Presentation Title: Blackfin® Optimizations for Performance and Power Consumption

Presenter: Merril Weiner, Senior DSP Engineer

Chapter 1: Introduction

Subchapter 1a: Agenda

Hello my name is Merril Weiner, I'm a Senior DSP Engineer at Analog Devices. I'd like to talk to you today about optimizing your software design for the Blackfin processor for better performance and lower power consumption.

In this module we'll talk about different ways to optimize your software design for having better performance and/or lower power consumption. We'll be using audio and video player examples throughout this module and we'll show those examples in action as a demo at the module.

It's recommended that viewers have seen the module Basics of Building a Blackfin Application and have a general working knowledge of the Blackfin processor before viewing this module.

In this module we'll start with an overview; what does it mean to optimize your software architecture on the Blackfin processor? Then we'll go into optimizing internal memory use, external memory use, low power modes, other hardware settings and then we'll go into the demo.

Chapter 1b: Overview

What do we mean by an optimized software design? An optimized software designs means increasing your performance, it means lower MIPS on the system, and if you have lower MIPS on the system that means you have the ability to add in additional features or to reduce power consumption. How do you do that? This is exactly what we'll be going into is how do we optimize the system? It's more then just setting the optimization flag on your compiler, there's a lot more that goes into optimizing a full system. We'll talk about how you use internal memory, L1; external memory, L3 or SDRAM; the power modes; as well as the effective use of Blackfin and SDRAM settings.

I have a chart here to help explain a little bit more about what I mean. If you were running a BF531, 400 MHz full tilt you'd be running around 150 mW of power. That's an awful lot for a system but if you were able to optimize your MIPS down to let's say 200 MIPS, now you have 200 more MIPS to play with, so you can add a lot more features onto the DSP. Or if you were interested in optimizing for power you could drop the voltage all the way down to .8 volts and then your power consumption would be under 40 mW. This is what we talk about is if you optimize your system then you have more headroom to do more tasks, that's additional features. Or you can drop the power consumption, a lot of people are interested in that especially on chips like the BF531 that's designed for lower power.

Throughout this module we'll be using audio and video play back. The reason why we do that is because a lot of the lessons that we've learned is through optimizing our audio and video players as well as the

entire rest of the system, audio post processing, video post processing. So we'll be using these scenarios to help flush out the theories as examples.

Chapter 2: Internal Memory
Subchapter 2a: L1 Instruction

Optimizing internal memory use. On the Blackfin processor there's a couple different types of internal memory, the first one is L1 Instruction, and there's always a question of how do you use your L1 Instruction, do you use it as SRAM, instruction cache or maybe both. It depends a little bit upon which Blackfin processor you're using. For instance if you look over here we have the BF531, it has as total of 32KB of L1 Instruction. The first 16KB of it can be used either as cache or as SRAM, and the last 16KB is only used as SRAM. This is more appropriate for an audio only application. Then there's the BF533, this has the same 16KB that can be used as cache for SRAM, but it has an additional 64KBB that's used only as SRAM. The question is; how do you use memory according to your software architecture?

One of the things that we look into is if you have an audio decoder, how much code space do you have? Our audio decoders they all take less than 16KB of memory, so the caching model makes a lot of sense because it's just a lot easier to use, and we'll get into that more later. The video decoder is used quite a bit more than 16KB. Some of them use even up to 80KB of instruction. The question is; what's your best memory model to execute that much instruction code?

We'll go into three different memory models that we use on our decoders, the first is a cache memory model, the second is the overlay, where cache is turned completely off, and the last one is a hybrid where we have cache turned on, and then we have the remaining L1 Instruction used as SRAM.

In the cache model for instance on the BF531, there is 16KB of cache available and then there's 16KB of SRAM, so this type of model is perfect for an audio only application, or if all of your critical models have less than 16KB of code. This also works well with multiple critical modules because the cache will just load in whatever code you need. The ease of programming is great, you don't have to worry about what's in L1, what's in L3, you just leave all of your code executing in L3 except maybe some critical code you can bring into SRAM but you don't have to worry too much about it, you can just let the cache do it's job.

We find that not only is this great for audio applications, but it's great for a lot of other applications as well. One of the advantages to using this model is that it allows you to buy a processor that has less L1 Instruction. When you do that you can save on your BOM and you can also reduce power, both of those are advantageous to the customer.

The cons are that you can have significantly worse performance when your critical modules are all greater or much greater than 16KB of code space. In summary this is a good model especially when you're critical modules have less than 16KB of code space each. Also it's good to keep in mind that this is the only working model available if you're using uClinux.

The second memory model is the Overlay Model, with this we turn the cache completely off and we use all of the L1 Instruction as SRAM. For this we set aside a little bit of space for the operating system, ISR's, any type of system services you need, and then we use the rest of our L1 Instruction. Now the pros of using this is that you end up getting increased performance for all of the code that you can place in L1 Instruction SRAM because you don't have to wait for any type of cache misses where a cache line has to be loaded if you're using the cache model.

