Presentation Title: Performance Tuning on the Blackfin® Processor
Presenter Name: Rick Gentile

Chapter 1: Introduction

Subchapter 1a: Overview

Hello. My name is Rick Gentile. I'm an applications engineer in the Blackfin Processor Group at Analog Devices. I'll be talking to you in this session about performance tuning on the Blackfin processor family. Before we get started, I'd like to tell you a little bit more about this module. The

purpose is to go over techniques you can use to tune system performance of the Blackfin processor.

Ideally, folks who view this session will have some basic understanding of the Blackfin architecture, a basic knowledge of software terminology, and some experience in embedded system development. With that, let's get started.

The module outline includes a brief introduction about the purpose of this session and why we feel it's an important session for developers of embedded systems. I'll talk a little bit about building frameworks and go over some of the common attributes of frameworks that we see our customers using and hopefully save you some effort as you get started in Blackfin processing. I'll go over memory considerations, how to best use the memory that's available in the Blackfin family, including both internal memory and external memory. I'll talk briefly about benchmarks and try to give you a feel for where you get the fastest performance and how to best use the system from that standpoint.

I'd like to go over managing shared resources, including some of the internal busses that we have that connect up to various portions of the processor. It's important to at least have a high level understanding of how those interoperate and how those work.

Interrupt management is a fundamental part of embedded processing and I'll talk about some of the most common things that are important to understand to ensure you get the best performance in your system. I will conclude with an example that tries to wrap some of the things that we talk about together in the end.

So with that, let's go ahead and get started.

Subchapter 1b: Some background

By now you probably have viewed the session on optimizing with the compiler, and as you know, the compiler really provides the first line of defense for optimization for C and C++ developers. The purpose of this session is really to expand that level of optimization at the system level to include three basic things: memory management, management of the DMA, and management of the interrupts in your system.

We will dig into some of the items that, with a high level understanding, let you improve performance in your system. One of the challenges that you can see in any video-based system, (it's actually very similar in non-video applications) is that you've got a lot of data that needs to be moved around your system, and because of the sizes of the buffers that you're actually working,

and the data rates that you have, you really end up using a combination of all memory resources in the processor, for example, internal and external memory.

Typically we see a lot of customers start on a PC or a workstation and prototype there. They will then actually move to the embedded processor. Similar questions and similar challenges come up. So the purpose is really to set the context for this presentation in the sense that the more powerful embedded processors become, the more customers look to move applications that were previously limited to wire-based applications on desktop processing systems to deeply embedded systems. And so that sort of sets the context a little bit for what we'll go through today.

This chart is actually common to a large number of applications. When you look at where we're actually spending time in the development process, from a compiler optimization standpoint and then, again, at the system level, you can see that typically we'll start off with some type of C code that we have running on a PC that we want to port to Blackfin.

In the case of Blackfin, the compiler has been built from the ground up, really from the beginning, the inception of Blackfin to handle these kinds of applications. And so, as a result, what you'll see is a great out-of-the-box experience with a compiler. And the purpose of this presentation is really to look at the rest of the story. Looking at using memory efficiently, looking at using the busses on the processor efficiently, and looking to move data and instructions around efficiently.

Chapter 2: Frameworks

Subchapter 2a: Common frameworks

So with that introduction, I'd like to now talk about the idea or concept of a framework. The concept of a framework is very important because what we see looking across lots of different applications from our customers, is that the frameworks the customers actually develop with fit neatly into several different buckets. And I'd like to describe those categories and give some of the common attributes that we see to hopefully get you thinking about this early on in your development.

I'd like to just start with a quick definition of what a framework is because there's many different definitions. For the purpose of this discussion, what I'm really talking about is a software infrastructure that you use to move code and data around your embedded system. And hopefully what you'll come away with is the importance of actually doing that upfront in the project and that it will pay big dividends later in the project.

As I mentioned, I'd like to go over three categories of frameworks that we see consistently from Blackfin developers. The three that I like to describe include "processing-on-the-fly", a second one where programming ease overrides performance, and the third one which is where performance supercedes all else.

The first case, processing on the fly, really consists of two sub-cases. Either you can't afford to wait to buffer up data. I use the video example again in this case, where I may not want to wait 33 milliseconds to buffer up a frame of NTSC data. I may want to actually operate on a line-by-line basis. Another case where this model comes into play is if I don't actually have external memory and I want to do everything "on-chip". Here, I don't have any resources actually in my system to buffer up data.

