

Highlights

- ◆ Demonstrates Cascade methodology with MP3 Decoder case study.
- ◆ Create industrial strength, royalty free, coprocessors from Open Source software.
- ◆ Rapidly develop programmable, high performance application accelerators.
- ◆ Offload computationally intensive tasks from the main processor with minimal area cost.

Introduction

Cascade's high level, software-centric methodology allows the user to rapidly evaluate design alternatives and make improvements to the coprocessor. The user can experiment with alternative parameters within Cascade and use the feedback provided to further refine the design to meet the needs of the project without investing large amounts of effort in the details of processor implementation. Such an approach ensures the coprocessor and the software running on it will be the best fit for the system as a whole.

This paper discusses how Cascade can be used to produce two very different coprocessor implementations for the real time playback of a 48KHz MP3 stream based on the same open source software decoder. The techniques introduced can be applied to any software application to meet the specific design requirements of any customer project.

The first approach is to minimize the area of the coprocessor. The assumption for this scenario is that the target platform will be clocked at least 66MHz but silicon area is at an absolute premium.

The second approach looks at how to maximize the performance of the MP3 decoder running on a Cascade generated coprocessor. In this case minimization of clock frequency is the driving design requirement.

System Level Design

Since the system containing the decoder may be unknown at design time, the flexibility of Cascade's software driven design approach allows a modular solution to be developed featuring very simple hardware and software interfaces for ease of integration.

The original open source API supplies a frame by frame decode function which was improved by the addition of a wrapper which manages the flow of data through the frame decode function. The top level API to the decoder then simply consists of three function calls detailed below.

Function	Description
<code>long decode (long pBytes);</code>	Decodes <code>pBytes</code> from the input stream to the output stream. Returns the number of bytes decoded.
<code>void set_reader (int (*pReadDataFunc) (unsigned char* pBuffer, size_t pByteCount));</code>	Sets the function to be used by the decode function to read data from the input stream.
<code>void set_writer(int (*pWriteDataFunc) (short* pBuffer, size_t pByteCount));</code>	Sets the function to be used by the decode function to write data from the input stream.

This approach allows for a clear and efficient offload point for the entire decoder onto the coprocessor, minimizing the communication overhead with the ARM as well as supporting easy integration with the surrounding application. Since the reader and writer functions are passed into the decoder from a higher level, the decoder can be easily ported to different systems by taking advantage of the programmability of Cascade coprocessors.

Since the software interface allows the decoder to execute independently of the rest of the application the coprocessor hardware should also support this behavior. In Cascade, the coprocessor to be generated is specified with an AHB Master interface allowing it to access memory independently once it has been started. The diagram below illustrates a potential system architecture into which the coprocessor may be placed.

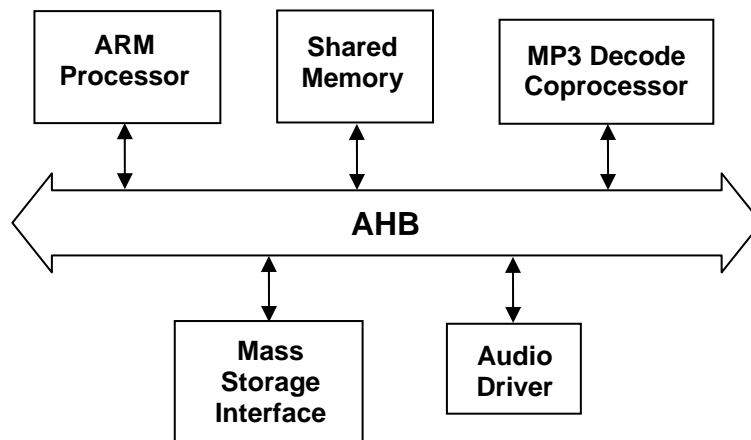


Figure 1: Basic System Design

By default the coprocessor will cache all data however this is not desirable if the coprocessor is going to access memory mapped peripherals which source and sink the audio data. This is addressed by

defining memory ranges as volatile within Cascade. The data cache will then be bypassed for accesses to these memory areas.