The cons on this are you have decreased performance for code that you can't squeeze into L1 Instruction, and then you have to leave it in L3 and you don't have the benefit of having the cache system. The issues here are how do you set this up? The first thing is if all your critical code can fit into L1 into a single overlay that's great. Then you go ahead and do it. If not then you may need to have multiple code overlays to handle each of the multiple critical modules. We do that specifically with our video decoders because each one is quite large, anywhere from 64 to 80KB, going through our set of Mpeg 2, Mpeg 4, H.264, VC1, we have a whole bunch of video decoders and if you want them all running on the same system you have to be able to swap each one in and out of internal memory.

Now the disadvantage of this is non-critical modules may not be worth the content switch, so you have to keep in mind; what is a critical module and what isn't, because you don't want to have to go through time of swapping something in if it's not going to be running for long. The other issue with this is that if you're handling multiple code overlays then you need to be able to have a swapping mechanism with a little extra code development, but otherwise it works out pretty well. For helping this set up on your overlays, whether it's single or multiple overlays we have a PGL linker tool that's available with VDSP.

In summary this method is optimal if all of your critical modules can fit into a single or multiple code overlay, and you have little or no none critical modules executing out of L3.

The next module that we have is a Hybrid Model. With a Hybrid Model we use what we call the best of both worlds. We turn on the cache and then we use the rest of the L1 Instruction for overlays. The pros on this is we get the same excellent performance for all the critical models that can be placed into L1 Instruction. We can't fit as much in there, so some is left out in L3, but then we have the cache to pick up the slack on those. Then we still have very good performance for code that's left out in L3. Another thing is that the non-critical modules will then run much faster.

The cons on this is that this method requires having a processor with more internal memory. If you're looking at a situation where you're needing these code overlays in the first place you're probably at that point anyway. The issues are exactly the same as the overlay module, the only difference is that you have 16KB less of L1 Instruction for your single or multiple code overlays.

In summary this gives the optimal performance for multiple critical modules where the size is greater than 16KB for each of these critical modules and where you have multiple non-critical modules that need to executive out of L3 external SDRAM.

Just to flush this out a little more I thought we'd go through a situation with our video decoder example. We turn the cache on, 16KB of I-Cache and then we set aside 8KB for the operating system, ISR's, anything else that we need. We found we really only need about 4KB of that and we like to have a little buffer. The we have 56KB that's left over for our multiple code overlays. We've done a lot of profiling on this because obviously we want our decoders as optimally as possible. The whole thing of more features or better on power consumption because if it runs faster than you can run a larger resolution, higher bit rate, it doesn't matter there's always more features that a customer wants. On the flip side is if we're saving on MIPS then we can reduce the power, especially reducing the voltage because that's where we get our really big power savings.

When we use the cache-only model for video, we have about a 20% performance degradation due to the fact that we have very large critical modules so there's a lot of cache misses. We thought our Overlay

Model would be the best because we're able to put all our critical code directly into L1 but we found that there's a 10% performance degradation system wide because non-critical modules were executing out of L3 and we didn't realize how much of an impact that had. When we moved to the Hybrid model we found that gave us the best performance, that was as we call it, the best of both worlds.

To summarize on this, this is specifically for video. If you're using audio or something that has smaller code spaces or fewer critical modules this may not be the best model for you. This is the best for having the large critical modules.

Subchapter 2b: L1 Data

We've talked about L1 Instruction, now let's talk about L1 Data, it's a very similar situation in that if you have a certain amount of data space in L1, you can use it as either data cache or as SRAM or both. Here just to review on the BF531 we have 16KB of L1 Data and that can either be used as SRAM or as cache. On the BF533 we have two different banks, L1A and L1B, each of those has 16KB that can be used as SRAM or cache, and an additional 16KB that's available only as SRAM.

Before going into this analysis I just want to flush out a couple of concepts we'll be using, and that is what are the different types of data? I want to get into this because when it comes to code, code is very straight forward, it's read-only. When you're using it you only have to swap the code in you execute, if you want to execute some other code using overlay module you just have to swap it in. With data it's a little more confusing because you have read/write data, you have temporary variables and you have your read-only data. The read-only data we use mostly for static variables, tables and stuff or sets of coefficients for your algorithms. Those do not change so we consider those read-only. Then you have read/write, that's your state variables, things that change between each time you run your module. Then you have the temporary variables, the ones that use only during that one run of the module.

Once again we found our audio decoders use less than 16KB of data, so it's a much simpler model, but our video decoders and our video post processing use much more than 32KB of data so we have to be very careful as to exactly how we set that up. Once again we have the cache-only model where we turn on D-Cache, that's the situation that we'd use for audio on the BF531. We have the Overlay Model where

we turn of the D-Cache and use all of L1 Data as SRAM and then once again we have the Hybrid Model, where we turn on the D-Cache and we use the remaining data as SRAM.

