte operands, we start off with a two 32 bit register field, which provides 8 bytes, and we select any contiguous four bytes in a row from each of those fields. There's one operand the second one being fed to the four 8 bit ALUs.

The Blackfin also supports a number of other additional instructions to help speed up the inner loop of many algorithms, such as bitwise XOR , so if you're creating linear feedback shift registers, and you might be doing this if you're doing CRC calculations, or involved with generating pseudo random number sequences, there's instructions to basically speed up this process. Also the Bitwise XOR and Bit Stream Multiplexing are used to speed up operations such as convolutional encoders. There's also add on sign, compare, select which can be used to facilitate the Viterbi decoders in your applications.

There's also support for being compliant with the IEEE 1180 standard for two dimensional eight by eight discrete cosine transforms. The add subtract with pre-scale up and down, we can add or subtract 32 bit values and then either multiply it by 16 or divide by 16 before returning 16 bit value. There's also a vector search where you can go through a vector of data and a pair of 16 bit numbers at a time, search for the greatest or the least of that vector.

Subchapter 2c: Addressing Unit

Now we're moving on to the addressing unit. The addressing unit is responsible for generating addresses for data fetches. There are two DAGs, or Data Address Generator arithmetic units which enable generation of independent 32 bit addresses. 32 bits allows us to reach anywhere within the Blackfin's memory space, internally or externally. The other thing to make note of is we can generate up to two addresses, or perform up to two fetches at the same time.

We're looking at the address registers contained within the addressing unit, there are six general purpose pointer registers, P0 through P5. These are just used for general purpose pointing. We initialize them with 32 bit value to point to any where in the Blackfin's memory space, and we can use them for doing 8, 16, or 32 bit data fetches. We also have four sets of registers which are used for DSP style data fetches. This includes 16 and 32 bit fetches. DSP style means the types of operations that we typically use in DSP

operations such as dual data fetches, circular buffer addressing, or if we want to do, say, bit reversal, that would be done with these particular registers.

We also have dedicated stack and frame pointers as shown here. The addressing unit also supports the following operations; we can do addressing only where we specify index register, pointer register to point to some address in memory that we're going to fetch from. We can also do a post modified type of operation where the register that we specify will be modified after the data fetch is done automatically in order to prepare it for the next data fetch operation. Circular buffering, for instance, is supported using this method. We can provide an address with an offset where a point register might be pointed at the base address of the table and we want to fetch with some known offset from that base address. When we do that the modify values do not apply, no pointer update is performed.

We can do an update only, just to prepare a register for an upcoming fetch, or we can modify the address with a reverse carry add, if we're doing bit reversal types of data fetches.

All addressing on the Blackfin is register indirect, which means we always have to use a pointer register or index register to point to a 32 bit address we want to fetch from. If we are using index registers, I0 through I3, we're going to be using these for doing either 32 or 16 bit fetches from memory. The general purpose pointer registers, P0 through P5 can be used for 32, 16 and 8 bit fetches. Then stack and frame point registers are for 32 bit accesses only.

All the addresses provided by the addressing unit are byte addresses, whether we're doing 8, 16 or 32 bit fetches we're always pointing at the little end of that word that we want to fetch. Addresses need to be aligned. For instance, for fetching 32 bit words, the address must be a multiple of four.

Subchapter 2d: Circular Buffering

We're going to talk about circular buffering here. First of all, circular buffers are used a lot in DSP applications, where we have data coming in a continuous stream, but the filter that we're applying usually only needs to have a small portion of that, so our circular buffers are typically the size of the filter that we're using. What happens is we have a pointer, pointing to data that we're fetching from the circular buffer. As we step through the data we need to do a boundary check when we update the pointer to make

sure that we're not stepping outside the bounds of the circular buffer. If we do step outside we need to wrap the pointer to be put back in bounds of the circular buffer. This is done without any additional overhead in hardware in the Blackfin. We can also place these buffers anywhere in memory without restriction due to the base address registers.

In this particular example what we have is a buffer, which contains eleven 32 bit elements, and our goal is going to be to step through every fourth element and still stay inside the bounds of the circular buffer. What we're going to do is initialize the base and the index register. The base register is the address of where this buffer is in memory, and the index register is the starting address of where we're going to start fetching from. They're both initialized to zero in this example.

The buffer length will be set to 44, so there's 11 elements, but there are four bytes per elements, four 32 bit words per element. We initialize the buffer length with the number of bytes in the buffer. We also want to step through every fourth element, so because the elements are 32 bits wide, we have to provide a byte offset of 16 in this case.

