cadence®

Optimizing Tensilica Processors for SSD Controllers

Solid-state drive (SSD) controller architects face a number of design challenges that require them to balance performance and energy to achieve design goals. This white paper discusses different design approaches and strategic tradeoffs that help optimize a design for specific applications. Cadence® Tensilica® processors offer an ideal alternative to the traditional solutions, combining an RTL-like performance and energy profile with ease of use and programmability to create an optimal blend of risk, performance, cost, and energy consumption.

Contents

Introduction

All SSD controllers perform the same basic function of managing host system requests for data storage and retrieval using NAND flash devices (e.g., accessing data on your PC or tablet). SSD controller architectures vary, focusing on issues such as latency, throughput, longevity, energy used, and cost. For example, designs for enterprise-class SSDs may favor throughput, called IOPS (I/Os per second), while designs for client-class SSDs may favor low cost and energy. Designers generally innovate their SSD controller designs by adding architectural and algorithmic uniqueness that addresses the specifics of their markets.

Differentiation at the RTL level can provide the lowest energy consumption and highest performance, although this approach is inflexible for last-minute design changes and adds risk from longer development and verification time. Using off-the-shelf processors instead of custom RTL can solve the risk problem, but often cannot meet the performance and energy requirements.

These issues can be solved using optimizable Cadence Tensilica processors, which provide RTL-like performance and energy consumption as well as design flexibility with full programmability and advanced development tools.

This white paper highlights common design challenges that SSD controller architects face and how they can solve them by using optimized Tensilica processors. A basic knowledge of SSD controller architectures is assumed.

HDD to SSD Migration

A host system can use a hard disk drive (HDD) or SSD to store data centrally within the system.

HDDs, which have been used for decades, consist of one or more rotating disks (platters) coated with magnetic material to store data. A motorized moving arm reads and writes data to and from a specific disk. Data blocks can be accessed randomly. HDDs are non-volatile memory that retain data even when powered off.

In contrast, SSDs use integrated circuit assemblies as memory to store data, so have no mechanical moving components. SSDs use flash memory (typically NAND), which retains data even when powered off. This approach is more expensive, but yields a number of advantages over HDDs, including higher data transfer rates, better reliability, and significantly lower latency and access times.

A host system has a map of where it stores data on each device within the system. Historically, the host expected the storage device to be an HDD and the cylinder-head-sector (CHS) addressing was used to access the physical media. When SSDs were introduced as HDD replacements, they had to perform address translations to be compatible with—and therefore be transparent to—the host system. Today, logical block addresses (LBAs) refer to the location of the data in the SSD itself and a translation from the host's address into this new LBA is required for each read or write operation.

In contrast to most memory storage types, NAND flash is much cheaper due to its higher bit density, but is read and written one page (4 to 32Kbyte) at a time, and the pages wear out with use. As the pages wear out, the data read back becomes corrupted, and eventually the controller's error correction logic can no longer correct the data—making that page unusable.

This cheaper flash memory needs to be managed so that the flash pages wear out evenly over time (generically called wear leveling). Without wear leveling, a repeated update of a few files can cause high wear in just a few pages, quickly making them unusable and rendering the whole device useless if the host is not able to avoid those locations.

For most users, modern flash management approaches eliminate these wear concerns over the lifetime of an SSD. For example, the PC Worldtest (2014) of six SSDs shows the first failure would take over 700 years with one terabyte of writes being made per year.[1]

SSD Controller Architecture

SSD controllers perform the basic function of managing host system requests for data storage and retrieval using NAND flash devices. A typical SSD controller consists of a host interface control block, a flash translation layer (FTL) control block, and a NAND control block. See Figure 1.