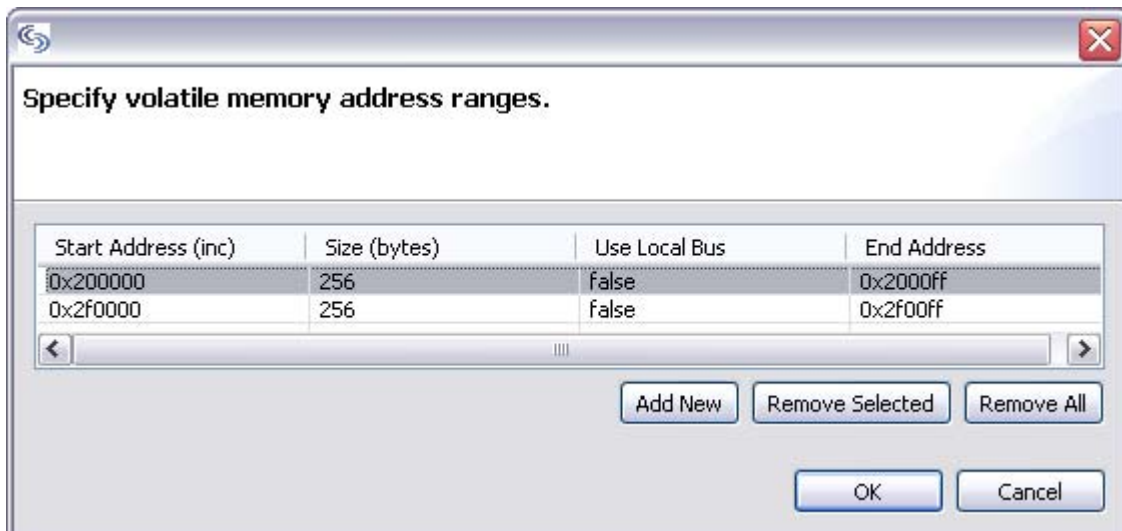


Figure 2: Defining Volatile Ranges

Minimizing Coprocessor Area

By default Cascade will generate a range of candidate architectures for a coprocessor. This allows a trade off between performance and area. Further design options can be generated by increasing the synthesis effort level which expands the search space covered by the tool. The results in this case were at 30% effort level.

Constraining Architectural Synthesis

Cascade can be configured to favor smaller logic area when generating candidate architectures. However, this must be balanced against the need of the higher implementation clock frequency.

The basic settings governing this are configured in the Architecture settings for the coprocessor. Setting the reprogrammability to maximum decreases the amount of instruction decode logic generated for the coprocessor, which for complex code may save over 10K gates of logic against the default settings. The trade off here is an increase in microcode size which may lead to a larger instruction cache requirement. However, since the cycle budget in this scenario is not too restrictive there is scope for using a smaller instruction cache without the higher miss rate becoming unacceptable.

In order to support the 66 MHz target clock frequency, operation chaining is set to zero. This means that the output of every functional unit in the architecture is registered, resulting in a heavily pipelined processor.

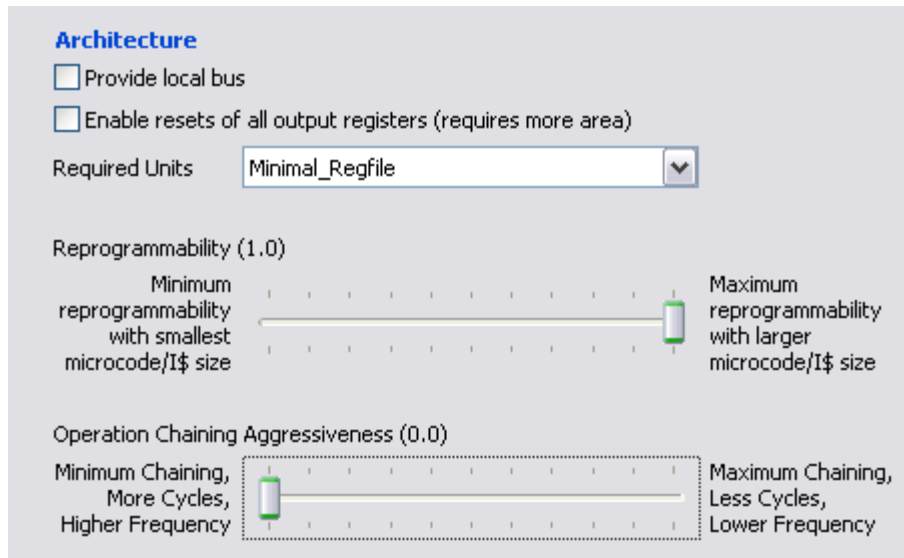


Figure 3: Setting Coprocessor architectural constraints

The Required Units option allows the user to select a base set of units that will be included in the architecture. By selecting the *Minimal_Regfile* profile, Cascade will use a reduced core register file set that will reduce the amount of dual port RAM required in the design as well as forcing more serialization of the instruction set due to the lack of temporary storage available. This will also result in less logic, although this comes at the price of further reduced performance.