So what we'll look at in this option is really just where you're actually taking the data in, you're operating on it, and then you're making a decision and throwing the data away. I like to use an example of a lane departure system from an automotive application where I may not want to wait for a full frame of data to come in to actually make the decision. So I may be tracking one small subset of the frame and, for example, the line on the side of the road. And that's what I'm looking at as I bring lines of video in.

The second general framework is really where programming ease is the most important parameter. And there's really two general buckets that the customers that use this model fall into. One is they're just trying to shorten the time to market, e.g., they want to make sure that they get to market as quickly as possible. The other case is actually where they prototype something on the PC and they take some additional steps to move the data around in a format such that it actually looks like the same memory model as a PC. I'll talk some more about that later in the session.

The bottom line is, in this framework, the focus is on making it easy to develop with for both novices and experts actually.

As a simple example here, I show video coming into the part and actually operating on it. It might end up involving more passes of the data, but the tradeoff really is formatting the data as you need it so that it looks like it fit in your prototyping environment.

The third category is really what I call "performance rules", which is where you select the processor first and really do lots more of what I'll call advance techniques to actually fit the application in the processor. It might involve a lot more time spent moving the data around and really thinking through every cycle as data moves around the system. And an example of this would be something like some kind of video encoder, decoder system where I could be taking a

line of video in, I might bring it into internal memory first to buffer it up in a line buffer fashion and then transfer it to external memory. But in the end, there's a lot of activity happening both from a data movement standpoint and from a processing standpoint.

Now, the reason, really, to go through that as an introduction, a quick introduction, is really if you look at no matter which bucket of applications your application fits into, there's some common attributes that come across. And really, the common theme is instruction and data management in the system. And as you'll see, hopefully, by the end of the session, it's important to take that, do that work upfront in your project and really save a lot of time and headaches later.

Hopefully I've given you a background as to why you want to keep watching the session. The next thing I want to do is really go over some features that are inherent to the Blackfin design that you can use and take advantage of. And some, as you'll see, you can take advantage of automatically, but others with, with just a small amount of work, you can actually exploit the feature and increase your system performance. So with that, let's go ahead and continue on.

Chapter 3: Tuning performance

Subchapter 3a: Feature Overview

I've lumped the categories of concepts together, and really they fall into four high level categories. One is the use of cache and DMA in your system. I'll go over what cache and DMA are and how to best select the combination of the two. I'll go over managing memory, getting into more advanced topics, and let you take advantage of some of the things you've learned in the other sessions on, for example, LDF files and how to actually map your code and data in the system.

I'll talk about managing DMA channels. The DMA controller is a powerful feature in the part and I'll talk a little bit about prioritizing the channels and how to make best use of those. And then finally, end on managing interrupts.

So the first of those topics is cache versus DMA, and what I'd like to do is just give you a high level picture of where cache and DMA fit in the overall space of the programming universe. As you can see, there's really two items that get traded off, performance and ease of use. For example, turning on cache is easy to do and the performance increases. But as we'll see, sometimes using the DMA controller can involve a little bit more programming effort, but with some real big high payoffs. And I'll show you some examples of this in a minute.

Subchapter 3b: Cache Overview

I want to start with cache, just to make sure we have a common terminology and common understanding of how the cache works on the Blackfin processor, and in general in embedded systems. I'll start with the instruction cache. It's important to just point out that a cache is really, by definition, a place where you hold "things". On the Blackfin processor, those "things" are really either instructions or data. The whole purpose of cache in an embedded system is to improve performance. Ideally we could put unlimited amounts of internal memory that all had single cycle access. But if we did that, or if any vendor did that, the size of the processor would be too large and cost would be prohibitive. So we try to balance out the amount of internal memory we put on the processor, along with the connection to external memory. As a result, we use cache to help us bridge that gap.

The purpose of cache in this case is really to bring in instructions and data into a single-cycle access space on chip and, ideally, as you access them, they'll be there in cache waiting for you. The nice thing about this is when you access the instruction or data in cache and they're actually there when you want them, they execute in single cycle in case of instructions or they're fetched in a single cycle in case of data.

