cādence®

Virtually Unlimited I/O Bandwidth

More Performance and Less Energy from Direct, Wide Connections

Two big bottlenecks facing high-speed block design for ASICs and systems on chip (SoCs) are I/O performance and computational performance. A processor's main system bus represents a significant I/O bottleneck, as system-wide data passing into and out of the processor must travel over this bus. Cadence® Tensilica® Xtensa® processors can relieve system bus congestion by allowing the processor to bypass the bus and move more data through direct, wide connections significantly faster than conventional 32-bit RISC processors. These I/O features allow Xtensa processors to achieve data-transfer rates that can match those of hand-designed RTL blocks. Additional computational performance improvement can be achieved with Xtensa processors by performing more operations in a single instruction, and by executing multiple independent instructions in parallel.

Contents

Introduction 1

The Tyranny of One Bus....................2

Breaking the I/O Bottleneck with Faster Local Direct Interfaces 3

Attaining the Ultimate Bandwidth.....3

Boost Throughput Further with Multiple Computations per Cycle 5

Achieving Lower Energy Consumption...................................6

Conclusion 6

Additional Information 6

Introduction

When designing high-speed blocks for ASICs and SoCs, two factors constrain I/O performance:

- The system bus can only perform one transfer at a time so other pending transfers must wait for the current transfer to clear
- Because the system bus is designed to accommodate many system configurations, multiple cycles are required to complete bus transactions

As a result of these limitations, processors lack the I/O bandwidth required for many SoC tasks. In this white paper, we discuss how to increase I/O performance with the Xtensa processor by creating direct connection interfaces that give single- and fixed-cycle throughput, bypassing the system bus and eliminating bus overheads.

Xtensa processors can achieve high I/O data-transfer rates because of a variety of features: direct connections to local memory (including cache, ROM, and RAM), Ports (general-purpose I/O), Queue (FIFO) interfaces, and memory Lookup interfaces. The Ports, Queues, and Lookups allow designers to add many new input and output connections directly into and out of the processor's execution unit. These custom interfaces can be described using the Tensilica Instruction Extension (TIE) language, which allows designers to extend a processor's functionality using a Verilog-like syntax—without the need to describe the structure of the hardware. Hardware structure is generated automatically by the processor generator and is guaranteed correct-by-construction.

Using these features to maximize I/O bandwidth, ASIC and SoC designers can implement high-speed, processor-based I/O with transfer rates that are similar to those achieved by hand-coded RTL function blocks but with much less time and effort. In addition, as it is firmware-programmable, the block's function can easily be changed at a later date to accommodate updated industry standards, new features, and bug fixes in the system design without changing the silicon.

To increase computational performance, you can define a single TIE instruction that executes multiple independent operations in parallel by using the Xtensa Flexible Length Instruction Extensions (FLIX) technology—a form of VLIW with variable slot widths, without the code bloat issue normally associated with VLIW.

The Tyranny of One Bus

Figure 1 shows an embedded processor configuration typically found in SoC designs. The sole data highway into and out of the processor to other parts of the design is its system bus. Because processors often interact with other types of bus masters, including other processors and DMA controllers, these system buses have sophisticated transaction protocols and arbitration mechanisms for sharing the bus in a multi-master system environment. These mechanisms result in bus transactions that occur over several clock cycles.