The Cache-Only Model, this is very straight forward. We put all of our data, whether it's read, read/write, or temp in L3 and we let the D-Cache do it's job. The pros for this are ease of programming, this also works well with multiple critical modules when the code size isn't that big, or when you have very tight nested loops because then you won't have that many cache misses.

We find that this is perfect for audio only applications and it also helps when you're buying a processor to be able to buy a processor that has less L1 Data because you can save on the BOM or power consumption. The reason for that is the more internal memory you have the more power the processor is consuming.

The cons on this are that you may have significantly worse performance when each critical module uses more than the D-Cache size that's available. This is good for most cases especially when each critical module uses less then the data cache size, or even a little bit more, you won't see any impact, and also it's the only model that's available from UClinux.

Once again we have the Overlay Model, with this we turn of the D-Cache and then we set aside a space for OS and systems services etcetera, and then we place our read/write and static variables in L1 and that needs to be swapped in and out. We allocate our temporary variables directly out of L1. One of the other things to keep in mind is that when setting up where all of your variables are, it's important to make sure that variables or buffers that are accessed concurrently are not within the same 4KB block because you might introduce some stalls in your algorithms if so.

The pros for this are increased performance for modules that use the L1 Data overlays. The cons are you have to increase performance for modules where data must be accessed directly out of L3. Once again these are in non-critical modules and it depends on how much data they have and how often it's accessed.

The issues are very similar as with L1 Instruction overlay module, if you can fit all of your data into L1 Data SRAM then you can just use a single overlay, you don't have to worry about multiple overlays and place all the data directly into L1 Data SRAM. If you have multiple critical modules that have a total data size of

greater than 64KBB then you'll need to set up the multiple code overlays and swap them in and out. You need to be careful of what's a swap in and out because read-only you only need to swap in, read/write you need to swap in and out and the temporary you only need to allocate directly, you don't need to swap in and out. We've noticed that programmers tend to get a little bit lazy and through everything into the read/write, so you need to be careful with that or else you'll be increasing the amount of context switch time in your system.

Once again we have the PGL linker tool to help with setting up your sections and overlays. In summary this is optimal if the data for your critical modules will fit into your multiple overlays and if you have non-critical modules that access very little or no L3 data.

Once again we have the Hybrid model. With the Hybrid model we have the D-Cache turned on and you have the option because you can turn on L1A D-Cache by itself or you can turn on both L1A and L1B. We choose to do this, we found that we have better performance with our video decoder and also when running other applications simultaneously we get better performance on those other applications because those will not use L1 Data, they'll only use the D-Cache, so they get much better performance that way. We set a little aside for the operating system, etcetera, and then we have a total of 8KB or so in L1A and another 16KB in L1B for temporary data or code overlays.

We set up the L1 D-Cache in write back mode, and we use the remaining L1 Data for SRAM. We place all of the read/write state variables, those tend not to be accessed too much when they are it's over and over again so we find there's very good performance, placing those behind the D-Cache. We let those sit there. For the critical tables we find that sometimes they end up getting over written in the cache, so we like to swap them directly into L1 and not have them be cached. Then the temporary variables can be allocated straight out of L1 Data SRAM. Once again it's important to keep in mind not to keep concurrently accessed buffers or variables in the same 4KB data blocks in L1 Data.

The pros for this are you get excellent performance for your critical modules, and very good performance for your non-critical modules that use D-Cache only. The cons for this is once again it requires having a processor with more than 32KB of L1 Data. The issues are the same as the Overlay Model. In sum this provides optimum performance for when you have multiple critical modules that use more than 32KB of data, and where you also have multiple non-critical modules running on your system.

Just to flush this out we'll go through the audio and video examples. For audio it's very simple, we don't have that much data so it's just easiest to use the cache model, replace all of the data directly in L3 whether it's read, read/write, or temp and we just let the D-Cache do it's job. Very simple. Video is a little more complex. It requires using a Blackfin processor with at least 6C 4KB of L1 Data. Some of the processors have L2 as well, so there's a lot that you can do with that. We've chosen to use the Hybrid model, it's really the best of both worlds. We've done a lot of research into this, once again performance is really critical for our video decoders.

We turn on the D-Cache in both L1A and L1B, we set aside 8KB for the operating system services etcetera, drivers, and then we have the remaining 8KB in L1A and 16KB in L1B for a total of 24KB available for data overlays and or temporary variables. To simplify our system we've also been moving towards using this only for temporary variables. If a module feels that it needs to have a static variable loaded in, it will swap it in directly and this reduces the amount of overhead and management of the overlays by the system, it only has to worry about temporary variables and just allocating them, it's much simpler.

