Real-time apps using multi-threads

Paul Evans, Business Development Manager, Meta CPUs, Imagination Technologies

Consumer electronics devices are increasingly high performance, real time applications where high data rates and sophisticated real time event handling are required. Delivering suitable real time performance can be achieved by using a variety of techniques – including multiple processors. A more effective solution (low power, low cost and small silicon area) is hardware multithreading which delivers real time response in a tight silicon and power budget.

But wait a minute! Aren’t real-time and multi-threading diametrically opposed? If I have two processors in my system, one for real-time, say a DSP audio algorithmand the other for non-real-time, say runningthe User Interface (UI) and communications (Ethernet) etc., thenhow could I ever combine them? How can hardware multi-threading offer the determinism required for real-time tasks to meet their schedule?

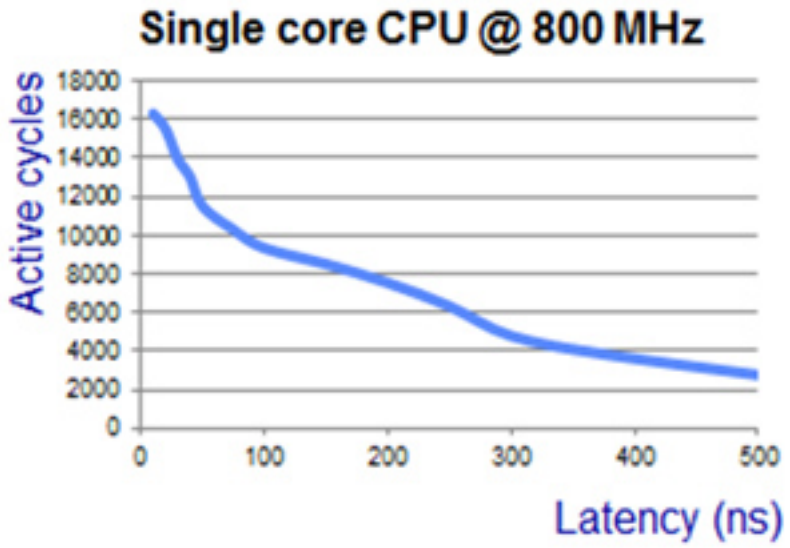
With a conventional single threaded CPU, if you schedule tasks with disparate event rates and activity patterns on a single thread using a common OS, then you are likely to quickly encounter problems.

However, hardware multi-threading (distinct from software threading) allows separating real-time tasks with widely differing scheduling requirements into different software schedules. This means a subsystem based on a complex multi-functional OS such as Android can run on one hardware thread while a real-time data-driven DSP task runs on another. The DSP task might be based on a simple synchronous IO driven scheduling strategy which would be completely independent of the interrupts and device drivers in the other subsystem.

How is this possible?

It requires a processor with 3 capabilities. 1) Multi-threading, 2) Simultaneous Multi-threading (SMT) and 3) Intelligent scheduling & prioritization.

Why hardware multi-threading and not super-scalar?