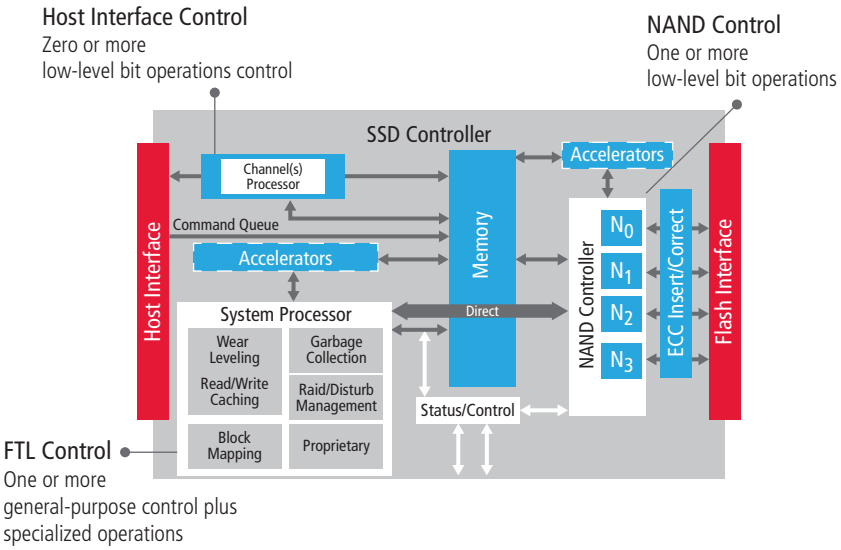


Figure 1: SSD controller block diagram

Host interface control block

The host system performs SSD accesses, making read and write requests for data. These requests are supplied as a series of commands in the interface protocol. The electrical interfaces used vary from ATA, SAS, FiberChannel, PCIe® and DDR (for NVDIMM) while the protocols used across these interfaces for the commands are usually NVMe, SATA, or SCSI.

Multiple commands can execute simultaneously, enabling the SSD controller to optimize the completion order (often favoring reads over writes) to hide or reduce latency and increase throughput. The host interface protocol usually specifies how many outstanding commands it supports. For example, SATA has one queue of up to 32 commands, and NVMe has up to 65,536 queues of up to 65,536 commands. The SSD architecture is designed to handle a maximum number of commands. For example, lower-end SSD architectures may handle only a few commands at once while higher-end SSD architectures may handle a few hundred.

Traditionally, the host interface logic is written in RTL, as it requires fast, low-latency responses and implements a single fixed protocol. However, to support the recent NVMe protocol standard and other standards, some architects are now adding one or more processors for flexibility. These processors do not need to process the actual data—they are used to decode and prioritize the host commands.

Flash translation layer (FTL) control block

The SSD controller's FTL control block performs most of the main SSD functions, handling all of the host command actions and managing the NAND flash to maximize endurance.

Handling host commands involves mapping the logical block address (LBA) of the data from the host to the physical block address (PBA) of the data in the flash, then setting up the movement of data between the host interface and the NAND controller for reads and writes. Performing this task quickly increases the throughput of the SSD—usually measured in input/output operations per second (IOPS). The SSD architecture is usually aimed at optimizing the design for reads, writes, or a balance of the two, with innovative techniques for command re-ordering, hiding data latency, compressing data, and more.

NAND flash management functions often use sophisticated, proprietary, wear-leveling techniques.

Management functions include:

- Bad block mapping—taking corrupt blocks out of play
- Garbage collection—preparing invalid pages for immediate writing
- Read disturb—monitoring flash cells for potential corruption from nearby reads
- Read/write caching—keeping data in RAM while active, writing to flash only when needed
- Encryption—encoding data before storing onto the flash

One or more processors are typically used to perform these functions, providing the flexibility to experiment quickly with new design implementations and make changes right up to and after silicon fabrication to support continual product improvement and keep pace with evolving standards. Some SSD controllers also enable customized algorithms targeted at specific data patterns, typically for SSD drives targeting in-house databases.

NAND control block

It can take up to 75us to read a flash page, and up to 1.3ms to write one.[2] This limits the total throughput of the device unless operations can occur in parallel. Note that each flash chip can only perform one operation at a time.

Several flash chips are typically connected to the controller via a standard bus such as ONFi or Toggle. These buses allow access to one flash chip at a time, so for higher throughput devices, designers add multiple buses or channels so that more than one flash chip can be accessed in parallel.