As is the case with data cache and instruction cache as well, the highest bandwidth path into the Blackfin core is really through the cache-line-fill mechanism. The cache also has a nice feature in the sense that things that are most often used, are the things that actually stay in cache. So, for example, if you're executing code over and over again, it's used most often and with the Blackfin cache architecture, that's actually the code that stays into cache and is least likely to be replaced.

That's the instruction cache summary. I will now discuss a similar piece on data cache. Data either comes in from peripherals or data is generated statically and it's available in tables. In both cases, the cache can offer a high performance bandwidth pipe to bring data into the internal single cycle access memory.

In addition to all the features I've described on the instruction cache, the data cache has a few additional attributes, which we call "write-through" and "write-back". "Write-through" is actually a configuration option of the data cache, which essentially allows you to keep the source memory up to date. And what do I mean by source memory?

Well if I've got a buffer in external memory that cached, that's what I'm referring to as a source memory. And in "write-through", imagine that I actually read a value in from external memory and modified it a million times. If the "write-through" option is selected, every time I modify that value, it's propagated out to its source memory.

Alternately, there's an option called "write-back" which can further improve performance. And in that example I talked about where I'm fetching a piece of data and modifying it a million times, unlike "write-through", in "write-back", the data is not propagated out until it's time to actually be replaced in the cache. So it can actually have some significant system performance improvements.

And, in general, we've seen a lot of different applications. You're probably best to start off with "write-back". It's about 10-15% more efficient than we see with "write-through" for a given algorithm. Now the "write-through" does have its place, and basically if you're going to be sharing data between multiple resources- and by multiple resources, I'm talking about either two cores in the case of our, ADSP-BF561 dual core processor, or in the case of a single core, when I have either a DMA controller and a processor accessing the same data. In these cases, if you want to maintain coherency, the "write-through" is a better option.

Now, it's also important to note that you may do this while you're actually compiling your C code and you may not see as much of a difference. But it's important to actually try the "write- back" option when all the system peripherals are running too, because as we'll see later in the session, you're going to have some interactions between the various DMA controllers and core accesses external memory, and most likely you will see an improvement with "write-back", unless you're actually sharing the data with multiple resources.

Subchapter 3c: DMA Overview

With that introduction of cache, I want to talk about DMA and then I'll talk a little bit about when to use both. As you will see, there's no one answer, but I will give you some ground rules as to how you can go about seeing what's best for your application.

The DMA controller is an important piece of the Blackfin architecture, because it runs completely independent of the core. It doesn't cycle steal, that is, it doesn't take away cycles from the core processor at all. In an ideal configuration in your application, all the core really needs to do is set up the DMA controllers and respond to interrupts as data is moved. And so that's sort of the target that you want to try to shoot for as the model.

In general, all the high-speed peripherals and most peripherals in general, have DMA capability. And that's the capability that you use to actually move data in and out of the peripheral. We also have a set of memory to memory DMA controllers that let you move data either from external memory to internal memory, internal memory to external memory, or within memory space. And, again, as we'll see, the goal is really for you to set up, as part of your data movement system, to really take advantage of the DMA controller because for every piece of data that the DMA controller moves, it's one less thing that the core has to do.

I just wanted to show a quick example of a 2-D DMA which is a feature of the Blackfin processor that's worth exploiting. In this example, I'll start with a red, green, blue buffer on the left side. And sort of in the middle here is an intermixed red, green, blue, red green blue. And with a 1D to 2D DMA I can actually create multiple or separate red, green, blue buffers. And, you know, depending on which way the processing moving, it could be the opposite way, but in the bottom left corner of the chart you can see the sort of pseudo C code and that's code that if a processor was doing, it wouldn't be able to be doing something else, versus the DMA controller which if it's set up, have it move the data and get an interrupt when it's done. And the core could be off doing something else. And so this is just a simple example of that kind of concept.

Another important feature of the DMA and cache, and a lot of customers choose to take advantage of it this way, is that you actually turn data cache on to bring in data that was transferred in from the peripheral using DMA. So, the flow would go something like this. A high speed peripheral generates a data buffer that's deposited in external memory. Typically there'll be some type of ping pong buffer set up where while I'm filling one buffer, the processor is operating the other buffer. And in a lot of cases we found that the customers are actually able to use the DMA controller in conjunction with the cache.