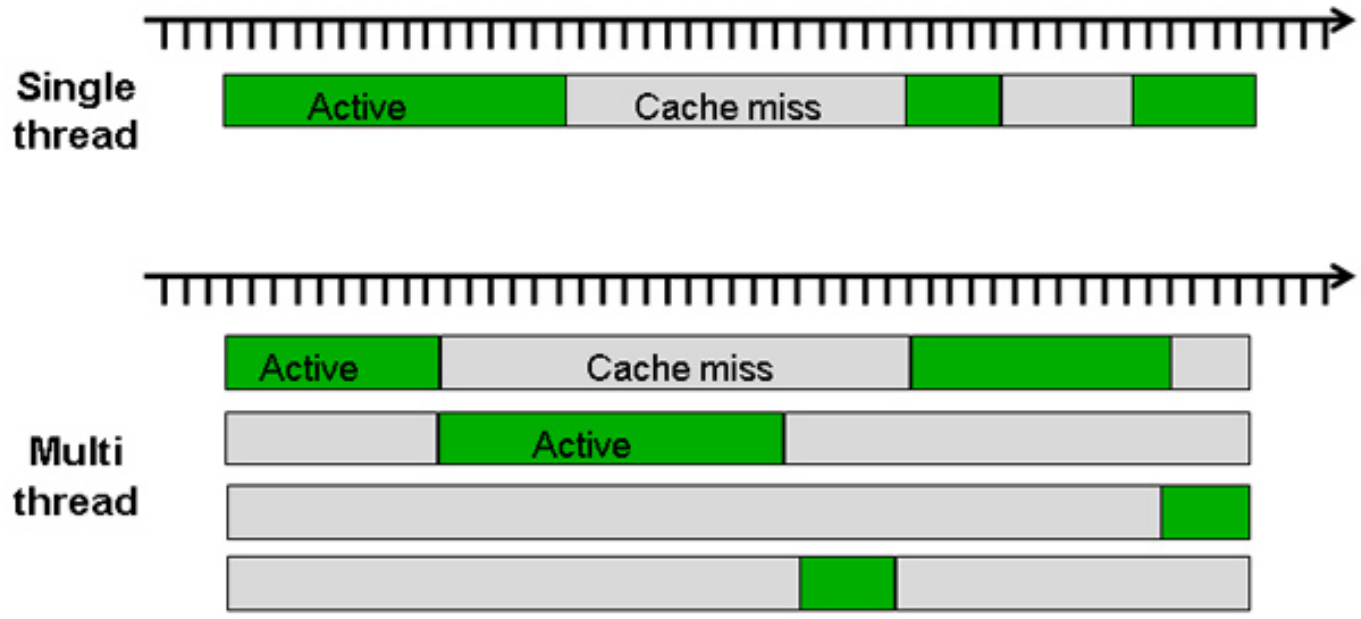
In super-scalar you are launching multiple instructions from the same thread. Most standard programs don't inherently have a lot of fine-grained parallelism in them and so finding a 'close-by' second instruction to execute in the same cycle is difficult because it commonly it isn't there. In multi-threading you are launching multiple instructions from different threads or programs. Your system now looks like multiple processors running multiple OS or programs independently from each other, occasionally communicating. This makes programming a hardware multi-threaded processor, like the MetaCPU developed by Imagination, straightforward since you are not trying to write a program designed to handle parallelism; you're writing normal independent programs and just running them on the same silicon.Meta provides an inter-thread communication protocol to talk between the virtual processors.

But aren't these threads or virtual processors all fighting for the same resource? Not really, cache missescreate space in time or idle resource. And a technique known assimultaneous multi-threading maximizes resource usage rather than creating resource conflicts as we shall see.

Multi-threaded CPUs do share many resources but to accommodate the multiple threads, each execution unit holds a local register state, an execution pipeline and a program counter (PC) per thread. Additionally, separate control units hold mode bits and control registers for each thread. This means not everything is shared. Yes, the core is 10-15% bigger than a single core but you're getting dual core performance without the power, silicon area or wastefulness of idle pipelines of going to dual core.