When we went through we found that if we use the cache-only model we end up having a 100% system degradation and that's just because there's a tremendous amount of cache misses when dealing with large amounts of data. The Overlay Model, we found that the critical modules themselves had 7% better performance then the Hybrid model. We found that the non-critical modules were far worse and it caused a net performance degradation of 20% system wide. When we went to the Hybrid model we found that was the best performance. It was 20% better then the overlay model, and significantly better then the cache model.

Subchapter 2c: Cache and DMAs

Before I go on while we're talking about L1 Data and L1 Instruction there were a couple of other things to talk about relating to cache. I wanted to get into that a little bit. One of the things that you need to be aware of when dealing with the cache as well as having SRAM at the same time is data coherency. What I mean by this is when the data that's in cache does not reflect the data that's in L3 SDRAM. How does that happen? That happens because peripheral DMAs occur and the cache doesn't know that the

variables are changing. Or that cache is changing and the peripheral DMAs don't know. We'll go through the two different situations.

The first one is the outgoing data. The thing to remember is that a cache is not automatically flushed so if you start off a peripheral DMA and some data has changed in the cache then the peripheral DMA is going to send out the wrong data. We had a situation of that with a customer once, it took me about five minutes to figure out what was going wrong, but it took the customer about two weeks to ask the question so it's always important to make sure that you're aware of this type of situation. There are two ways around it, one is to always make sure that you flush the data before you do a peripheral DMA on that data buffer, or what we find is easiest is the buffers that peripheral DMAs access, we mark as being non-cached in the CPLB tables.

It's a similar situation with incoming data, because you may have a buffer that's cached, and then the peripheral DMA may update that data causing a data coherency problem. Once again it's a situation that when you update that data you should either invalidate the cache so it reloads it, or what we prefer to do is once again is any buffers that peripheral DMAs access we mark as non-cache.

Subchapter 2d: Memory Pipelining

If they're marked as non-cached then the situation is how do you get that data into L1? We consider these temporary buffers because they're used only once, generally with the system processing. You have data you just want to send out, or you have incoming data to process. It's only accessed once, the cache isn't going to help you out anyway because you're only accessing the data once. We use memory DMAs to bring it in, and when you bring in this temporary data from external memory, whether it came in off of a peripheral DMA or from another application, it doesn't really matter what the source is, we do the memory DMA and we execute code in the mean time. We call this memory pipelining within our group. When you do this you can save a lot of processing power. Obviously you want to be able to run your memory DMAs asynchronously or else you're just blocked and wasting time. What we found is that where we have memory band width limited modules the amount of time spent on bringing in the memory and sending it back out, or bringing in the data and sending it back out could be just as much time as the algorithm execution. In that situation if they're exactly the same and you're blocking, now you're taking twice as long as necessary. By using this you reduce your MIPS by up to 50%.

Just to flush this out a little bit of how that would work in our video example, with our video post processor it's very memory bandwidth limited so while working on a certain line, processing a certain line of video on the horizontal we'll be brining in the next line so that when we're done processing this we can just go forward and we'll be sending out the last line that we process. The memory DMA is working continuously while we're executing continuously. By using this technique we've reduced our MIPS by 40% in the video post processor.

Subchapter 2e: L1 Data Scratchpad

There's one more part of internal memory that we haven't talked about and that's the L1 Data Scratch Pad. The L1 Data Scratch Pad is 4KB on all of our Blackfin processors. There are a couple of limitations, one it's can't be used for D-Cache and, two, you can't use it for memory DMAs. That limits what you can use it for by quite a bit. Throughout the company generally we use it for the stack. When you have just one stack that's great you just use it for the one stack, and that's great if you're using a single task solution or you're not using an operating system. If you're using an RTOS now you have a situation where every single task on your system has it's own stack and not all of them are going to fit. We found you can fit two, maybe three, at least for us, of our application stacks into this Data Scratch Pad. Generally you want to place the most critical task stacks, or the largest most frequently accessed stacks into the Scratch Pad, and then you place the remaining stacks in L3 and let the D-Cache do it's job to bring them in. Because the stacks are accessed frequently in the same area, we find there aren't that many cache misses and so we get pretty good performance that way.

If you place them L3 and you don't have them cached we've seen really horrible execution numbers. We've seen our audio decoders go from 20 MIPS up to nearly 200 MIPS, a ten-fold increase. It's very important to either place it in the L1 Data Scratch Pad or have it cached. The rule of thumb here is to experiment because you may have an idea of which stacks you think are best to put there, but you really need to experiment to find out and you should also figure out how big each stack is so you can see how many you can fit in, and just play around with it. We thought we had the best scenario and then just for the fun of it we started playing around and we saved another 20 MIPS with our video solution. 20 MIPS was a 10% of our total MIPS budget, which is quite a bit, so we're quite pleased with that. You'll hear me talk about this throughout the rest of the module, is sometimes it's important to experiment because although

I'll go through some of these theories and you'll try some of the theories that I've mentioned and some of your own, sometimes the exact situation that you're dealing with is counter-intuitive and so you just need to experiment with some of the different variables, some of the different factors to get the optimum performance.