The first access will occur from address 0 and when we're done we're going to bump the pointer and be looking at address 0x10 when we're done. On the second access we'll fetch from 0x10 , bump the pointer again automatically to be pointing at 0x20 . There we are. On the third fetch we'll fetch from 0x20 hex, but when we apply the 0x10 offset, the 0x30 will be out of bounds so what happens is the address generator wraps the pointer to be back in bounds. Where's it wrap to? Well we want to be four away, but still inside bounds of the circular buffer. If we're starting at 0x20 , one, two, three, four, we get wrapped to address 0x4. This is handled automatically in the address generator.

Subchapter 2e: Sequencer

The last block that we're going to talk about in the core is the sequencer itself. The sequencer's function is to generate addresses for fetching instructions. It uses a variety of registers in which to select what's the next address that we're going to fetch from. The Sequencer also is responsible for alignment. It aligns instructions as they're fetched. The Sequencer always fetches 64 bits at a time from memory, and we'll talk about this later, but there's different size op-codes, and what it does is it basically realigns 16, 32 or 64 bit op-codes before sending them to the rest of the execution pipe line.

The Sequencer is also responsible for handling events. Events could be interrupts or exceptions and we'll talk about that shortly. Also any conditional instruction execution is also handled by the Sequencer. The Sequencer accommodates several types of program flow, linear flow of course is the most common where we just execute code linearly in a straight line fashion. Loop code is handled where you might have a sequence of instructions that you want to execute over and over again, some counted number of times. We can have a permanent redirection program flow with a jump type instruction where we just branch to some address in memory and just executing from that point on. Sub-routine types of operations where you call some sub-routine , some address, and we execute and we return back to the address following the call instruction.

Interrupt is like a call, it's a temporary redirection except it's governed typically by hardware. An interrupt comes in during the execution of an instruction, then we vector off to the interrupt service routine, execute the code, the ISR, and return to the address following the instruction that got interrupted in the first place.

We also have an idle instruction. What the idle does essentially just halts the pipeline, all the instructions that are in there get executed, and then we just sort of wait for either wake up event or an interrupt to come along and then we'll just continue fetching instructions from that point on. It is actually also tied into the Power Management and we'll touch on that a little bit later on.

The Sequencer has a number of registers that are used in the control of program flow on the Blackfin. There's an arithmetic status register, any time we do an arithmetic operation you could be affecting bits in here. AZ for zero, AN for negative for instance. The Sequencer uses these status bits for the conditional execution of the instructions. We also have a number of return address registers, RETS, RETI, RETX is shown down here, for sub-routine or any number of events. These hold the 32 bit return address for any flow interruption, either a call or some sort of interrupt. We also have two sets of hardware loop management registers; LC, LT, LB, so loop counter, address at the top of the loop, address at the bottom of the loop. They're used to manage zero overhead looping. Zero overhead looping means once we set up these registers, we spend no CPU cycles decrementing the loop counter, checking to see if it's reached zero, branching back to the top of the loop. This is all handled without any over head at all, just by simply initializing these registers. Two sets of these registers allow us to have two nested levels of zero overhead looping.

The Blackfin also has an instruction pipeline and this of course is responsible for the high clock speeds that are possible. The one thing to notice is the pipeline is fully interlocked. What this means is in the event of a data hazard, such as a couple of instructions going through, one instruction sets up a register that the next one is supposed to be using, if it's not ready because of where it is in the pipeline, the Sequencer automatically inserts stall cycles and holds off the second instruction until the first one is complete.

Of course the C compiler is very well aware of the pipeline and will automatically rearrange instructions in order to eliminate stalls or reduce the number of stall cycles.

Quickly going through the different stages, we've got three instruction fetch stages for the addresses to put out on the bus, we wait for the instruction to come back and the instruction comes in at the IF3 stage. Instructions are decoded. Address calculation stage is where we calculate the address for any data fetches, so for instance if we have a pre-modify this is where the offset would be added to the pointer register. We have a couple of data fetch stages. DF1, if we're performing a data fetch, this is where the address of the memory location we want to fetch data from is put out on the bus. If we have an arithmetic operation DF2 is the stage where the registers are read to get the operands.