And the one important point that's important to understand is that you have to maintain cache coherence between the data that the DMA controller's dropping in and the data that's actually in cache. And so, on the bottom of the chart I just want to make sure you understand the flow of data in that, again, I mentioned that a new buffer would be generated by a peripheral. An interrupt would come in. The interrupt would signify to the core that the data was there, ready to be processed. The core would then go off and process the buffer and when it was done, we'd go in and actually invalidate the cache lines associated with that buffer to ensure that the coherence was maintained between the two buffers. So, in the end, what you end up with is something that provides better performance than if you were just accessing the data from a pure core standpoint, but it wouldn't give you the full benefit of a full DMA model where you're actually taking those buffers and then using the DMA controller to bring them into internal memory directly. And I'll talk a little bit more about that in a minute.

Subchapter 3d: Cache vs. DMA

So, for instruction partitioning, it's a pretty simple flow chart of things to look at to get to your best performance. The first thing is in the simplest case is if it fits into internal memory, then that's where you map the code and you're pretty much done. You have desired performance. You have maximum performance because any instruction that gets executed gets executed in single cycle.

Now, in a lot of the applications that Blackfin is used in, there's often a network stack or lots of code that's running in external memory. And as a result, the best approach there is really to turn instruction cache on. It's amazing how true the rule of "20% of the code runs 80% of the time" is. This is good from a cache standpoint because the cache is typically much smaller than the end memory it's supporting. So it's good news in the sense that typically the code that runs most often is always the smallest. So a lot of times you'll be done right there, and it's most of the customers, I would say, would fall into this bucket, and so that's good.

For, I mentioned the three frameworks earlier, and you know, one of them was really performance oriented. So for that mechanism where you actually have more code that doesn't fit into internal memory and you don't want to use cache, you have the option of doing what we call an overlay mechanism, which is bringing the code into one memory bank while you're executing from another one and managing it that way. And as the line on the bottom shows, the more as you go across on this chart, it just requires more programming effort. So the nice thing about the Blackfin architecture is you have the choice, you know, you can satisfy all those pieces in the continuum.

Similar discussion on data DMA and cache. If you can fit the buffer into L1 memory, and in a lot of applications that's possible, then you definitely map it internal memory. If you're using DMA as part of the programming model, and in most cases the DMA is as a minimum brought in from a peripheral standpoint, but the question is, when you get the big buffer into external memory, how do you actually bring it into internal memory to process it? And in that case, again, it's usually a combination of either DMAs with mem DMAs set up to bring data in as you need it, or some combination of that and turning data cache on and invalidating the buffer before you access it the next time.

So, there's no one answer on this, and typically it's a combination of things that work best. So I think the most common configuration that seems to work the best for the largest number of customers, is the case where instruction cache is enabled. Things that are static, like tables and data that doesn't change would be handled through data cache. And then data that's changing is best handled with the combination of peripheral DMA to external memory, and then mem DMA to internal memory.

Subchapter 3e: Memory Considerations

Ideally now you've got some idea in terms of looking at it from a macro level, DMA versus cache, and it's important to also, kind of the same lines, understand the memory architecture.

Not to become a memory architecture expert, but just go over the handful of things that you have control over that can make a big difference in your system performance. So with that, I'd like to just quickly review the basics of the memory architecture.

On Blackfin, you know, the cores run at what we call a core clock, which is typically 600 MHz for the parts. We offer a wide range of frequencies, but I just use this frequency here. The various levels of memory, again, are put there so that the customer has the best combination of on-chip resources versus off-chip resources. And by level one memory, we really are talking about the memory that runs at the same core clock rate. Same rate as the core clock, which is again, 600 MHz in this example. You get single-cycle access per instruction and per data. The memory is typically configurable. It's either SRAM or cache. And usually the sizes range in the tens of Kbytes, maybe a hundred Kbytes max.

Moving out a little bit, still on chip, but a little bit further away from the core if you will, where can have several cycles to access is really what we call our on-chip L-2 memory. And instead of being in tens of Kbytes, it's really in the hundreds of Ks. 128, 256, that kind of range. And then moving off chip, we go to, instead of having single core clock access, we're measuring accesses at the system clock, which is typically 133 MHz, and instead of sizes measured in Kbytes, hundreds of Kbytes, we're talking about 100's of megabytes.