Chapter 3: External Memory
Subchapter 3a: Memory Settings

We've talked about optimizing the internal memory use quite a bit, now we'll go on and talk about the external memory use. Before I get into this too much there's you need to be aware of with the external memory and that there is a priority setting on most Blackfin processors of which has higher priority access to external memory. Is it core access or is it peripheral DMAs? This can have a really big impact on your system because if the core has higher access then the peripheral DMAs then you may have FIFO under-runs or over-runs on those DMA channels so you really need to be aware of that. As I said most Blackfin processors you can switch the priority on that. Some of the Blackfin processors have even a more fine tuned priority scheme where you can have some peripheral DMAs having higher priority and some having lower priority than core access. It's important to just with the hardware reference manual for your Blackfin processor and figure out what the best scenario is for your application.

Subchapter 3b: External Memory Banks

Going into the external memory banks just want to make sure we're clear on the terminology first, banks are used in a lot of different ways so when I say banks through this next section I'm going to be talking about the four sub-banks that are in each of the external SDRAM banks for the Blackfin. Some Blackfin processors can have more than one external SDRAM, each one of those is called a bank and then within each of these external SDRAMs there are four sub-banks. The reason why this is important is that in each of these four sub-banks only one page is open at a time. If you want to access a different page in that sub-bank, it has to first close the current page, and then open the next one. You'll get hit with the latency of multiple SCLK cycles so you don't want to do that too often.

We'll be going through some of how we try to place things in order to get better on that. The main theme on that is to place concurrently accessed buffers so if you're processing data you want the input buffer in

one sub-bank and the output buffer in another one, because that way you're not flopping pages. You only have a page miss when you extend past the end of a page rather than toggling back and forth between two buffers. The other thing that we'll talk about is for Mobile SDRAM when you have sub-banks that are not being used currently, you can put them in self-refresh mode instead of being active to save on power.

Just going through our examples to flush it out a little bit, for our audio example we don't find that there's that much of a performance decrease by placing all of our buffers into a single bank. A lot of that is because it's low bit rate on the input and on the output next to video. If you have a 10% increase in MIPS, or a 20% increase of MIPS, this is only a couple of MIPS, it doesn't effect the system that much. We put it all into one bank and then we place the other banks into self-refresh mode if we have Mobile SDRAM to save on power.

For a video it's a little bit different because we have two input reference frames, so we want to put those in separate banks. Then we have an output frame, and we may have more than one output frame just because we're ping-ponging our buffers. Then we also have our audio buffers, so they just need to be placed into a bank that isn't going to be accessed at the same time as the others. You only have four banks so you do the best you can, there's always going to be some overlap, but the video buffers we find to be the most critical parts.

We've also found that whenever we miss some where because we've been shuffling around memory and we haven't set up our banks correctly or one buffer crosses things we end up having performance degradation of up to 100%, purely due to page misses. So it's very important to set up your buffers correctly. Just as another gotcha on this, is this usually happens when you change your SDRAM memory part size, you have to go back and change where you're putting all of your buffers because that's surely to hit you because otherwise you're going to have all your buffers in two banks instead of four, and then you'll get that 100% performance degradation.

Subchapter 3c: System Memory

Another thing that we found when going through with setting up memory is we ran into a situation where the operating system and all the system memory, everything that was going on, all the internal buffers, variables, etcetera were consuming more than half of the SDRAM size. Some of RTOSs and uClinux

require that all of the system memory be contiguous, so if that uses over half of the memory, then you're left with only two banks in SDRAM to use for your data buffers. What we did is we shifted the system memory to start later, there's different configurations on this, you can start it later in bank zero, or early in bank one, but we found that this gives us the access to three different banks for the data buffers, which gives us a huge performance increase.

Chapter 4: Power Modes
Subchapter 4a: Hibernate and Sleep Modes

We've gone through some of how to set up the external memory, now we'll go into power modes and we'll get a little bit more directly in how to save on power consumption.

There are three different power saving modes on the Blackfin processors, the first one is hibernate mode. Hibernate mode gives you the best power savings and is ideal for non real time systems, or when you're a non real time part of your application, and should be used when you have long idle periods and when you don't need your peripherals. Hibernation turns off the core clock, and it shuts down the core power, so you get a savings both on the core power, that's your static power, and then the core clock drives the dynamic power. You're getting a huge power savings there. It drops the power consumption down to about 50 uA. The wake up time is pretty fast, it takes about 600 us to wake up and that's the time for an external power regulator to power up as well as the software to restore its states.