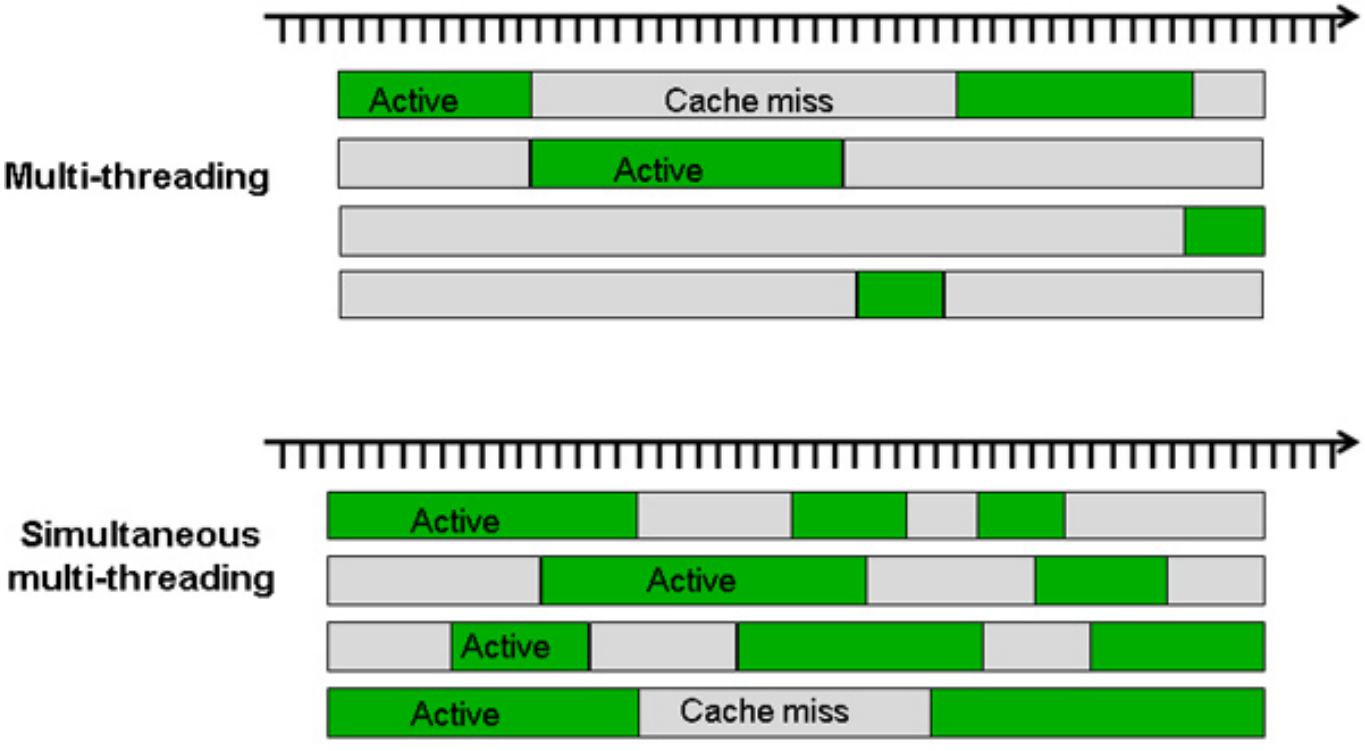
Cache misses mean that conventional processors are frequently stalled. The problem gets even worse when you think of a highly integrated SoC with many requestors on the bus all trying to get at a common shared memory. When the processor experiences a cache miss it may have to wait hundreds or thousandsof cycles – and thatequates to a lot of lost processing power while sitting idle.

If you're running multiple threads instead of remaining in a paused loop, you can hand that processing power over to the other thread and allow the second thread to utilize the time and resource.



Ok, so that's a neat idea to use time that would have otherwise been wasted -- clearly that's good as long as you don't spend a lot of time switching the context. However, it still doesn't explain how this system could be deterministic for real-time systems.
We will address these concerns; first let's take multi-threading one step further and hit the overdrive button.