Before I go into a little bit about the bank structure, I think it's important to point out that on Blackfin, in a single core clock cycle, the 600 MHz clock rate, the processor can perform one instruction fetch and up to two 32-bit data fetches or one 32-bit data fetch and one 32-bit data store. That all happens in a single clock cycle. The other point that I made earlier is that the DMA controller actually can access similar banks or the same kind of bank architecture without stealing any cycles from the core. In a lot of cases, access sub-banks without any bank conflict stalls, and I'll go through that in a little bit.

I'm going to go over internal memory and external memory, but I want to start with internal memory. And on Blackfin, all the internal memory banks are built up of sub-banks. And the reason they're built up of sub-banks is so that we can, again, have the core accessing multiple pieces of data in the same cycle that we have the DMA controller accessing as well.

And just at a high level, I just drew this illustration just to show sort of what's happening, is the solid lines in these banks represent the sub-bank definition and in this case, I've got all the buffers and coefficients sort of bunched into two sub-banks in internal memory. And so what ends up happening is if the core and DMA are accessing the same sub-bank, it's possible that I could get stalled, and that's not desirable.

On the right, on the right side of the chart, I've spread out those buffers so that I can have DMA access and core fetches, actually fetching from the buffers in harmony, as I say. And I'll go a little bit more into that in a second here, but what I'd like to do is first go over the internal memory bank structure. Blackfin in general has blocks of 16 Kbyte configurable banks. And each of those 16 Kbyte banks are made up of 4 Kbyte sub-banks. As I mentioned earlier, they can either be SRAM or they can be cache.

When it's configured as SRAM, it's single cycle access, it can be accessed by the instruction fetch unit, as well as by a DMA controller to fill instructions in if necessary. It also can be configured as a cache, in which case it behaves as a four-way set associative cache. And we talked about that.

So the next piece is looking at the data side, and similar to the instruction memory bank, we again have four 4 Kbyte sub-banks for each 16 Kbyte chunk. Again, the goal is really to allow multiple core fetches to different sub-banks and at the same time a DMA controller connects with a third sub-bank. One other note is that when actually the address- actually the core fetches to the same sub-bank, if the address bit two is different, that is they're on even and odd boundaries in the same cycle, you can actually avoid a stall as well. But it's important to understand how this architecture works because in a sense it, if I'm operating on four Kbyte chunks of data, I can have the DMA controller filling one of the sub-banks while the core processor is actually accessing data in other sub-banks that were previously filled. And when that's done, you know, the bank's operations kind of switch.

So now we've talked about internal banks, I'd like to move to external memory banks. In this case, it's important to understand a little bit about, for example, SDRAM physics as I call it. Essentially any time you access an SDRAM on a page that hasn't been activated already, it consumes multiple system clock cycles. And so in some architectures, every time you access a piece of data or an instruction that's more than, let's say, one Kbyte or two Kbytes apart, you actually take a penalty, a latency penalty to access that row.

On Blackfin, we have a feature where the external bus interface unit actually keeps track of up to four rows open across the four SDRAM internal banks. So any SDRAM you connect up to Blackfin will typically have four internal banks associated with it. In this example, I have four 16 megabyte banks connected up to our external bus interface unit. And, again, similar to, but for different reasons, similar to what you saw in the internal performance, it's important to take advantage of the internal banks the SDRAM has to offer.

It's typical, especially in external memory, to sort of bunch all of your buffers up together in external memory. So for example, I might have code immediately followed by a video buffer,

immediately followed by another video buffer. And if you're not careful, the default could actually have you put those back to back, in which case they most likely would be lumped together in an internal bank.

In our LDF file, we try to help you do that. But again, depending on which size SDRAM you're using in your system, it's important to take a look at that and make sure that you're taking advantage of the multiple internal banks in SDRAM. In this example, it works out nicely because typically I'd have some code in one section, and if I separate out my buffers, it doesn't really require a lot of work, but it really can have a big payoff in terms of performance.

Subchapter 3f: Key memory benchmarks

I talked about that I was going to talk about benchmarks a little bit, and I'd like to really just boil it down to one chart because for further reference, by the way, on benchmarks, the chip bus hierarchy sections of all of our hardware reference manuals usually have a table that actually gives you the full gamut of benchmarks of accessing various busses. What I tried to do is summarize the most important ones, emphasize again, that it's really important to try to use the DMA controller whenever possible.