Then we have two cycles for execute, EX1, EX2, this is where we start our two cycle operation for instance like a multiple accumulate. EX2, any single cycle operations or the second part of multiple accumulate would be done here. Finally the last stage is the write back stage where any data that was read from memory or any results from any calculations are going to be committed at this point in time.

Once the pipeline is filled then what we see in this case here at cycle N+9. Once the pipeline is filled every clock tick results in another instruction exiting the pipeline, or leaving the writeback stage. Effectively every clock tick, another instruction is executed, so this is where we get our one instruction per core clock cycle execution speed.

Subchapter 2f: Event Handling

Blackfin also handles events. Events can be categorized as either interrupts, typically hardware generated DMA transfers completed, timer has reached zero. You can also generate software interrupts. Other events would include exceptions. These could be either as a result in an error condition, let's say an overflow or a CPLB miss, or something like that, or the software requesting service. There's a way for software to generate exceptions intentionally.

The handling of events is split between the core event controller and the system interrupt controller. The core event controller and core clock domain, system interrupt controllers and the SCLK domain. The core event controller has 16 levels associated with it and deals with the requests on a priority basis. We'll see this priority on the next slide. The nine lowest levels of the core interrupt controller are basically available for the peripheral interrupt request to be mapped to. Every level has a 32 bit interrupt vector register associated with it. This can be initialized with the starting address of our ISR so that when that particular level is triggered, that's the address that we're going to start fetching code from.

We also have the SIC, or the System Interrupt Controller portion of the Interrupt handler. This is where we enable which peripheral interrupt request we want to allow to go through. Also, the mapping of a particular peripheral interrupt request to a specific core interrupt level happens here. What this allows us to do is change the priority of a particular peripheral's interrupt request. What we see here is an example using the BF533 interrupt handling mechanism. The left hand side is the system interrupt controller side, and here we have the core interrupt controller side. Again 16 levels at the core and what we're showing is that these are prioritized. Level 0 or IVG0 has the highest priority in the system, level 15 has the lowest. In case multiple interrupts came in at the same time, the highest one would be given the nod.

What we also see over here is we have quite a few different peripheral interrupt requests possible, and we end up mapping multiple peripheral interrupt requests into, say, one GP level because we have more peripheral interrupt requests then we have general purpose levels left over. We also see the default map, in other words coming up out of a reset. These particular peripheral interrupt requests are mapped to these specific core levels. What you can do is if in your particular application you've determined that say programmable flag interrupt A must have a higher priority then say the real time clock interrupt, you could remap using the interrupt assignment registers.

The Blackfin also has variable instruction lengths. There's three main op-code lengths that Blackfin deals with. 16 bit instructions include most of the control type instructions, as well as data fetches, so these are all 16 bits long in order to improve code density. We also have 32 bit instructions. These include most of the control types of instructions that have immediate values, for instance loading register with immediate value or things of that sort. Most arithmetic instructions are also 32 bits in length.

There's also a multi issue 64 bit instruction, and this allows us to combine a 32 bit op-code with a pair of 16 bit instructions. Typically an arithmetic operation with one or two data fetches. What we do is when we combine them as we see in the example below, we have a dual multiple accumulate operation in parallel with a dual 32 bit data fetch. We use this parallel pipe operator to tell the assembler that we're trying to combine these operations at the same time. What happens is as a 64 bit op-code, these three instructions go through the pipeline together so they all get executed at the same time.

Instruction packing. When code is compiled and linked, the linker places the code into the next available memory space so instructions aren't padded out, for instance, to the largest possible instruction that's handled on the Blackfin. What this means is there's no wasted memory space. As a programmer you don't have to worry about this, we mentioned earlier the sequencer has an alignment feature so as we read the 64 bits of instruction from memory, it performs the alignment for us. So it pulls out the individual 16, 32, and 64 bit op-codes. If they happen to straddle an octal address boundary it'll also realign that before passing it on to the rest of the pipeline.

Subchapter 2g: FIR Filter example

Blackfin is optimized for doing operations such as FIR filtering or FFTs. What we have here is an example of a 16 bit FIR filter. In fact what we're doing is we have an input data stream and we're applying two filters to the same data stream. We're showing here that we're using R0 and R1 as the two input registers , R0 contains our two pieces of 16 bit input data, R1 contains our 16 bit filter coefficients. As we're going through here, here's our data in our delay line, our filter coefficient is being fetched from the buffer, first filter coefficient will be used with the first two pieces of data and what we're going to do is start fetching another piece of data while we use the next filter coefficient with two other pieces of data. Our code example here at the bottom just shows our two multiply accumulate instructions in parallel with either single or a dual data fetch. In addition to doing those two data fetches we're also setting up the pointers to point to the next pieces of data when we're complete.