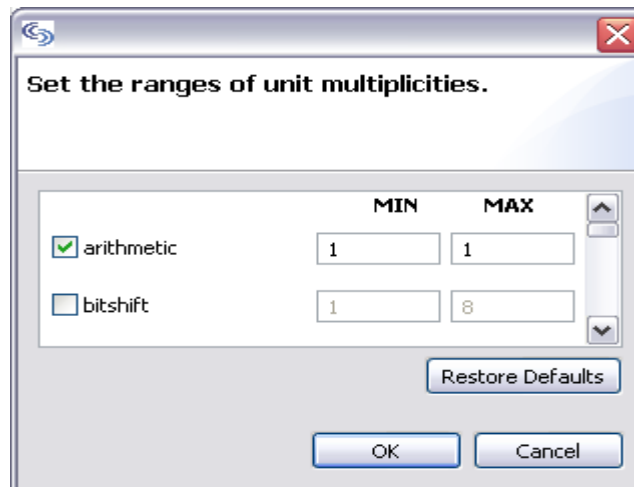


Figure 4: Capping FU multiplicities

The growth of the coprocessor can be further restricted by placing limits on the number of instances allowed for each type of functional unit in the architecture. By capping the number of units to 1 of each type Cascade is limited to only optimizing the connectivity between them.

Data Cache Selection

One of the main ways in which the user can reduce the total coprocessor area is in the Data Cache configuration. By default Cascade evaluates two types of Data Cache- Associative Cache and Remapping Cache, and displays the results as a graph as shown in Figure 5.

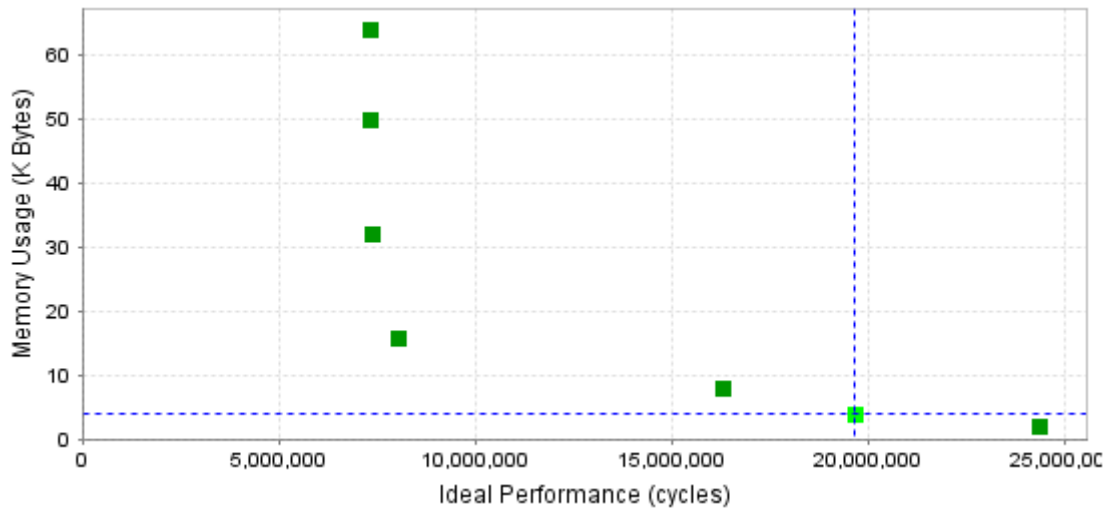


Figure 5: Data Cache size vs. performance

The Remapping Cache must be large enough to contain the entire runtime memory footprint for the software on the coprocessor. It is clearly too large in this case at around 32KB.

The Associative Cache operates as a standard cache and so can be reduced to any size allowing a trade off against the performance reductions due to the increasing miss rate as shown in Figure 5. Additionally, the amount of logic required for the associative cache also drops linearly with the cache size.

Candidate Low Area Architectures

Many candidates were generated allowing the user to select different optimization points should the project requirements change. This range is shown in Figure 6 where 32 MCycles on the testbench is equivalent to the 66 MHz target performance.