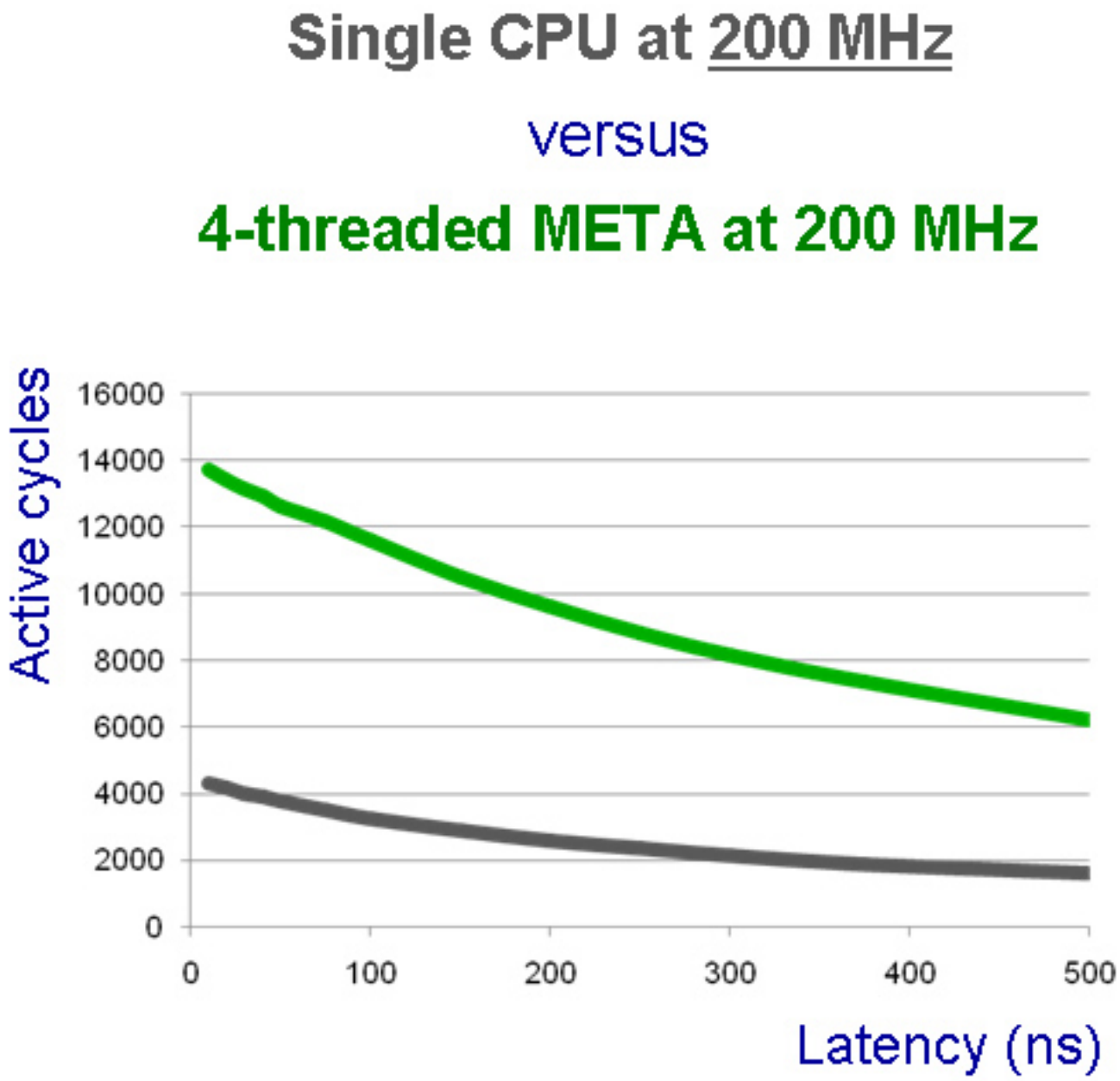
Simultaneous multi-threading (SMT)
SMT allows these multiple threads to execute in parallel (rather than simply utilizing only the cache –miss time).



In this circumstance the hardware is starting to do something a little more

intelligent. First it has a shallow instruction FIFO for each thread. It decodes the multiple instructions then takes a look at which resources those instructions require and at the same time looks at the execution units and sees what resources are available. If the scheduler can match ‘available resource’ to ‘required resource’ then the instruction will execute. It does this for every thread in parallel on every cycle.