Sleep mode. Sleep mode we find to be the best power savings for real time systems, you can keep the peripheral DMAs going. If you have audio or video play back it can continue while you're sleeping. This mode should be used within the RTOS or system idle routines for shorter idle periods or when you're peripherals are used continuously. For this we keep the core power on so you still have your static power, but we turn off the core clock so it's completely off and we save on the dynamic power for the period that you're sleeping. With this the peripherals continue to run and all of internal memory and registers are maintained. The wake up time is very fast, it only takes 10 core clock cycles to wake up from sleep mode, that's about 40 ns if you're running at 250 MHz.

There's the Deep Sleep mode, this one has some of the advantages of each, it's lower power than the Sleep mode, but like the Sleep mode you have your static power, but the difference between Deep Sleep

and Sleep mode is that the peripherals do not run so you save power on shutting down the peripherals but then it's not quite as real time. You save a little bit of power there, it also has a very fast wake up time. We found that the Hibernate and Sleep mode tend to cover most situations. Because if you don't need your peripherals you might as well hibernate because it really doesn't take that long to come out of hibernation. With our applications we found that Hibernate and Sleep mode have been best.

Subchapter 4b: Frequency and Voltage

The next thing we're going to talk about is the direct control over static and dynamic power, before we get into that it's very important to have an understanding that with the Blackfin processors there is a whole bunch of different voltage tiers for each Blackfin processor and there's a maximum speed that you can run the processor at for each voltage. I have a sample posted here but you really need to refer to your processor data sheet because it different for every single Blackfin processor.

The static current on a Blackfin is directly related to the voltage and temperature. If you want to reduce the static power the best thing to do is to lower the voltage. What we want to do is lower the voltage to the minimum amount necessary to run your application. Now the dynamic current on a Blackfin processor is directly related to the core clock cycles. There's two ways of handling that, one you can either set your frequency to be exactly at the point that you need it, or what I prefer is to set the frequency to be the maximum allowed at that voltage level, and then you just sleep because that saves you the core clock cycles. You just sleep when there's nothing to do. The advantage of that is you can handle the peaks and troughs in processing power requirements, where as if you have it at a steady state, you can get stuck and have bad performance for a while.

Just to flush this out a little bit, in our audio example if you're using a BF531, 400 MHz and let's say the audio decode plus post processing and everything else on the system is running about 54 MIPS, which varies upon the system, and we have a lot of different scenarios, that's one of them that I pulled out. If you were to run the system full power at 400 MIPS, 1.14 V, it would cost over a 150 mW, and that's not low power for audio by any means. If you would use the sleep mode when there was nothing to do, but keeping at the same voltage level, you would get down to around 50 mW of power, much better, still not great. If you were just to reduce the voltage without using sleep mode and go down to .8 volts and run full tilt there, your power consumption will be 45 mW, a little better, still not great. If you combine both

methods, to reduce both the static and the dynamic power, you'll get down to .8 V, 54 MIPS, and that'll cost you only about 17 mW, and that's much more competitive in the market today. Sometimes we've gone through this, each chip is a little better, 17 mW is what the spec says at 25C, so you do at least that well.

I have two different video examples, the first one is QVGA example and this is similar to what we'll be showing later on in the demo where we have QVGA decode, and it has video decode, video post processing, audio decode, audio post processing and also alpha graphics blending and we can do this in about a 183 MIPS. What we do is we can set the voltage to .8 volts, run the CCLK at 243 MHz, and the reason for that number is it's as multiple of 27 MHz which is the master clock that we have available on EZ-Kit and we sleep 25% of the time in this model. In this situation once again if we were to take a BF533 600 MHz part, the reason for choosing this processor is it has a lot of internal memory, but there are other processors that are available from the Blackfin 52x series that also have a lot and are more geared towards low power. This was the example that we used, this is one of our development platforms.

If you were to run at full tilt it would cost over 320 mW and that's nearly as much as an LCD will cost you so that's not very good. If you only use the sleep it gets you down to 180. If you were to only reduce the voltage and not use the sleep you get down to 70 mW, much better. If you use both tactics you get down to around 58 mW, so that's great performance.

Just to show something a little different then the .8 V scenario, I threw in a WQVGA, that's Wide QVGA, it's another standard for a smaller LCD that has the aspect ratio better for wide screen. That's 480x270, so in a similar scenario we find that our MIPS are running around 303 MIPS. For this 303 MIPS will not fit at .8 volts, the maximum speed there is 250. At .85 V however we can go up to 375 MHz so it can fit in there. We set the voltage to point .85 V, the CCLK, the best multiple of 27 MHz is 351. We can get a little bit higher on it by running the VCO higher and then taking odd multiples of the master clock and then doing half, it gets a little complicated so I'm just doing even multiples here. There's always room for a little more optimization on your system is what I found. In this scenario it'll sleep 14% of the time, that's the reason why I brought up kicking up the CCLK a little bit because sleeping at only 14% of the time may not be enough to handle the peaks in your system, it may be worth getting an additional half a master clock is an additional 13 ½ MHz, that could really help out with this system. Otherwise if we found that wasn't

working quite well enough we'd bump up to .95 V. The voltage is what really gives us the biggest increase in power consumption, so always try to run at the lowest possible voltage.