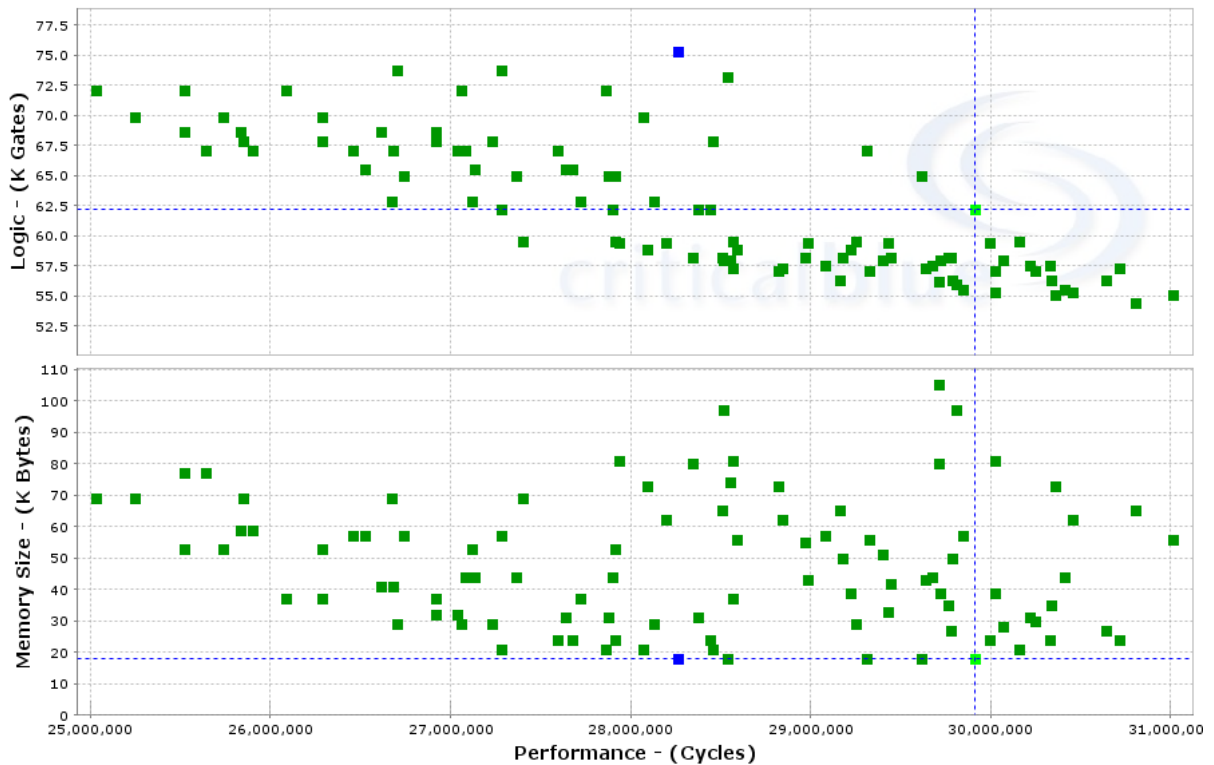


Figure 6: Candidate architectures for low area coprocessors

Maximizing Coprocessor Performance

In order to enhance the performance of a coprocessor the primary strategy is to enable as much parallel computation as possible across as much of the offloaded code base as possible. This may require software modifications to expose more instruction level parallelism in the implementation, if not already present, along with removing architectural bottlenecks in the coprocessor to ensure that there is enough computational resource available to run as many operations in parallel as possible.

Optimizing Bandwidth

In the case of an MP3 decoder the most computationally intensive areas of the code are in clearly defined loops and long sequences of data processing operations which can be easily parallelized by Cascade.

However, by extracting all of this parallelism Cascade quickly runs out of memory bandwidth to move data from the cache and through the necessary computations. This lack of available bandwidth is due to the default data cache configuration since it provides only a single Read/Write port interface to the functional units of the coprocessor. This results in computation becoming serialized on memory access operations reducing the available instruction level parallelism available with a dramatic impact on performance.

Fortunately, Cascade allows the user a great deal of flexibility in how the data cache is configured and it is very straightforward to add additional ports to the configuration via the cache generation dialog shown in Figure 7.