When multiple channels are in operation, it is critical to optimize the host command traffic so that all channels remain as active as possible; the optimal flow depends on the access length and type. The logic required for these decisions has historically been written in RTL as it is fairly simple and predictable. However, many architects are adding programmability for more sophisticated performance and lifetime extension optimizations.

Transferring and Storing Data

The process for data transfer and storage varies for host data, management data, and DDR data within the SSD controller blocks.

Host data

Transferring data between the host and flash storage typically requires a straightforward DMA transfer once the NAND channel is ready to stream the read data from the flash to the host, or the write data from the host into the flash. Typically, data is not buffered on the SSD unless it needs processing.

While some level of buffering is required on the SSD if data is processed for functions such as compression, encryption, and malware detection, it is minimized by processing the data as close to real-time as possible using RTL blocks and accelerators.

Management data

To manage the NAND flash, a limited history needs to be recorded and used to assess the next page that should be written, the pages that may need read-disturb correction, and the pages that need garbage collection. This additional management data also needs to be stored (see "Managing and Processing Data" for more information).

For devices with 32Kbyte pages, this means tracking over 30 million pages per terabyte. With more sophisticated algorithms, 128 bits of management data per page can be stored. In addition, some architectures hold the recently used hot data cached in RAM for faster access and to reduce wearing. As a result, the amount of data that gets stored just to manage the flash can be huge in itself. To reduce cost, the management data is often stored in cheaper off-chip DDR memory.

DDR data

While DDR memory is cheaper, it has a long access latency. To hide as much of that latency as possible, SSD controller architects try to issue multiple DDR requests and continue to perform operations while the DDR requests are being fulfilled. Even though the DDR bandwidth fundamentally limits the achievable performance, the latency and the number of outstanding accesses that can be handled form the practical limit for any SSD controller architecture. Architects are building accelerators to increase the number of outstanding accesses as the limits imposed by processors and their bus architectures prove inadequate.

Managing and Processing Data

Many tables or data lists are maintained to manage NAND flash, such as mapping logical to physical blocks, the free block list, the wear status, the read-disturb status, and others.

Multiple types of information with different numbers of bits are often packed into 32-, 64-, or even 128-bit words that need to be read, unpacked, and then perhaps modified, re-packed, and written back. Fetching the data 32 bits at a time can take multiple reads for the larger sizes, and then unpacking a specific quantity requires shifting and masking for each value. Fetching and processing wider data reduces the cycle time and latency of these operations. Where this is a significant bottleneck, designers often create an accelerator logic block to handle this processing with only a few cycles of latency.

Tables require indexing to find the correct data. For very large amounts of data, multi-level tables are often used for faster access to data in the first level that is most often used and is stored on-chip. Typically, the entire FTL table structure is nested as shown in Figure 2.

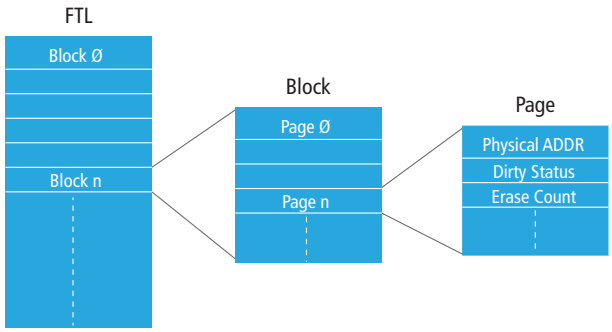


Figure 2: FTL nested table structure diagram

The smallest element can be a structure that represents a page's information, such as physical address, used and dirty status, erase count, and more. An array of these page structures and other variables form each block's data structure. Arrays of these block structures then cover the entire device.