Chapter 3: The Blackfin Bus and Memory Architecture
Subchapter 3a: Overview

Next we're going to get into the bus and memory architecture of the Blackfin. The Blackfin uses a memory hierarchy with the main goal of achieving performance comparable to what we have in the L1 memory, with an overall cost of that of the least expensive memory, which is the L3 in this case. What do we mean by hierarchal memory? First of all; in a hierarchal memory system you basically have levels of memory. The memory that's closest to the core is faster, but you don't have as much of it. L1 memory has the smallest capacity but items that are in L1 can be fetched single cycle. Some of the Blackfin variants have L2 memory, this is a larger block of memory but it's further away from the core, latency is a little bit greater. All Blackfins support L3 or SDRAM so this is where the bulk of your code or data could be stored. Very large capacity but also high latency associated with it.

Portions of L1 can also be configured as cache, and this allows the memory management unit to pre-fetch and store copies of code or data that might be in L2 or L3, stored in L1 that way you get L1 performance. What we're looking at here is the internal bus structure of, we're looking at the [ADSP-BF533] processor as an example in this case. We're going to look at the modified Harvard architecture as well as talk about some of the performance aspects of the fetches work.

Subchapter 3b: Bus Structure

First off we have the two different domains, the core (CCLK) and system (SCLK) clock domain. When we talk about a processor running at 600MHz for instance we're talking about the CCLK frequency. That's the rate at which the core can fetch instructions, fetch data from L1 memory. What we see here are the data paths from L1 to the core. In a single cycle we can fetch 64 bits worth of instruction and up to two data fetches, up to two 32 bit fetches from L1 can occur at the same time. In many applications the code and data for instance could fit into L1 and that's fine, so you're always running at the highest performance possible. Of course if you have a large application you could store parts of your code and data in external memory, in SDRAM. Because we can generate addresses for any where within the Blackfins 32 bit memory space, the address for instruction could very well be pointed to something in SDRAM. If that was the case , the core is fetching say 64 bits from SDRAM, we're going to go across the external access bus,

through the EBIU to the external port bus which in this example is a 16 bit wide bus. To fetch that same 64 bit instruction we're going to be requiring four fetches from SDRAM. We're in the SCLK domain, the SCLK domain in the BF533 is limited to 133MHz, so if we're running at 600MHz this will top out at 120MHz, so you have a five to one difference in clock rates. Those four cycles that we just spent fetching from SDRAM actually cost us 20 core clock cycles. Of course that assumes we're on the right page in SDRAM. If we have to pre-charge a page and activate a new one, that's going to add a few more cycles.

Subchapter 3C: Configurable Memory

What we see here is that even though we have the SDRAM it could be potentially expensive executing out of SDRAM. What we do have is a couple of mechanism that allow us to get around this. DMA of course is one where you can move in overlays for instance or data, you can do memory to memory transfers from SDRAM to L1. There's also caching that we'll talk about momentarily.

The Blackfin has configurable memory and as we saw from the previous diagram the best over all performance can be achieved when the code that you want to fetch or the data that you want to process resides in L1. Now there's two methods that we can use to fill the L1 memory, caching and dynamic downloading, and Blackfin processor supports both. Micro controller programmers typically use the caching method where you just set up the cache in the memory management unit and any large programs will benefit from this, will just automatically store or cache copies of the code in L1.

DSP programmers have typically used Dynamic Downloading or dealing with code overlays where you anticipate a function that you want to execute and what you do is you set up a DMA transfer in the background, move them to L1 so that by the time it's there, now you can execute it at the L1 speed.

The Blackfin processor allows the programmer to choose one or even both methods in order to optimize system performance. Again portions of L1 instruction and data memory can be configured to be used as either SRAM or cache . SRAM means that that block of memory appears in your memory map and you have full access to it just by assigning a pointer to it. If its' configured as cache you no longer have direct access to it, but instead the memory management unit decides what goes in to that block.

Once you enable the 16K blocks this does reduce the total amount of L1 memory available as SRAM, however you still have in the case of instructions, you still have code memory left over that you can still put key interrupt service routines or functions such as that and have it resident in L1 always.