Once again we have the huge savings that starting at 320 mW running full power we can get down to 88 mW, Being able to do WQVGA in this range is pretty good.

Subchapter 4c: Memory Speed

We've talked about so far reducing the system power by reducing the Blackfin power. However that's a very microscopic view of the system, and when we you look at any type of portable device that you want to have low power for, there are three major consumers of the power. The first one is the LCD, that's the biggest, the second and third depending on the scenario is the processor and the external memory. We have no control over the LCD and we just talked about the Blackfin processor, so now we need to talk about what can be done to improve the power consumption by the memory part.

What we found is the first question is how do we do that? One of the things that you can do is by reducing the external memory speed because there's a direct correlation between the memory speed and the power consumption. Similar to the Blackfin processor we have the static and dynamic power, reduce the CCLK you reduce the power so it's the same thing. How much can we reduce it? The first thing to know is what is our data throughput, how much do we need to access the memory? What we do is we calculate out and what we've found is that when our data through put exceed half of the total memory balance that's available, we start getting memory degradation so we use that as a general guideline. We figure out what our data through put is, figure out how many memory accesses per second we have, double it, and then that's a good guideline for memory speed.

When doing this if you really want to reduce memory speed a lot, you may have to be changing around your VCO, and your CCLK to make adjustments for that. If you reduce your VCO to go lower, that's going to adjust your CCLK. You have to make sure that you don't reduce your CCLK too much or else you may not have enough horse power to run your applications.

Just to flush this out, we have our audio example, and when we figure out about how much our data through put is, it's about .4 million accesses per second, so I would say we have a minimum of external

memory speed of .8 MHz. Now in a scenario that we're going through for our audio example, we set the CCLK to 243 and the VCO to 243 and we set the PLL deviser for the system clock to the maximum and that gave us an SCLK of 8.1 MHz, that's a full order of magnitude higher then what we need. What can we do about that? One of the things we can do is lower the VCO, well if we lower the VCO we're lowering the CCLK, so one of the things we do is lower the VCO to 54 MHz, and also lower the CCLK to 54 MHz which should be enough for an audio-only application. Then set the SCLK to about 2.7 MHz, I can lower it a bit more but I always like to have extra head room just in case. Also there's not going to be that much difference in power consumption between 2.7 and 1.something MHz, so at that point I think it's pretty well optimized.

Now the other thing to keep in mind is if you reduce the memory speed too much then all the sudden certain parts of your application are going to be running slower because it's waiting on memory to be transferred. Slowing down the memory can cause the MIPS to increase, and if the MIPS increase then your processor may not be running fast enough to handle those MIPS, also there could be a huge increase in power consumption on the Blackfin processor so you have to balance these things out. One of the other things when doing this is thinking back about the memory pipeline we were talking about before, if you're transferring data into internal memory, then if that's taking longer then you don't want to be sitting there waiting for the memory transfer to complete, so the best thing to do is go into sleep mode and that way you're saving on your dynamic power. You just use a DMA frame completion interrupt to wake you back up to execute the code. However, what we found is that in some scenarios the over head of handling all these interrupts can accumulate especially with certain RTOSs and even more with uClinux. We find that handling interrupts isn't necessarily the best way, so we prefer to poll for the DMA to complete because that way the polling is happening before the DMA is complete, where as the interrupt handling occurs afterwards, so we can help reduce MIPS that way. We found that with our video post processing the difference is approximately 5 MIPS. Now that may not be a lot and in a lot of different scenarios maybe it's not that much, probably it's not that much and is insignificant, but if you have tons of interrupts, tons of DMA and the interrupt overhead could be too great, so now if you're polling now all of the sudden all of this polling is consuming core clock cycles, and that's driving up the dynamic power. In order to handle this better, we slow down the core clock so that reduces the amount that we need to poll. This is one of the situations I say experiment because counter intuitive to slow down your core clock to have better MIPS I think that's just not of those things that pops out into your head.

Now we're going to use a video example to help flush this out. Now we're going to just go through QVGA decode, we went through calculated out all the numbers, I have some of it here, it's about 17 million accesses per second, so that tells us we have about 35 MHz minimum for the SCLK. If the VCO and CCLK are set at 243 then we'd set the PLL divisor to 7 to get the SCLK to be around 35 MHz. However when we look at that, normally when we run video we have the SCLK running at over a 100 MHz, so if this is the situation then our DMAs will take three times as long. The MIPS will increase by 200% on memory bandwidth limited modules such as our video decoder and video post processor, or nearly that amount. That's not a good number, so we can't use a DIV of 7, that doesn't work out so well. One of the other things just to keep in mind just like we were talking about before is we have a lot of interrupts and just to completely optimize the system we prefer to poll rather than to use the DMA frame completion interrupts. It gives us a little extra savings on the MIPS.