Lists are implemented as linked lists where each link in the chain is examined for a match to a key that identifies it as the item being searched for. Linked lists are more compact than tables since they can grow and shrink as needed, and can easily be rearranged if their order needs to be updated, but they are searched serially giving an unpredictable search time. Linked lists of structures are often used to hold bad pages, cold pages, and hot pages where the key is the page address. For long chains, hashed link lists are used to reduce the average and worst-case number of cycles needed to find the right entry—the chain is split up into many chains with the chain selection determined by a hash of the search key. (A hash table uses a hashing function to compute an index into an array of slots where the desired value is located.)

For increased throughput, the SSD controller designer needs to implement an architecture that reduces cycle counts and hides/ minimizes latency overheads for managing and processing the data.

Assessing SSD Controller Design Priorities and Development Options

Your SSD controller design will have specific goals in terms of performance and energy, requiring you to consider different factors and approaches when planning your development path to achieve these goals. For example, you can run the processor faster, architect the design to reduce cycles, or offload specific tasks to RTL. Additionally, you need to prioritize and assess what strategic tradeoffs to make, such as lower development risk, lower BOM cost, lower development cost, or faster time-to-market. The product's development stage can also affect design priorities, as new designs are inherently risker than a second- or third-generation product that requires enhancing an existing design with incremental changes.

Table 1 shows the typical development options available for achieving a specific product priority.

	Development Option						
Product Priorities	Run Processor Faster	Reduce Cycles	Offload to RTL	Build More In-House	Use More Third-Party IP	Use Better Development Tools	Add Flexibility
Higher throughput (IOPS)							
Lower energy							
Lower development risk							
Lower BOM cost							
Lower development cost							
Faster time-to-market							

Recommended Not recommended Mixed result Not relevant

Table 1: SSD controller design product priorities

When creating SSD controllers, you must rank these priorities and choose the development options that best fit with your design goals. Development options include:

- **Run processor faster**—Turn up the clock—may need higher speed memories, more complex interconnects, additional cooling, and board work to support
- **Reduce cycles taken**—Change the processor (ISA) or software implementation to take fewer cycles for the same algorithm
- **Offload to RTL**—Create custom RTL that helps implement the algorithms in fewer cycles
- **Build more in-house**—Develop more of the product with internal resources—may need to hire and train
- **Use more third-party IP**—Buy pre-designed blocks rather than develop them in-house—more capital expenses
- **Better development tools**—Improve development efficiency with tools that reduce hardware and software creation, verification, and debug times
- **Add flexibility**—Make the product more flexible so that it can be easily updated to address development issues and market changes

For example, a basic low-end SSD controller may have design goals that include lowering development risk, BOM cost, and development cost, and improving time-to-market. If you aggregate these priorities as shown in Table 2 (row 1), for this SSD controller design, using more third-party IP and better development tools is an optimum development choice, creating less risk and lowering development costs. Other options such as in-house development and offloading to RTL can be too expensive and risky, and running a processor faster increases BOM and development costs.

Table 2 shows a similar analysis for mid- and high-end SSD controller designs, with possible design priorities and the recommended development options for four case examples. These SSD controller designs show a few trends:

- Cycle count reduction is always beneficial
- Offloading to RTL gives substantial benefits, if significantly increased development risk and cost are acceptable
- Adding flexibility can significantly reduce development risk, if some increased cost is acceptable

	Development Option						
SSD Controller Design Priorities	Run Processor Faster	Reduce Cycles	Offload to RTL	Build More In-House	Use More Third-Party IP	Use Better Development Tools	Add Flexibility
Basic low-end: • Lower development risk • Lower BOM cost • Lower development cost • Faster time-to-market							
Mid-/high-end Case 1: • Higher throughput • Lower energy							
Mid-/high-end Case 2: • Higher throughput • Lower energy • Lower development risk							
Mid-/high-end Case 3: • Higher throughput • Lower energy • Lower development risk • Lower BOM cost							
Mid-/high-end Case 4: • Higher throughput • Lower energy • Lower development risk • Lower BOM cost • Lower development cost							

Recommended Not recommended Mixed result Not relevant

Table 2: Recommended development options for SSD controllers

Creating a High-Performance, Low-Energy SSD Controller Design

When prioritizing for both performance and energy consumption, offloading to RTL and reducing processor cycles are the only development options usually considered.