Look at the startling results where a 4 threaded CPU like Imagination’s Meta HTP can achieve the same amount of work at 200MHz as a conventional processor can achieve at 800MHz. That highlights the affects that memory latency (created by many requestors and shared memory, especially on an SoC) on CPU performance.



The scheduler will always attempt to launch the maximum number of threads possible on every cycle. With the Meta HTP CPU that is up to 4 threads per cycle. Put another way, on every cycle the scheduler always attempting to run 4 separate programs or 4 x Main() in parallel.

Unless that is,you want to tell the scheduler to do something different. And that brings us back to real-time and determinism.

Real-time and determinism with SMT

Now you have a very powerful system that can look at available resources, look at desired resources and on a cycle by cycle basis decide how to assign those resources to maximize the work being done. If you add prioritization into the equation you have a very interesting mix. First you have the multi-threading which you can view as independent virtual processors. On to this you add your real-time task which the scheduler can set as the highest priority task and you have resource matching that occurs on a cycle by cycle basis. You can allocate precisely the right amount of resource to keep your real-time application on schedule and use any spare cycles or spare resource to keep you non-real time OS running. Enter AMA, or:-.

Automatic MIPS allocation -- a patented technology from Imagination

With AMA you can say ‘I know my real-time Audio DSP requires 100 MIPS’ and so you can assign thread 0 the top priority and say it must always have 100MIPS.

AMA provides automatic resource management in hardware, ensuring that each thread of execution gets the MIPS it needs and has the required response time. It allows thread instruction issue rates and priorities to be controlled dynamically based on a pre-defined priority assigned by the user.

Since many embedded applications in the communications and consumer space are required to perform to a minimum level in order to meet user expectations, e.g. video frame rate and audio quality with no dropped packets, these systems have historically shied away from trying to combine too much, but AMA allows system architects to find new ways to reduce silicon area, licensing cost, fabrication costs, power usage and heat dissipation all without affecting the quality of service

How AMA works

In this example threads 0 and 1 have elevated priorities for real-time response. Thread 2 is controlled by AMA for an execution rate of about 40% instructions/clock.