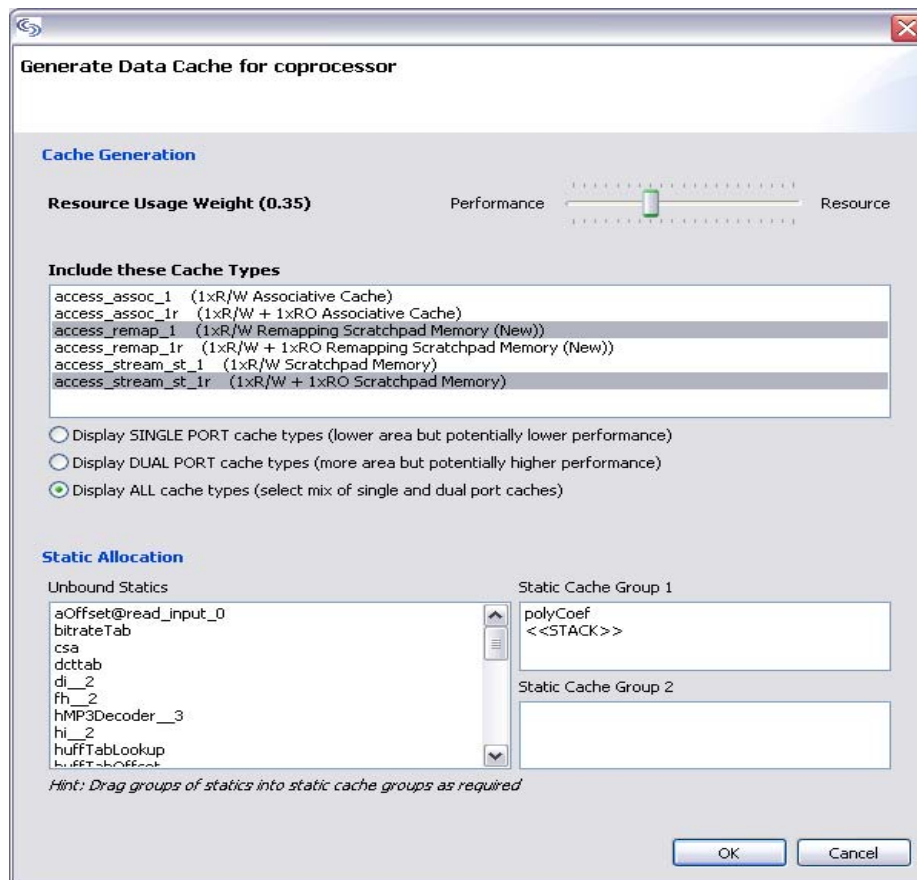


Figure 7: Configuring High Performance Memory

The most straightforward approach to increase bandwidth is to simply allow the use of dual port RAM as the underlying memory cell for the cache. Unfortunately this approach can result in a substantial area penalty when used on larger cache sizes which are necessary when trying to create a high performance coprocessor running large working sets such as for MP3.

The alternative is to use the analysis features of Cascade to identify the areas of the software which are being most significantly restricted due to the lack of available bandwidth. Typically there will be a few key lookup tables or data areas which are very heavily accessed during periods of intense computation.

These memory areas can then be mapped by name onto Static Caches which act as local scratch pad memories in addition to the main cache. By adding a second cache the bandwidth available to the regions of code using the mapped data is doubled. If the amount of storage required for these data areas is quite small it then becomes reasonable to use dual port RAM for these caches, further increasing the bandwidth.

In this case the coefficient table for the polyphase filter along with the stack were mapped to a 4K dual port static memory.

Low Latency Architecture

Cascade allows the user to take advantage of a low implementation frequency by making various architectural optimization options available in order to decrease the number of pipeline registers used in the construction of the coprocessor architecture.