Offloading to RTL

Most processors do not provide an easy way to improve the instruction set or add more bandwidth, so the ability to reduce cycles is limited to software optimization. That means that offloading to RTL is the only viable option to increase performance and keep the energy consumption low. This has the unfortunate downside of increasing development risk and cost due to the additional verification cycles required and the possibility of a re-spin if errors are not caught. This approach is also less flexible and adds some communications latency overhead to/from the RTL. For this approach to be a worthwhile option, the RTL must make a significant difference to the cycle counts.

Reducing processor cycle counts

After software optimization, reducing the cycles required for a task requires new processor instructions that perform the function of multiple instructions in one, and if possible, execute instructions in parallel. Using a profiling tool to analyze the operations can illustrate where most cycles are being spent and help you evaluate how to perform the function with fewer instructions—usually guided by what operations can be performed in an RTL implementation. By adding or increasing the superscalar or very long instruction word (VLIW) capabilities of a processor, more instructions can be executed at the same time.

Once new, more efficient instructions and extra parallelism are identified, the processor, models, and software tool chain (e.g., instruction set simulator, compiler, and debug tools) needs to be updated and verified. Typically, you need the source RTL (Verilog) for the processor design.

The process of adding instructions and parallelism to processor RTL requires deep knowledge of the processor, significant RTL development and verification time, and significant software development time. Alternatively, most designers will simply choose the RTL offload option instead, as that does not require additional processor and tools expertise—although it does have the downside of reducing flexibility due to less programmability.

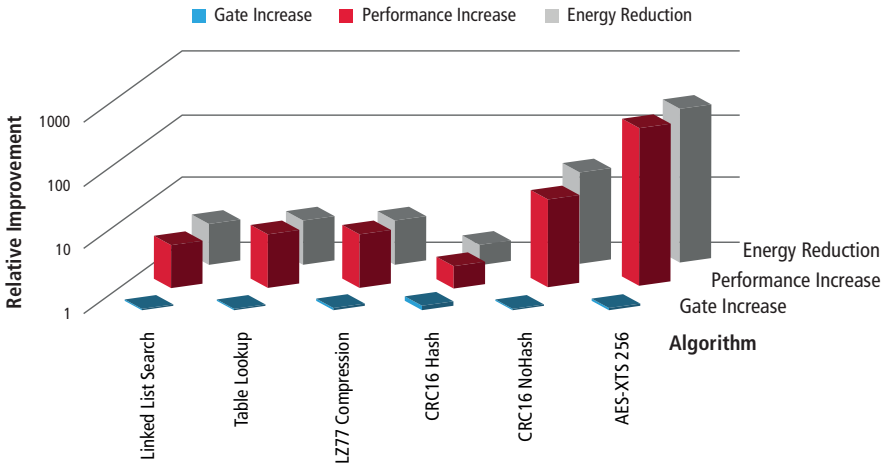
Reducing cycle counts using Tensilica processors

There is an alternative that gives designers an RTL-like performance and energy profiles while retaining programmability, and does not require deep processor expertise or source code. Designers can create processors that are ideally suited to their architecture using the Tensilica® Optimization Platform. Here, new instructions, instruction parallelism, internal storage, and new interfaces can be added to achieve not only much higher processing throughput, but also much higher memory bandwidth. Using a simple Verilog-like language to describe these additions the processor hardware (RTL) is produced along with all development tools automatically. Detailed processor design expertise is not required, as the tools automatically generate the software development tool chain. Using an Eclipse-based integrated development environment (IDE), called Xtensa® Xplorer, the instruction-set simulator (ISS), SystemC models, C/C++ compiler, assembler, and debugger needed for any SoC project are all automatically generated.

The cycle count reduction you can achieve with Tensilica processors is mostly limited by what is physically possible within any RTL. For a specific application you can choose to design just to the performance level you need, or optimize further to perform more demanding functions.