In the case of SDRAM, it really is important as well because for a 16-bit access to external memory, the Blackfin takes eight system clock cycles to access. And for a 32-bit access it takes nine system clock cycles. Whereas with the DMA controller, assuming that the row has been activated and assuming you're on an open bank and there's not a conflict, you can access data either read or write, every system clock. So again, the bottom line is that the data and instructions are best moved with the DMA controllers.

Chapter 4: Managing resources

Subchapter 4a: Shared resources

We've talked about cache, we've talked about the DMA, and we've talked a little bit about the memory architecture and some of the highlights of things that you need to understand as you're putting together your system. The last two items are going to focus on managing shared resources and, in our case, the most important one is really managing the external bus. And then also managing interrupts.

So I'll start with the shared resource piece. And I want to focus on the external memory interface. And I use the BF561, which is a dual core processor as an example because it is a superset of everything you'll see on a Blackfin processor. The important item to understand here is that by default, the core has priority to the external bus, over the DMA controller. Now a couple of

important points to note. By core access, I could either be fetching instructions or fetching data via the core. That could also include something like a cache line fill where the core is actually fetching a cache line.

The other point is that, as I mentioned, by default the core beats the DMA controller, unless the DMA channel is actually urgent. And I'll talk in a few charts about what I mean by urgent. But the important point is that you understand that and also that you know that actually by default, you can actually change the priority and make the DMA controller always look urgent, in which case it would always beat the core accesses.

Now on the bottom of this chart, you'll see that there's multiple cores here and there's multiple DMAs. That actually applies to a subset of the processors. For example, the dual core piece applies to the 561 where there's actually multiple cores in there. I wanted to highlight that the way to change the priority between the core and the DMA controller is really through a bit in one of our EBIU registers and it's really the asynchronous memory global control register. And that can be used to change the priority between the core access and the DMA controller as you need to fit your application.

I mentioned earlier, I used the term urgent and it's an important concept. I think it's a fairly simple concept to understand, but there's only two scenarios that you can have an urgent condition. In one case, the peripheral is transmitting, in once case the peripheral's receiving. So in the case where the peripheral's transmitting, I get an urgent condition when the DMA FIFO is empty, but the peripheral is actually requesting data. So what that really means is that something is preventing the DMA controller, for example, the core in this case most likely, from filling the DMA FIFO. And what it's trying to do is prevent the case from the peripheral under-running, so there's still data in the peripheral FIFO, but it wants to keep that peripheral FIFO fed so there's not an under-run condition. So what it'll do is it'll generate an urgent condition, which can elevate the DMA's priority in the scheme.

Likewise, when the peripheral's receiving, the condition that we're trying to prevent is an over-run. And so in this case, instead of the DMA FIFO being empty, it's actually full. And as soon as the peripheral tries to provide a sample to a full DMA FIFO, the urgent condition is set. And again, the purpose of the urgent piece in our system is really to prevent the peripheral from actually underflowing or overflowing. And it gets back to why was this DMA FIFOs be empty or full is because of some shared resources actually is being consumed by some other portion of the processor.

In the end, this is sort of a high level summary of who wins and it's probably a good reference chart for you to understand, is that in descending order at the external bus, a locked core access,

something like a testset instruction, it's an atomic instruction is once it starts it goes to completion. But after that, the order is urgent DMA, cache-line fill, a core access, and a DMA access. So in the case of a system where there's no urgent DMA and you haven't set that bit that I talked about, the core will essentially win over the DMA controller.

As part of that, you know, a lot of times we'll see customers that will be accessing either in a tight polling loop, accessing external memory, or operating some tight instruction loops from external memory. Just be aware that's really where this arbitration comes in and understanding at this level is important to see what's going on in your system. At the L1 side in descending priority, DMA accesses take priority over core accesses to L1 memory.

Subchapter 4b: External memory

Okay. So one other piece of the tuning performance discussion. We talked a little bit about the physics of memory architecture; the multiple sub-banks, internal memory, multiple internal banks of external memory. One of the other things that's very often not understood is that as part of SDRAM physics, there is actually inherent latencies when you're actually doing reads after writes, for example.