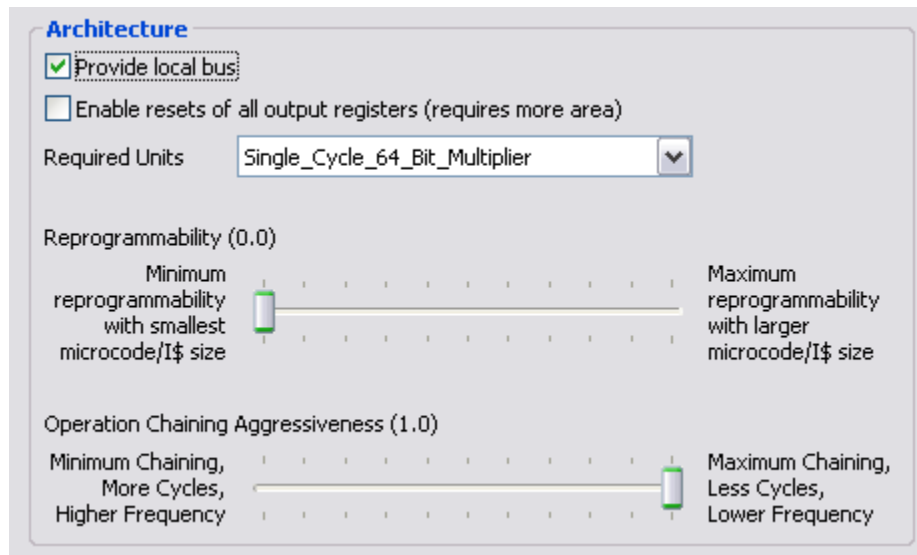


Figure 8: Low latency architecture constraints

In Figure 8, operation chaining is set to maximum which allows Cascade to synthesis a large number of *combinatorial* paths between functional units in the architecture. This allows operations on these units to be chained together so that many sequential operations may be executed in a single cycle. Since the MP3 Decoder often contains multiplication operations in its key functions, the single cycle multiplier unit is selected instead of the standard heavily pipelined multiplier.

By setting the reprogrammability option to zero, Cascade can highly compress the instruction word format, which means less instruction traffic on the bus as it allows a greater amount of the total code base to be cached on the coprocessor. The coprocessor is still fully reprogrammable, but the efficiency of the generated microcode for new functionality may not be as good if a more flexible instruction format was used.

Custom Functional Units

To achieve very high performance it is often necessary to turn to customized hardware blocks for small sections of functionality which can be implemented far more efficiently as hardware than as software. Cascade supports the integration of these hardware blocks natively at the software function call level. This approach means that the user can define a Custom Functional Unit (CFU) in Cascade to implement any of the offloaded functions as in Figure 10. Cascade will then synthesize coprocessor

architectures containing place holders for the CFU blocks which are fully integrated with the automatically generated hardware of the coprocessor (Figure 9).

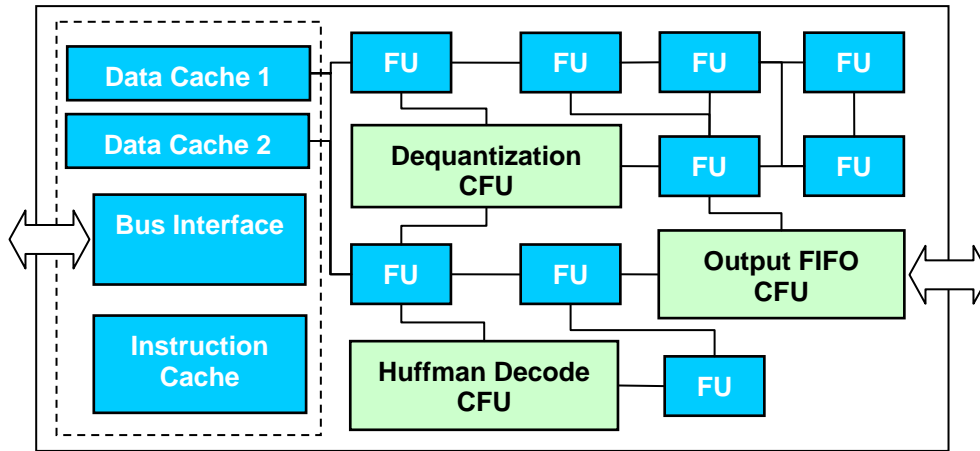


Figure 9: Coprocessor Architecture with Custom Units

Whenever a function mapped to a CFU is called in the software running on the coprocessor, the data normally passed as function parameters is directed to the relevant ports of the hardware block, and the resulting data is then read back without changing the behavior of the surrounding software.

Since Cascade does not require the RTL implementation of the CFU to be available until after the RTL for the coprocessor is generated, it is straightforward to take a what-if approach to trying out different CFUs. This allows the user to establish that sufficient performance gain will result from the design of a CFU, before having to design it.

In the MP3 case, CFUs are implemented to perform the following functionality:

- Some simple but runtime intensive operations in the kernel of the main Dequantization loop with the software handling the loop logic and data movement.
- The Huffman decode function. This is commonly implemented as a dedicated block in otherwise programmable decompression accelerators.
- An output FIFO for the decoded samples.

Selection of the sections of code to replace with CFUs was guided by feedback from Cascade. The Hotspot Spot viewer feature allows the user to identify the areas of the offloaded code in which most of the runtime is being spent. Next the user can experiment with replacing various code sequences or entire functions in these hotspots with CFUs and evaluate the impact on performance before deciding on the CFUs to actually implement in RTL.