Designing SSD Controllers Using Tensilica Processors

Tensilica processors can be created to accelerate many popular functions used in SSD controllers, such as those listed in Figure 3. The instructions created to optimize these functions do not show the maximum performance possible—instead they are points in a continuum that show the additional gate count required to reach a certain performance level. With a very small increase in gates, we can gain a tremendous performance increase and reduce overall energy. For example, with the AES-XTS 256 algorithm, an 11% gate increase resulted in a 265X performance gain and a 239X overall energy reduction compared to a conventional RISC processor.



	Linked List Search	Table Lookup	LZ77 Compression	CRC16 Hash	CRC16 NoHash	AES-XTS 256
Gate Count Increase	2%	2%	9%	15%	2%	11%
Performance Increase	4X	6X	6X	2X	24X	265X
Energy Reduction	4X	6X	5X	2X	24X	239X

Figure 3: Relative improvement using a Tensilica processor versus a conventional RISC processor

To build an SSD controller, you can choose from multiple Tensilica processor templates as a starting point—these are competitive RISC controllers with a range of performance and size with the same base instruction set.

After selecting a template, you can enhance the processor further by choosing from numerous checkbox options such as execution units, processor states, memories and memory interfaces, floating-point unit, caches, ECC, general-purpose I/O (GPIO) ports, FLIX (a form of VLIW), and single instruction, multiple data (SIMD). Additionally, you can differentiate your SSD architecture by creating proprietary optimizations for the processor, such as adding new instructions and custom I/O that are specific to your design and data structures.

Designing the host interface control block

RTL implementations have limited built-in flexibility, so processors are ideal when multiple protocols need to be supported and where standards are still evolving—as long as the processor offers sufficient performance.

A single Tensilica processor provides both the needed programmability and low latency for prioritization and control of the data path, plus it enables you to investigate other algorithms during design optimization.

For host interface control processing, choose a minimal Tensilica processor template then add custom I/O for direct connection to the interfaces and FTL block. These processors often include custom instructions that allow many operations and I/O accesses to occur in parallel to make them very similar to hardwired logic.

Tensilica processors can add GPIO for status/control and FIFO interfaces that can be used for a number of tasks. For example, it is very convenient to use FIFOs for command queues, DMA interfaces, or even passing data to/from other hardware engines in the system. Since they can be very wide (up to 1,024 bits each), it is easy to achieve performance similar to that of pure RTL, while retaining the programmability and debug features of software.

Designing the FTL control block

FTL processing requires numerous data lookups for the logical to physical address mapping. Other throughput optimization data is also tracked, such as which data is often used (hot) and seldom used (cold) so that hot data can be cached in local memory rather than written to flash each time. Additionally, data from many other proprietary optimization algorithms is also tracked.

Because the tracking data scales with the size of the flash storage—and can be huge—it often requires large amounts of external storage in longer-latency DDR memory that has a more limited bandwidth. Consequently, the SSD architecture needs to be optimized to hide the DDR latency as much as possible, and to more efficiently use the available bandwidth.

Designers often use multi-level tables or linked lists to split the data into more manageable chunks.

Using multi-level table lookups

For our example, the multi-level table approach uses two tiers of tables to hold data. The small T1 table may hold hot data for immediate return, or index into one of many larger T2 tables that hold other data. If the data is not in table T1, then the index returned is used to perform a second table lookup to get the data, as shown in Figure 4. This approach can yield a large latency if the T2 tables are in DDR memory, upwards of two-hundred processor cycles. The key to hiding this latency is to simultaneously perform multiple table lookups by minimizing the number of cycles between sequential lookups.

In this implementation, the Tensilica processor implements several instructions that reduce the latency for the Table Lookup algorithm by a factor of 6X for tables in local memory (see Figure 3).

See the Accelerating IP Address Lookup and Multiple Tables Access with Xtensa Application Note for examples on using multi-level table lookups. While the examples described in this application note are for looking up IP addresses, the same approach can be used for address lookups in SSD applications.