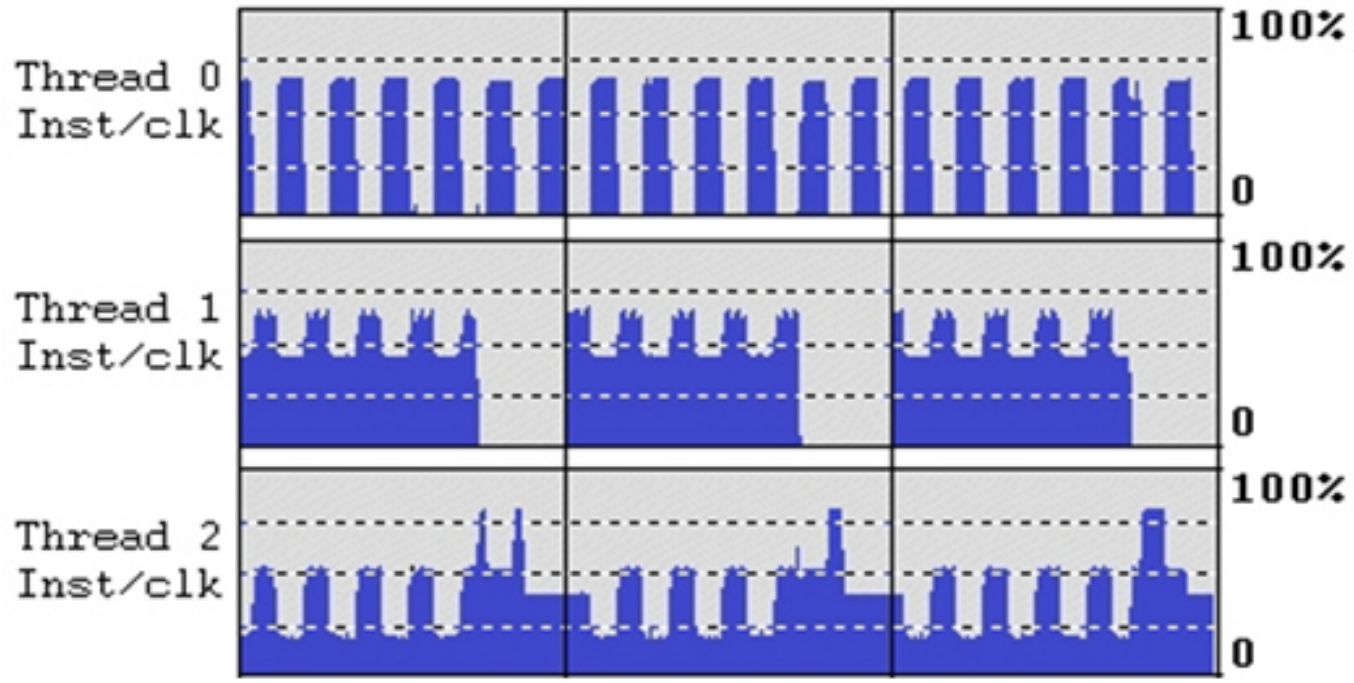


Figure 2. AMA rate control.

Since thread 0 and 1 have a higher priority than thread 2, then thread 2 cannot always run at its desired rate and builds up a processing deficit.

Whenever possible thread 2's execution rate is increased to make up the deficit. You can see this happening in Figure 2 immediately after thread 1's activity burst. Note how thread 2's execution rate is increased to make up the deficit, and after catching up the execution rate returns back down to a lower value.

In this manner AMA is able to use spare resourcesto run the non-real time tasks and always maintain the required resource for the real-time activities.

Real-world example

Let's see a real-world example of this working. We'll focus on Pure's DAB radio. Pure is the world's leading DAB radio manufacturer and as you know with radio the broadcast is real-time --, you don't get an opportunity to drop packets or retransmit. Pure combines Internet radio and conventional radio into a single device running on a single multi-threaded processor. Along with the audio it also has a Linux based UI that connects to the Internet.