The bottom line to understand at this level is that transfers in the same direction are more efficient than a set of intermixed accesses in different directions. And that's sort of the most important thing to understand because in the end what we'd like to be able to do is group transfers as much as possible in the same direction to reduce the number of these turnaround latencies associated with external memory. And on Blackfin, what we tried to do is allow you to tune this yourself and we have a register that we call the traffic control register, that lets you sort of improve system performance when multiple DMAs are ongoing in a system. And that is a typical system with multiple DMAs going on at the same time.

As you'll see in this functionality, depending on the values you program, you can actually increase the amount of traffic in one direction and what it'll do is help you group transfers in the same direction. The nice thing about it is, it's just a register that you program and there's actually multiple traffic control field in the register. There's one for what we call our DMA access bus, there's a DMA core bus, but the one that really has the biggest impact in your system is what we call the DMA external bus, the DEB. And so I've highlighted that one here because that's the one that you want to start with. And that'll give you the biggest bank for your buck in terms of performance.

We tried to have some type of formula that gives you a magic number, but really it is application dependent. What we found is that if there's not a lot of things going on in your system, then

typically the larger the value, the better. Now the maximum value is 15, so usually a value of 15 is something that makes sense. But more likely, you've got more than three things going on in your system. Typically when there's a lot of things happening, something closer to the mid-value, typically four to seven is usually better for this characteristic.

Subchapter 4c: DMA priorities

With that, I'd like to go into the priority of the DMA controller channels and just, again, this is an area that just understanding a little bit will go a long way in terms of helping you sort of set up your system. So, I've talked about the DMA controller, how it fits in the system compared to cache, how you can use it in concert with cache, and moving data round the system. What I'd like to do now is just quickly go over the DMA controllers themselves and really focus in on prioritizing within the DMA controller.

It's important to understand that each DMA controller has multiple channels and in general, if one DMA channel tries to access the controller or any given bus at the same time, the highest priority channel wins. And the good news is that the DMA channels are actually programmable in priority, so you can actually put the ones that you feel are most important for your application in the proper order. We have a default configuration for you. But again, if you want to tune that for your application, the channels are programmable.

I should note that when more than one DMA controller is present, for example on the BF561, then we also provide arbitration that's programmable between the DMA controllers. It's important to take advantage of the DMA controller and with what we call our DMA manager which is provided as part of the system services, really provides you with a good set of tools and APIs actually access the full functionality of our DMA controllers and really allowing it to interoperate with a range of operating systems and kernels or, in some cases, just a basic flowing application.

Subchapter 4d: Interrupt processing

I'd like to go over interrupt processing, because this is something that's really important in embedded systems, I'm sure I don't have to tell you that. But the most common mistake from an interrupt standpoint is really when people are getting started in their application is having one task spend too much time in an interrupt service routine and, in general, preventing some other critical code from executing.

From a Blackfin architecture standpoint, interrupts are disabled automatically once an interrupt is serviced. And we have a return register for each event, of which an interrupt is an event, and until you or the software actually program to save the return address to the stack, higher priority

interrupts are not enabled. So once you're in an interrupt service routine, and you save the return address to the stack, it then allows higher priority interrupts to actually happen.

It's important to understand in your application how much time you're spending in each ISR and really, the question is how you can best nest interrupts. And by nesting, I'm talking about again, being able to interrupt one interrupt level with an interrupt at a higher level.

Similar to the DMA controller, the most important interrupts are usually the highest priority, should be programmed at the highest priority. We have a default configuration, but again, you can program the levels of the interrupts to meet your application. You can use nesting to ensure that higher priority interrupts are not locked out by lower priority events. Similar to the DMA controller, we strongly recommend that you look at the call-back manager that's provided with System Services because what it allows you to do, is it allows you to service all your interrupts quickly and return to the lowest priority interrupt. What that really ensures is that higher priority interrupts are not locked out for any amount of time. But you have complete flexibility on how you actually architect this in your system. The system services are there to help you configure it for the best performance in your application.

Chapter 5: An Example

Subchapter 5a: A Video decoder

We've gone through a bunch of different techniques, and I want to show sort of how they might come about in a real example. And really, it gets into the discussion, again, of setting up the frameworks at the beginning of your application, understanding the highest level basics of the memory, the cache, and the DMA just to see which piece you should be using. And then effectively taking advantage of the processor resources in the easiest way possible.