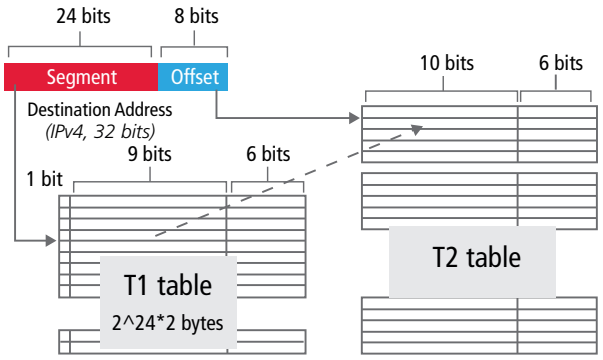


Figure 4: Multi-level tables

Using linked lists

Linked lists enable data lookup via a series of lists, linked in a chain that are traversed in sequence. Each link's reference key is compared to the search key until a match is found. The worst-case latency occurs when all of the links in the chain are traversed. Linked lists are generally used when the amount of data being stored for a particular reference is unpredictable or dynamic (e.g., the bad page list, hot page list) or has sequencing that needs to be updatable dynamically because they consume less space than a table that has to be designed for the worst-case storage. To reduce latency, linked lists can be made bidirectional (at the cost of some additional storage) so that a search in both directions can be interleaved—with a new worst-case latency of only half of the chain length.

To further reduce latency, you can split one chain into multiple shorter chains, using a hash of the search key to determine which chain to start with. If the hash function can be performed more quickly than traversing the list, using this approach reduces the average and worst-case latency. The hashing function requires multiple bit manipulations and acceleration logic for quick processing. See Figure 5.

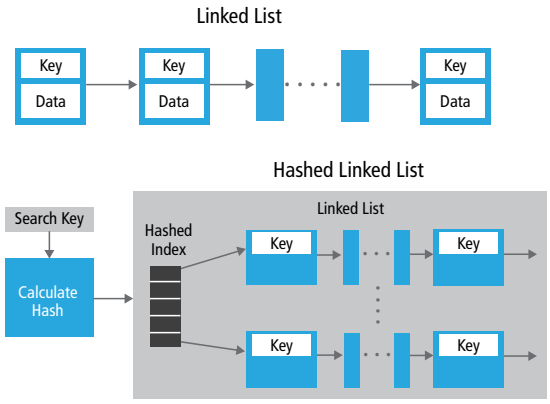


Figure 5: Linked list diagram

In this example, the Tensilica processor implements several instructions that reduce latency for the Linked List Search algorithm by a factor of 4X, to a fixed three cycles-per-link for data stored in local memory (see Figure 1). For hashed linked lists, a secure hash algorithm (SHA) based hash function can be executed in a single cycle to bring the cycle count down by a factor of ~8X.

See the Accelerating Hash and Linked List Computation with Xtensa Processors Application Note for additional details.

Hiding latency

To hide long memory latencies, use multiple outstanding read accesses (assuming that write accesses do not add latency) so that a new read request does not need to wait for a previous read to complete before it can be issued. Ideally, a new read request can be made every cycle to keep the total latency to that of a single access. For latencies of 100 cycles, this can mean having up to 100 outstanding reads—a feature that is not supported by most processors.

Designers often create their own lookup engines so that they can add as many outstanding accesses as they need, and then push requests to it from the processor and return later to get the results. Performing these accesses over the system bus adds more latency and often requires multiple cycles to read the wide data that gets loaded.

You can add dedicated point-to-point interfaces to Tensilica processors to make interfacing with custom lookup logic predictable and more efficient. See the Mitigating Long Latency Memory Accesses with Xtensa Processors Application Note for details about using these interfaces as well as the benefits available when using cache prefetch and DMA.

For FTL control processing, use a higher performance Tensilica processor template with VLIW capability for issuing multiple operations at once to boost throughput of the more general-purpose application code. Custom instructions are typically added to accelerate some tasks that are not efficiently executed by general-purpose controllers, such as table lookups and linked-list operations.