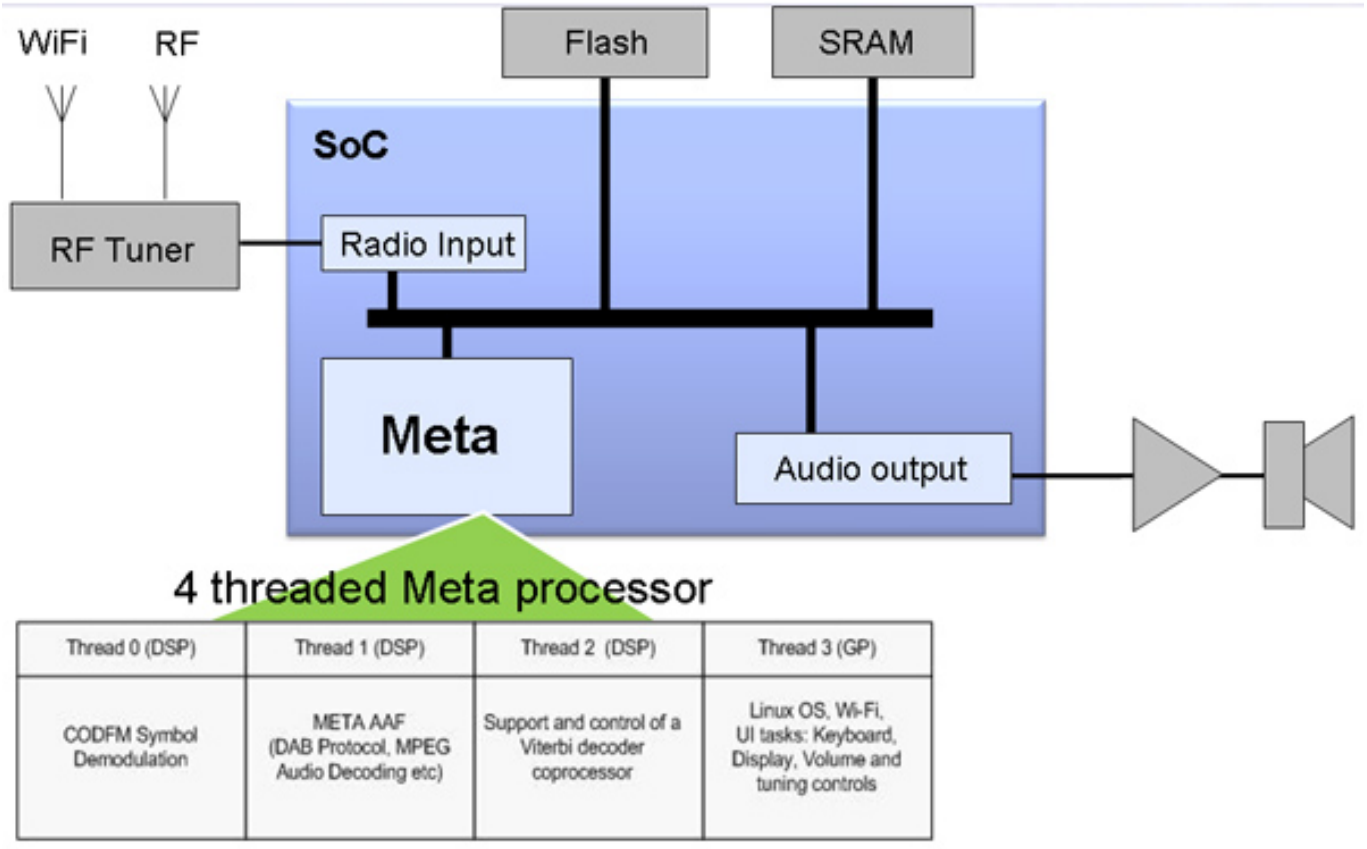


Figure 3. 4-threaded deployment in a DAB/Internet radio application.

Figure 3 shows a real-world implementation of a 4-threaded Meta HTP SoC where threads 0 and 2 are dedicated to real-time DSP tasks. Thread 1 runs the Meta Advanced Audio Framework and is fed from the other threads.
Let’s close with looking back at a couple of earlier concerns, first -- determinism. We’ve outlined how AMA always ensures that the real-time process gets the MIPS it requires. It’s up to the actual use to determine exactly how many MIPS are required but once that data is supplied to the schedule it will always ensure that thread has at least that many MIPS.

Second, time taken to context switch. This processor isn’t really context switching the classic software sense of the word. Rather it is always attempting to launch the maximum number of threads that it can. It can opt to give more resource to one thread compared to another and it can do that on a cycle by cycle basis in which case you could view the threads as having zero cycle context switching but the chances are that the core is nearly always running multiple threads at the same time and never spending time storing one context to switch to another.

Summary
The uniquely flexible and scalable Meta architecture has all the benefits of multi-processing, but with less silicon resourceand development complexity than a multi-core system. Simultaneous multi-threading addresses the real world SoC challenges conventional processors simply cannot. And thanks to AMA, Meta is ideally suited toreal-time applications by ensuring resources are made available to complete real-time tasks on schedule and spare cycles are used to execute the non-real-time tasks. All this means that Meta is able to deliver a significantly lower cost and lower power approach to real-time solutions than a multi-processor approach.

Paul Evans is business development manager, Meta CPUs, for Imagination Technologies. Imagination Technologies Group Limited (LSE:IMG; www.imgtec.com) – a global leader in multimedia and communication technologies – creates and licenses market-leading multimedia IP cores for graphics, video, and display processing, multi-threaded embedded processing/DSP cores and multi-standard communications and connectivity processors. Contact Mr. Evans at paul.evans@imgtec.com [1].

Links:
[1] mailto:paul.evans@imgtec.com