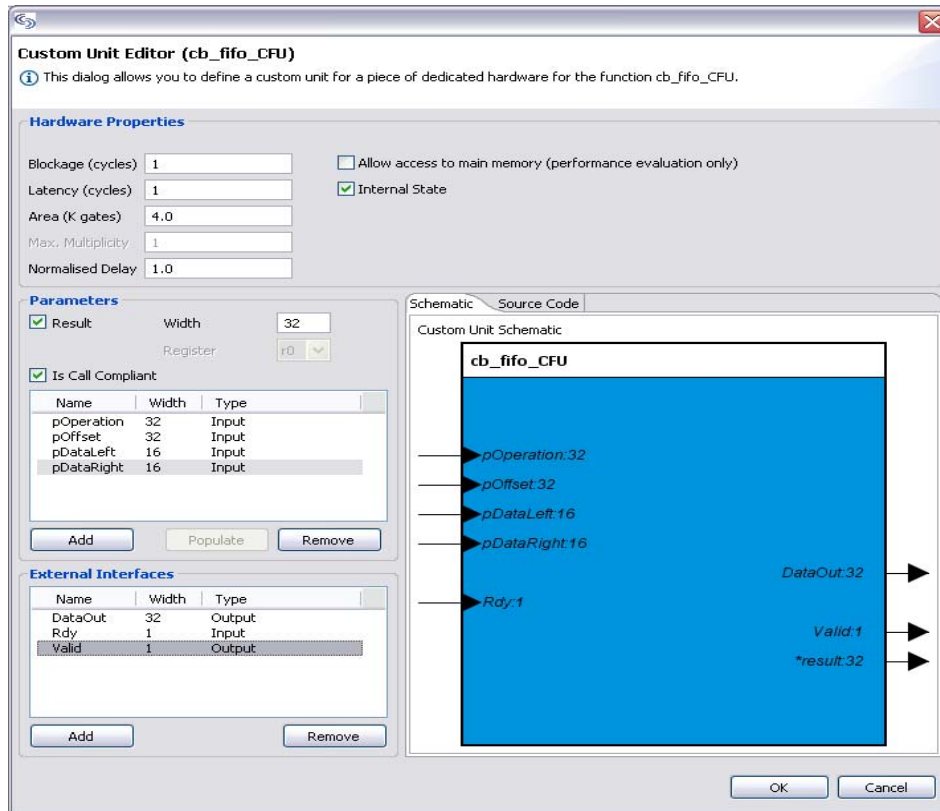


Figure 10: Defining the Output FIFO Custom Unit

The output FIFO CFU differs from the others as it introduces an asynchronous component into the coprocessor. The FIFO is filled by the sample written out by the Polyphase filter function. Once full, the FIFO then writes the samples out asynchronously to the rest of the software which can carry on processing the next batch of samples. This removes the bottleneck of waiting to write a frame full of samples to the audio driver peripheral before starting to decode of the next frame.

To enable this asynchronous behavior, the FIFO CFU takes advantage of the coprocessor External Interface feature. These external interfaces allow the user to connect a CFU to an interface they define (Figure 10) on the top level of the coprocessor generated by Cascade. Taking advantage of this, the FIFO CFU is connected directly to the Audio Driver peripheral, and this removes substantial traffic from the system bus as shown in Figure 11.

If Cascade is used to map all the audio processing onto a coprocessor then the coprocessor and an audio codec device could easily be combined into a single subsystem.

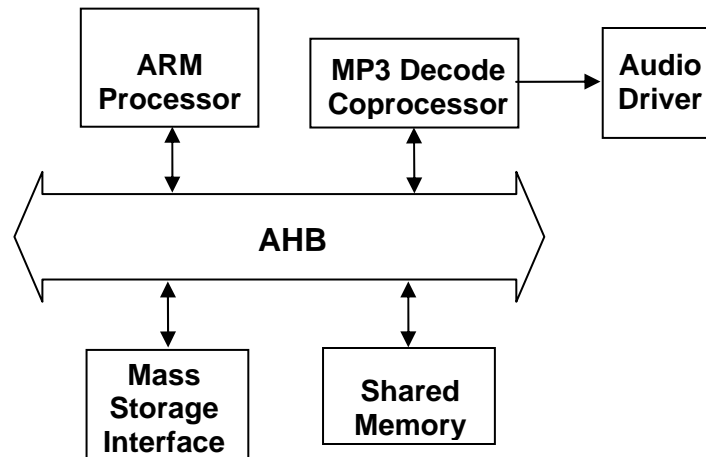


Figure 11: Coprocessor interfaced Audio Driver

Candidate High Performance Architectures

As for the low area case, Cascade generates a wide range of potential architectures based upon the constraints it has been given. Figure 12 shows the range generated at 10% effort level. For this testbench, 3.4 MCycles corresponds to a 7MHz real time decode frequency.