Subchapter 3d: Cache and Memory Management

We're going to talk a bit now about the cache memory management. What is cache? Well cache is the first level of memory that we reach once the address leaves L1. Cache allows the users to take advantage of single cycle memory without having to specifically move instructions or data manually. In other words it takes away that setting up a DMA and moving the data or instructions to the background.

L2 and L3 memory can be used to hold large programs and data sets, also the paths to and from L1 memory are optimized to perform with cache enabled. What happens is once a cache line transfer occurs nothing can interrupt that cache line fill so it does have a high priority over other transactions that occur. Cache automatically optimizes code and data by keeping recently used information in L1. There is an algorithm called LRU, least recently used, what this means if first of all we have several ways to store a cache line in internal memory. Once we have all four ways filled and we need to find space for another cache line, the least recently used algorithm basically says which ever cache line was used the longest time go, that's going to be the first one to give up it's space to the new cache line.

Cache is managed by the memory management unit through a number of CPLBs, these are Cache-ability Protection Look-aside Buffers. The idea of the CPLBs are we divide up the Blackfin's memory space into a number of different regions or pages, and each of these pages can be assigned different types of protection or cache-ability parameters. For instance you can provide user or supervisor mode protection, perhaps certain pages only supervisor code can access it, not user code. Read write access protection or in the case of caching this is how we define which pages of memory are supposed to be cached in L1 and which ones are not.

Chapter 4: Additional Blackfin Core Features
Subchapter 4a: DMA

What we're going to do now is talk about some additional Blackfin core features, starting with DMA or Direct Memory Access. All Blackfin processors have at least one DMA controller, some have several. Direct Memory Access allows the Blackfin to move data between memory and a peripheral or between memory and memory without the core being involved. The core does get involved initially to set up the DMA registers or to set up the descriptors in memory but after that there's no core cycles used to move the data.

The core ends up being interrupted just when the DMA transfer completes. This improves the over all efficiency because now the core is interrupted just when a block of data has been transferred and not dealing with each and every single piece of information. There's also status registers available with the DMA channel so polling of the DMA status can also be done without going through the interrupt controller.

The DMA controller has dedicated busses to connect between the DMA controller itself and the set of peripherals, the DMA capable peripherals in the Blackfin, the external memory L3 or SDRAM, as well as the core memory, so there's dedicated busses for all these different paths. The Blackfin also supports various power management options, one of them for instance is built to maintain low active power. Any peripherals that are not used, you typically have to set a bit to enable them are automatically powered down, there by conserving some power.

Subchapter 4b: Dynamic Power Management

There's also a dynamic power management mechanism that allows you to dynamically change the operating frequency and the core voltage to optimize power for your particular application. For instance there's an onboard PLL that used to multiple the clock in frequency to whatever your desired operating frequency is so we can step that up as much as 64 times. We can also optimize the core voltage for the desired operating frequency. Typically when you're operating at a lower frequency you don't need as high a core voltage.

There's also a number of stand by power modes, five all together. Full on, as the name implies of course doesn't have any real power savings, the PLL is running, your operating typically at full speed. You do have an active mode in this mode your running from the clock infrequency, you're not using the PLL, so this is a lower power operation, both the core and the peripherals are running. You have a sleep mode, in

the sleep mode the PLL is running, the SCLK is typically running at full speed, but the CCLK has stopped. The core has been put into an idle state just sitting there waiting for peripheral DMA transfer to complete for instance.

There's a deep sleep mode where both the CCLK and SCLK have been shut off. In this case there really is no activity, so we're going to wait for something like a real time clock for instance to wake us up. Hibernate is also the deepest sleep mode, in this particular mode in addition to shutting off both clocks, we also shut down the power to the core. Of course what this means is we've also lost whatever information there was in L1 so you wouldn't go into hibernate until after you save off any critical information.

The Blackfin does have a real time clock on most of the variance that has an alarm and wake up features. The real time clock is actually a separate sub system and has it's own clock, has it's own power supply. It's not affected by the any of the software hardware resets. In fact the real time clock is one of the devices that can wake up the Blackfin out of the deepest sleeps including hibernate.

What we're looking at here is the types of power savings that you can get as your application progresses. What we have is the ability to optimize the power consumption in the different phases or our application. We see here for instance that at the start we might be requiring a fair amount of performance, so of course there's no power savings there, but we might be going on to another stage where need to go into a much quieter state, so in this case here we want to go down to a lower frequency. We can adjust the PLL, adjust it down, but also when we're operating at a lower frequency we generally do not need as high a core voltage. What the chart is showing here is that just be reducing the frequency you do get a reduction in power consumed, but if you can also reduce the voltage then the power reduction can be deeper.