A PLL divisor of 7 doesn't work, so what is the optimum. What we see is setting the PLL divisor to 4 means that the algorithms would never have to wait because the memory is running fast enough that we never have to wait when we're done with the algorithm on the memory. There is no polling and it's perfect. That's best for the power consumption of the Blackfin processor. However setting the PLL divisor to 7 is best for the memory. What's the best solution? Well we found in this situation that setting it to a PLL divisor of 5 is the best compromise and that in fact is the default value so we didn't have to do anything but it's always good to go through the exercise because you never know what the result will be. This is what I'm talking about saying experiment. It's always good to go through the exercise and even if you don't go through a theoretical exercise like this, it's always good to at least go through and play around with the numbers of it and see if you can get better power savings.

Subchapter 4d: Mobile SDRAM

We've talked about one way of saving power, and that is by reducing the memory speed, the other one is by turning some of the banks, putting some of the banks into self-refresh mode instead of being active. One of the reasons to do this is just that there's a power savings, how much? We've seen that in one example we had a 1.8V Mobile SDRAM and when it's in self-refresh mode it consumes about 1 mW of power, where as when it's in active mode being used it uses about 20 mW of power. That's quite a bit, but when should we use that? I asked that question, and I thought well if one back is about 20 mW of power, then how many MIPS is that worth, trying to balance out the memory versus the Blackfin processor. You

have to keep in mind that if we reduce the number of banks of external memory we have fewer banks to put our buffers in, and that can have a system-wide degradation. 20 mW which is one bank is approximately 85 MIPS, on the .8 to .85 V, and that's quite a bit. The general idea is if reducing one bank causes a performance degradation less than 85 MIPS then it's worthwhile.

The one caveat with this is that the Blackfin processor allows you to keep one, two or four banks active. Three is not an option so if you wanted to have three active banks that's one bank in self-refresh mode, that scenario is not supported. You can only put two or three banks into self-refresh mode, but not one.

Once again we'll use our examples to help us understand the situation a little bit better. With our audio example and the audio decode and audio post processing everything can fit comfortably in one bank without any noticeable performance degradation, maybe a couple MIPS. We keep just one bank active and place three banks into self-refresh mode and we end up with a net savings of about 60 mW.

For our video example it became a little more complex but we did do a lot of testing on this and we found that it was 183 MIPS using four active banks, and when we tested things out we tried using just three banks and after optimizing where everything was placed, the best we could do was a 22 MIPS degradation over four banks and that's much less than 85 MIPS so that would be worth it. Unfortunately turning one bank into self-refresh mode is not an option so we had to look further. We thought well what about using only two banks, and when we tried that the best we could do was a 212 MIPS degradation, that's over 100%. 212 MIPS, not only is it more than 170 MIPS but that raises the total number of MIPS on the system to 395 MIPS. If you're trying to get away with running at a lower voltage, or lower speed processor it can blow you out of the water. This really didn't seem like a viable option either. For video we had to keep all four banks of memory activity, even for QVGA.

Chapter 5: Conclusion

Chapter 5a: Summary

In conclusion there are many things that you can do to help fully optimize your software architecture on the Blackfin processor. We've touched on a number of things, we've touched how to set your frequency and voltage settings, power modes, balancing memory, power consumption against Blackfin power consumption, how to use internal memory, how to use external memory, there's just so many things that

with the Blackfin processor it's such a flexible processor in it's architecture, that can help you take your application to the next level to increase performance and/or reduce power consumption.

If you have any additional questions please just go to Analog.com/Blackfin or click the 'ask a question' button.

Subchapter 5b: Demo

Now I'd like to show you a demo of the video and audio examples that we've been using in this module in real life.

Hi. I'm back here now, I have set up one of our development environments hooked up to my laptop so it's nice and portable and here we're going to stream an MPEG4 movie with audio across through, we have a 537 EZ-Kit here, it's really just a pass through to handle the Ethernet traffic, communicates directly over these wires to the BF533, and we have an LCD board hooked up to the U-Connector.

Here we have an MPEG4 movie with AAC-LC audio, the MPEG4 is streaming at 500 kbps, the audio at 128 kbps, and the container is an MPG2TS. It's streaming across over Ethernet cable to my laptop so I don't need any type of connection to anybody else for the development environment. It goes through the 537 EZ-Kit just because it has an Ethernet adaptor there, and we have some wires here. This is really just for the serial port (SPORT) interface to the Blackfin to transfer the data. The video comes straight out over the PPI directly to the LCD. You don't need any type of special hardware or anything, it just goes straight out to the LCD. The audio we have in audio codec, and the audio goes straight out to the speakers there.

This wraps up our demo, thank you very much for your time today to learn about Blackfin optimizations for performance and low power.