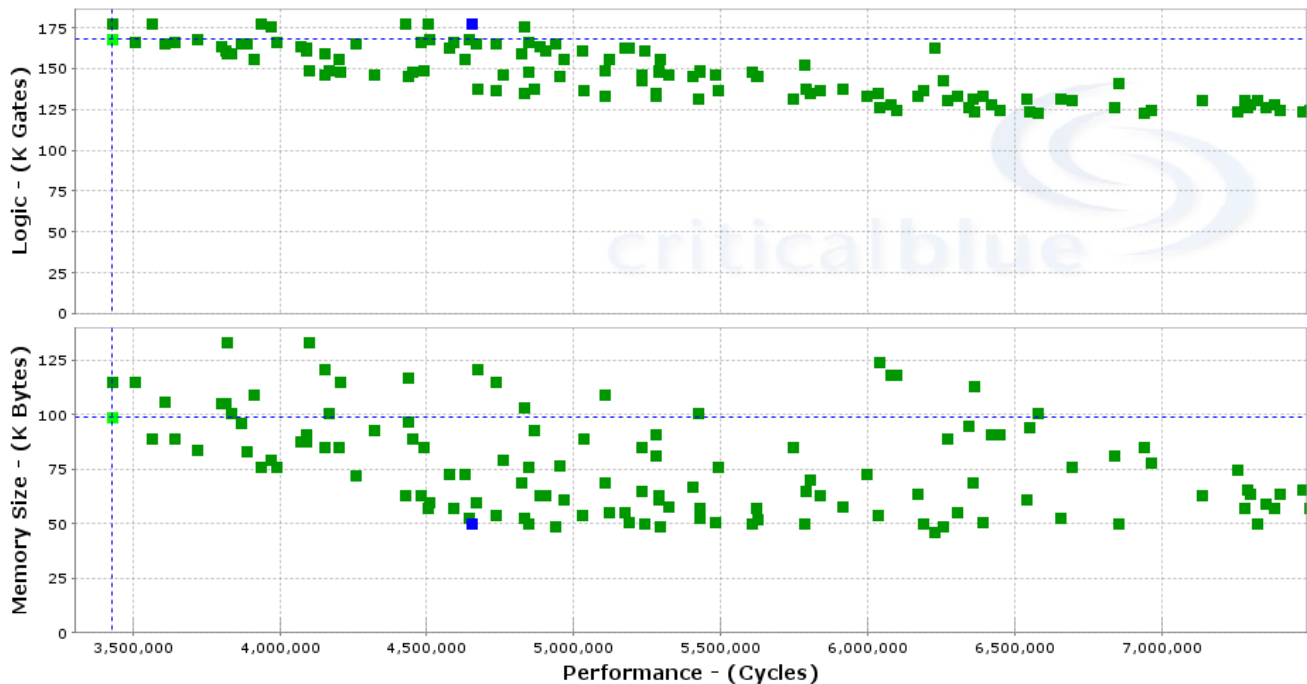


Figure 12: Candidate architectures for high performance coprocessors

Conclusions

This paper introduced a number of techniques for driving Cascade to rapidly generate coprocessors given different target design requirements. Each user and each project will have unique engineering situations to address, and this is why Cascade is such a useful solution. The results for the example of an MP3 decoder are summarized in the table below along with the figures for an ARM9 processor for comparison purposes.

Device	Frequency for Real Time Decode	Single Port RAM	Dual Port RAM	Logic	Approx Logic Area (TSMC 90nm)
ARM946E-S	27.6 MHz	32 KB	NA	NA	0.30 mm ²
Area Optimized	62.0 MHz	17 KB	0.8 KB	62 K	0.15 mm ²
Performance Optimized	7.0 MHz	94 KB	5 KB	168 K	0.41 mm ²
Performance Optimized (Small I Cache)	9.8 MHz	45KB	5 KB	177 K	0.43 mm ²

The techniques presented are applicable to a wide range of applications particularly in the multimedia, DSP and security domains. Additionally, the resulting coprocessor is still running the same fundamental algorithm as the ARM processor. Therefore improvements to the ARM runtime due to software enhancements are likely to be reflected in the performance of the coprocessor as well as providing more opportunities for area reduction.