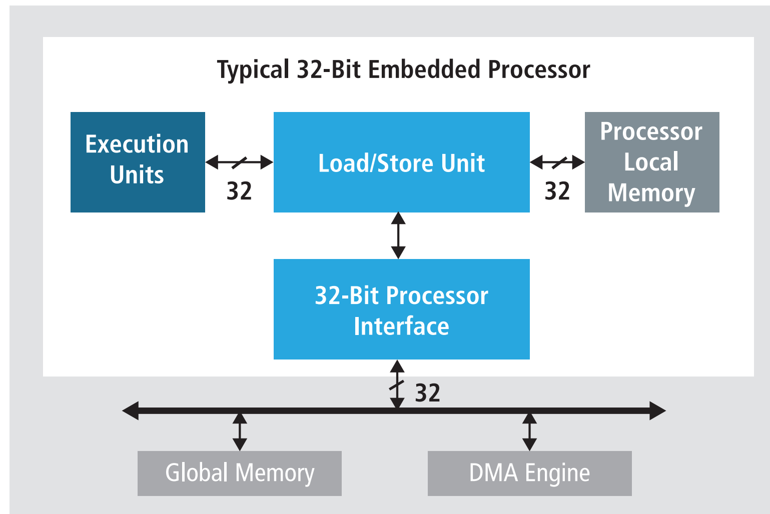

Figure 1: The Load/Store Bottleneck

For example, the Xtensa processor's "read" transactions over its system bus take a minimum of 6 cycle latency and "write" transactions over the system bus take at least 1 cycle, depending on the speed of the target device. From these transaction times, we can calculate the minimum number of cycles needed to perform a simple flow-through computation—where the processor loads two numbers from global memory, adds them together, and stores the result back to memory. The assembly code to perform this computation can look like this:

```
L32I reg _ A, Addr _ A                    ; Load the first operand
L32I reg _ B, Addr _ B                    ; Load the second operand
ADD  reg _ C, reg _ A, reg _ B            ; Add the two operands
S32I reg _ C, Addr _ C                    ; Store the result
```

To simplify this code, we assume that pointers to the memory locations containing values A, B, and C are already initialized in registers Addr_A, Addr_B, and Addr_C. If not, then more time will be needed for this computation.

The minimum cycle count required to perform this computation is:

```
L32I reg _ A, Addr _ A:                   6 cycles
L32I reg _ B, Addr _ B:                   6 cycles
ADD reg _ C, reg _ A, reg _ B:            1 cycle
S32I reg _ C, Addr _ C:                   1 cycle

Total:                                    14 cycles
```

Note that this cycle count total is a minimum number. Because the Xtensa processor is pipelined, the total number of cycles will be slightly greater than 14, but the additional cycles will overlap with the execution of other instructions. If this code sequence sits within a zero-overhead loop, the cycle count for each loop iteration is 14 cycles.

Load instructions (L32I) each consume 6 cycles, which is the minimum number of cycles required to return the requested information over the processor's main bus. Load operations must complete before the next instruction that uses the resulting data executes, so the ADD instruction must wait until the two preceding load instructions are complete. Also, the store instruction (S32I) consumes only one cycle because the stored value is immediately placed in the processor's store buffer, at which time the store instruction completes. The processor's bus control logic subsequently moves the stored value to the target location, but code execution can proceed while the processor's bus logic moves the data from the store buffer to system memory.

The large number of cycles required for high-speed data to flow through a processor-based function block, such as the simple example shown above, is often a major bottleneck that generally is resolved by designing an application-specific block of RTL.

Breaking the I/O Bottleneck with Faster Local Direct Interfaces

One way to improve the system bus bandwidth for processor-based I/O is to conduct the load and store transactions over faster local memory interfaces. For example, Xtensa processors have optional local-memory interfaces for cache, RAM, and ROM. For local transactions, the Xtensa processor performs load and store operations in a single cycle. By conducting loads and stores over the local memory interfaces, computation timing improves:

```
L32I reg_A, Addr_A:                  1 cycle
L32I reg_B, Addr_B:                  1 cycle
ADD reg_C, reg_A, reg_B:             1 cycle
S32I reg_C, Addr_C:                  1 cycle

Total:                               4 cycles
```

This result represents a 3.5X improvement in the function's cycle count, which may mean the difference between acceptable and unacceptable performance for a particular task. However, even with the performance improvement gained from faster bus transactions, the local interfaces still conduct only one transaction at a time, so loads and stores must occur serially.

Attaining the Ultimate Bandwidth

Even the improved bandwidth achieved with single-cycle loads and stores over a fast local interface can be too slow for certain on-chip tasks. They are increasingly inadequate for high-throughput applications such as video compression/ decompression or high-speed networking.

The Xtensa processor lets you directly connect to external memories and RTL blocks without using the system bus or the local memory interfaces that perform transfers using the load/store unit. Xtensa processors provide more flexibility, offering the ability to create as many new interfaces such as Ports (GPIO), Queues (FIFO), and memory Lookup interfaces with customizable widths as you need. These interfaces do not use the load/store units for data transfers.

For 32-bit wide interfaces, you can select from two checkbox configuration options: GPIO32 and QIF32.