Designing the NAND control block

The NAND control block sequences reads and writes to most efficiently use available bandwidth across all channels. This requires channel prioritization based on command queue, the current accesses, and their likely completion times. Using a processor to gain flexibility allows the algorithms to be more dynamic, but the prioritization can take many cycles as each channel's status has to be polled before it can be analyzed.

Tensilica processors with new instructions that allow channel status polling can be performed in a single cycle. Part of the prioritization logic can be added to the polling instruction so the algorithm can run quickly on pre-processed data, allowing the processor to behave much like an RTL logic block—but with more flexibility to change the algorithm and parameters for the specific flash vendor and to compensate for changes as the flash devices age.

For NAND control, choose a minimal Tensilica processor template then add custom I/O for direct connection to the interfaces and FTL block. These processors often include custom instructions that allow many operations and I/O accesses to occur in parallel to make them similar to hardwired logic.

Tensilica processors can add GPIO for status/control and FIFO interfaces that can be used for a number of tasks. For example, it is convenient to use FIFOs for passing commands directly to/from the FTL controller. GPIO can be used to read all of the channel priorities at the same time. Since these interfaces can be very wide (up to 1,024 bits each), it is easy to achieve performance similar to that of pure RTL, while retaining the programmability and debug features of software.

Benefits of Using Tensilica Processors in SSD Controller Designs

Using one or more optimized processors for different functions in an SSD controller design can make processing extremely efficient, significantly reducing cycle count and simultaneously lowering energy consumption. With virtually unlimited I/O, there is no need to compromise the ideal design due to system bus limitations. Directly connect to other parts of the SoC with wide buses for low and predictable latencies.

Improve performance and reduce energy from 2X to over 250X depending on the functions being run (refer to Figure 2). What you would usually design in offload RTL can now be added in a matter of days as one or more new instructions in a Tensilica processor—with all the development tools being created automatically in lock-step with any changes made.

Accessing processor source code is not required

Using a Verilog-like language called Tensilica Instruction Extensions (TIE), the functionality of new instructions can be described at a high level of abstraction, while still controlling the number of cycles the computations may take. No integration or awareness of the processor RTL is required—the TIE description is automatically merged with the other blocks when the updated processor is generated.

Processor expertise not required

You need an understanding of functional RTL design, but not about how to create processors. A knowledge of Verilog is helpful to write TIE, but not required. After writing new instructions in TIE, the TIE files are simply attached to the project in the Xtensa Xplorer IDE, and then the processor is rebuilt in a few minutes.

Development tools created automatically

Using the TIE description, all of the tools—including compiler, debugger, profiler, ISS and SystemC models—are automatically generated in minutes, so iterating to the ideal solution is quick. When you are ready for layout, the processor RTL and synthesis scripts are all produced in about an hour.

Conclusion

If your products require more performance and lower energy consumption, adding a faster processor or more-of-the-same processors is not the answer. Increasing efficiency by doing more in parallel and moving data around with wide, direct connections is the only option—you need to offload to RTL or create a better processor.

Offloading to RTL takes more development time and adds risk to product completion timescales, and is not as flexible as a programmable solution. On the other hand, creating a better processor retains flexibility, but can be more risky than creating offload RTL due to the design expertise needed and the huge investment required to develop and maintain the software development tools.

Cadence Tensilica processors offer the ideal solution, with a highly automated, sophisticated, and robust mechanism for creating processors optimized for any application. Designers can easily create a processor with RTL-like performance, reduced energy consumption, and virtually unlimited I/O–with full programmability and all development tools created automatically.

Additional Information

For more information on the unique abilities and features of Cadence Tensilica processors, see http://ip.cadence.com/ipportfolio/tensilica-ip

References

1. PCWorld Contributor: Paul, Ian. (2014, Dec. 5). Grueling Endurance Test Blows Away SSD Durability Fears http://www.pcworld.com/article/2856052/grueling-endurance-test-blows-away-ssd-durability-fears.html

2. Vatto, Kristian. (2012, Oct. 8). Samsung SSD 840 (250GB) Review http://www.anandtech.com/show/6337/samsung-ssd-840-250gb-review/2