So, for this example, I've chosen a video decoder and really, it represents sort of a complicated data flow in a system. Lots of data moving around the processor. Lots of performance required to actually process each pixel. And you can apply this same sort of a thought process to a bunch of different applications. I start with two items that are important. One is from a processing standpoint, one question you need to answer is really how many processing cycles can you allocate before you even get started in your system to each processing element?

Now in a video example, the processing element is really the picture element that you're actually working on. So I've started with a D1 video frame, a 720 X 480 pixel frame running at 30 frames a second. And when I calculate out the number of pixels that I have per picture element per second, I can get a good budget just going into the application. It forces me to think through sort of where I'm going to be spending time in the application.

In this example, we have on the order of fifty cycles per picture element. Now, along those lines, it's important to make sure you leave some amount of mips left over for things like, for example, audio processing in this case, system and transport layer mechanisms, maybe a small kernel, or OS and so making sure you account for that up in the upfront budget.

Equally important to the processing budget is what I refer to as the bandwidth budget. And if you just look at some of the macro-level things that are happening in a video decoder, you start with an encoded bit stream coming in via some external interface. You've got reference frames being created in external memory. You've got lots of interaction between the external memory and internal memory to get data in and out of the system as you build up your display frames.

In the end, at a macro level, the first thing you have to do is add up the bandwidth of each of those and make sure that it fits in the system bandwidth of the part you're using. In this example it's on the order of a 130 megabytes a second. Now as we'll find out after the example, that's not the only thing you have to do because of all the things that I've discussed in this presentation, to have some types of interactions you have to do a little bit more than that. But I'll get to that in a second.

So, just to kind of give you a picture of what I'm talking about, the input, the stream is input into the system. It's a series of encoded packets. These packets are decoded and slowly, and this is true in kind of a high level picture, it applies to really any video decoding process where we're actually building up reference frames and using those reference frames to build up the actual display frames that actually go out either on the LCD or the video encoder to a monitor display.

You can see that because of the size of the buffers we're talking about, in megabytes per second, you know, we're easily going to be utilizing external memory in this application. And you know, it maybe obvious but the larger buffers have to go into external memory. So in this case, let's place these reference frames and display frames, as well as the packet buffers into external memory, SDRAM.

And, of course, anything that's either static in terms of look-up table, or any of the result buffers that I'm actually going to be building as I go through the application, we want to have those in L1 memory, because again, we're going to get the single-cycle access, those memory buffers. And just to show sort of what's happening here, on the left-hand side of the chart, I've got these larger YUV buffers, these are components of video in this example, that I'm actually going through using the 2D DMA feature and pulling out macro-blocks into L1 memory and so the whole purpose, really, is to schedule that so that when I'm actually ready to process them, the data's sitting there waiting to be processed in single-cycle access. Likewise, when I'm done, I can build up the result

into what I call the result buffer, the display buffer that actually get sent out, the video port, either to an LCD or to a video encoder.

In the end, I tried to give you kind of an accounting of sort of some of the things that are happening in terms of megabytes per second. And you've got a lot of stuff going on in different directions. You know, there's lots of data being moved around the system. And so while I said that the first step is to add up the bandwidths and see sort of where you are at a macro level, that's not the only thing you have to do. You really have to look at the interactions between the busses, and a lot of it's driven based on the way external memory behaves and how, you know, as we talked about, some of the physics being bus turnaround, concurrent activities, all need to be considered upfront.

Chapter 6: Conclusion

Subchapter 6a: Summary

So, hopefully by now you've seen some of the techniques that you can go through to get better performance on Blackfin. One of the things you'll find with Blackfin processors, we've taken a lot of steps to allow you to take advantage of the features without actually having to do a lot of work. But as you've seen in the session, there are some factors that are important to understand that hopefully you'll find straightforward and that our tools help you actually manipulate. For example, moving data around the system, using our DMA controller, managing interrupts. We have a combination of tools and System Services that allow you to access those features of the part.

So I hope you found it useful and thank you very much.

We also have a bunch of additional information that you may find useful as well. Our hardware reference manuals are available for each of the Blackfin processors. We also have a programmers reference that I think you'll find useful. And there's also a text that was done called *Embedded Media Processing* (D. Katz and R. Gentile), which has some of those same kinds of concepts explained in more detail.

If you have any further questions, I invite you to click the "ask a question" button, and thank you.