For example, when you select the GPIO32 configuration option, it adds two 32-wire Ports to the Xtensa processor—one for input, one for output—to quickly control and monitor peripherals or other logic in the system. When you select the QIF32 configuration option, it adds two 32-bit Queue interfaces for FIFO-like data streaming into and out of the processor. The input queue functions with a familiar pop/empty/data interface to external logic while the output queue presents a similar push/full/data interface. All interactions with the Xtensa processor pipeline are automatically implemented when the option is selected. These interfaces are accessed as registers in the processor, so no separate load/store is required to operate on the data.

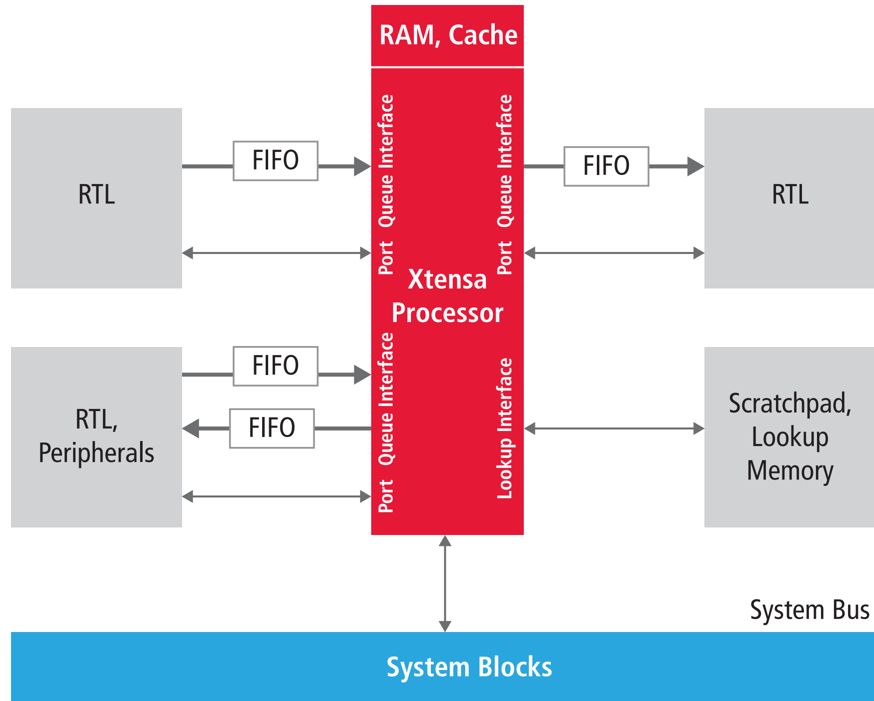

Figure 2: Ports, Queue and Lookup Interfaces Allow Direct Data Transfers

These TIE interfaces are based on tried-and-true system technology that provides your design team with as much bandwidth as any system can possibly use, virtually unlimited I/O bandwidth. This is much like the I/O capabilities of RTL design.

For example, Ports (GPIO), Queue (FIFO) interfaces, and memory Lookup interfaces allow ASIC and SoC designers to add multiple I/O interfaces to the processor. These interfaces are simple, direct communication structures. I/O transactions execute over Ports and Queues in a single cycle, and over Lookups with a fixed latency. Transactions conducted over these interfaces are not accessed by processor-supplied addresses, but by executing custom processor instructions that implicitly initiate the transactions. One instruction can initiate transactions on several interfaces at the same time to boost I/O bandwidth per cycle by two or three orders of magnitude.

See the *High-Speed Alternatives for Inter-Processor Communications on SoCs* and *Reduce Development Risk and Add Programmability with RTL-Like Performance and Connectivity* white papers for additional information.

Ports are associated with custom registers in the processor. The processor can read the value of input Ports simply by reading the state of the associated register. Similarly, the processor can put a certain value on an output Port simply by writing that value to the associated custom register in the processor. This technique is simple and fast.

Queue interfaces are designed to directly attach to FIFOs. The processor can read/write a Queue simply by reading/writing a value from/to the custom register associated with the Queue interface. Hardware built into the Queue interface shields the system designer from speculative operations that can occur in all RISC processors.