Power consumption or at least the dynamic part of power consumption is proportioned with the operating frequency, but it's also proportioned of the square of the core voltage. What we see here are just the different power mode transitions that Blackfin supports. Say for instance coming up out of a reset we go into full on state where the PLL is running, the CCLK is running, SCLK is running and through controlling bits in the PLL control register we can switch basically from one state to another. For instance we can go to the active state by disabling the PLL, switch back to the full on state by re-enabling the PLL with the by-pass bit. From either the active or the full on states we can go into a sleep mode by shutting down the

core clock and we can wake up to one of the other states depending on the state of the by-pass bit. Same thing, go into deep sleep mode by shutting down both clocks.

Of course when you wake up with a real time clock out of a deep sleep you wake up to the active mode where you typically start up the PLL and then transition over to the full on. Also from either of these states, active or full on you can go into hibernate state by setting the onboard switching regulator to zero frequency or shutting it down in other words. This basically allows you to achieve the deepest power savings when you don't require the Blackfin processor for a particular period of time.

There's a number of different things that you can use to wake up from hibernate, real time clock, CAN activity, PHY activity in some of the processors. Now a lot of stuff you don't have to worry about this at a very low level, there is something called System Services. This is a set of functions, libraries that allow you to with a single function call control things like the operating frequency, being able to change from one state to another, a lot of that is basically isolated from you in your application.

Subchapter 4c: Blackfin Hardware Debug Support

Next we're going to talk a bit about the type of hardware debug support that's available. The Blackfin processor has a JTAG port and one of the uses for it is to do in circuit emulation. This allows you to do no intrusive debugging, so your application is running full speed on the target and we're just sort of watching from the outside. Of course once the processor stops we can go in read memory, read registers, and change things. There's also BTC or background telemetry channel support with the Visual DSP tools. What this allows us to do is while the processor is running, if we want to say get some information from the memory buffer and display it in Visual DSP, we don't have to shut the processor down, instead we define what buffers we want to share, we need to link in the BTC library, there's a small bit of code we need to execute, but essentially we can do a data exchange for debug purposes with a running target which is very useful during the initial stages of your code development.

Additional hardware debug support, there's some hardware break points built into the Blackfin, eight registers allows you to set up six instruction addresses and two data addresses so when we say match that address through instruction fetch or data fetch, we can generate an exception and track that. We can use pairs of registers to specify addresses and break when we have something, have a fetch within that range or outside that range.

There's a performance monitor unit, or actually a pair of 32 bit counters which can be assigned to count a number of different performance parameters, like how many cycles you're spending waiting for data fetches, how many interrupts are occurring and so on. There's also an execution trace buffer, 16 locations deep. What it does is it stores, any time there's a change in program flow, so two pieces of information are stored, one is what address were we at when the branch occurred, and what address did we branch to. This could be for interrupts, jumps, sub-routine calls, or the first iteration of a loop.

These last three points by the way are available even without having anything connected to JTAG port so you can build this into your hardware.

Chapter 5: Conclusion

Subchapter 5a: Summary

In summary I would just like to make few points, the Blackfin core has a number of features and instruction that support both control and DSP types of operations, and it's important to know this is all within one core. The high performance memory and bus architecture supports zero wait- state instruction and data fetches from L1 at the core clock rate. These three fetches all occur at the same time. If you have large applications that reside in the larger but slower external memory, they can still benefit from the high performance L1 through the use of caching and or dynamic downloading. Even though your application doesn't totally fit in L1 you still get the benefits with those features. The Blackfin architecture enables both efficient implementation of media related applications as well as the efficient development of those applications.

Subchapter 5b: Resources

To wrap up I'd just like to point you to some additional resources that you might find useful, for additional information on the Blackfin architecture please refer to the Analog Devices website which has links to manuals, data sheets, frequently asked questions, the knowledge base, sample code, development tools and much, much more. www.analog.com/blackfin is where you can go and start with that.

For any specific questions you can also click the ‘ask a question’ button that you’ll see on the screen. Of course we Kaztek Systems provide world wide training on the Blackfin and other Analog Devices architectures as well as system development support on these architectures. For more information visit our website www.kaztek.com or send us an email at info@kaztek.com.

I’d like thank you very much for your time and good luck with your developments.