For the example problem discussed earlier, three Queue interfaces need to be created in the TIE processor-customization language: two input Queues for the input operands and one more for the result output Queue. With these three Queue interfaces defined, it is possible to define a custom addition instruction that:

- Implicitly pops input operands A and B from their respective input Queue interfaces
- Adds values A and B together
- Outputs the result of the addition operation (C) to the output Queue interface

The following TIE code defines two 32-bit input Queues (InQ_A and InQ_B) and one 32-bit output Queue (OutQ_C). The bus width in this example is 32 bits but it can be any width up to 1,024 bits just by adjusting the queue definitions.

```
queue InQ _ A 32 in
queue InQ _ B 32 in
queue OutQ _ C 32 out
```

Each statement causes the automated Xtensa processor generator (XPG) to add an interface to the processor. Each interface has the handshake signals needed to connect to a FIFO memory that is external to the processor. A custom instruction defined in TIE can implicitly read from one or more input Queues and write to one or more output Queues during one processor clock cycle. Input Queue interfaces look like any other input operand to a custom instruction and output Queues can be assigned values just like any other output operand.

The following example defines a custom TIE instruction called ADD_XFER that reads operands from the two input Queue interfaces, adds the values together obtained from the queues, and writes the result to an output Queue interface.

```
operation ADD _ XFER {in AR val}
{in InQ _ A, in InQ _ B, out OutQ _ C}
{
        assign OutQ _ C = InQ _ A + InQ _ B;
}
```

With this new instruction, the example problem reduces to the execution of one custom instruction:

```
ADD _ XFER
```

This instruction takes five cycles to run through the Xtensa processor's 5-stage pipeline. However, by placing this instruction within a zero-overhead loop, the processor can deliver an effective throughput of one ADD_XFER instruction per clock cycle. Then all data movement and the associated computation occur in the absolute minimum number of clock cycles—namely one. Even hand-coded RTL accelerators cannot perform this function any faster. Plus, the processor has the advantage of being firmware-programmable.

As another example, TIE Lookup interfaces can connect one or more memories (e.g., SRAM, ROM) directly to the Xtensa processor, and a single TIE instruction can perform a transaction on all interfaces at once. These Lookup transactions also occur independently of local and system bus transfers. The designer can define the address and data width size (up to 1,024 bits in total) so that all the data is transferred in a single transfer instead of over multiple cycles.

Boost Throughput Further with Multiple Computations per Cycle

The Xtensa processor is not limited to performing one computation per cycle. The TIE language provides two ways to perform two or more calculations at a time.

- You can create a custom, single-cycle TIE instruction that draws operands from several input Queues, performs multiple operations on these operands concurrently, and then outputs the results on several output Queues.
- Alternatively, you can use the Xtensa processor's FLIX technology to develop wide, multi-operation instructions that execute concurrently, i.e., with FLIX you can execute multiple independent operations in parallel, a form of VLIW with variable slot widths.

The advantage of the first approach is simplicity. If all the computations are always performed concurrently in the same manner, then they can be combined in one custom instruction. TIE lets you combine multiple operations into a single instruction that can be executed every cycle.

The FLIX approach offers you more flexibility. If some of the operations are only performed some of the time or if the operations are performed in various combinations, FLIX instructions provide the ability to easily create many combinations of concurrent operations. In some applications, either method works equally well. In other applications, the flexibility of custom FLIX instructions can make function-block development much easier.

See the white papers *High-Speed Alternatives for Inter-Processor Communications on SoCs*, *Reduce Development Risk and Add Programmability with RTL-Like Performance and Connectivity,* and *Increasing SoC Processor Computational Performance with More Instruction Parallelism* for additional information.

Achieving Lower Energy Consumption

By customizing the processor, designers can get a significant performance improvement per clock cycle while reducing the overall total energy consumption for an application. With custom I/O, what may have taken multiple operations to fetch wide data can now be done in a single operation.

Too often, designers fixate on a static "milliWatts per megahertz" (mW/MHz) number while ignoring the total energy consumption of the workload. There is an implicit assumption in this type of thinking that all RISC processors deliver about the same performance per clock cycle. This assumption does not apply to processors that have been customized for a specific task or application.

For example, adding a few custom instructions increases the processor core's size, which in turn increases the processor's average power dissipation per clock cycle (increases the mW/MHz rating). However, if the new instructions dramatically reduce the total clock cycles required to perform a given workload (the target C code application), then the total energy consumed (power-per-cycle multiplied by total cycle time) can be substantially reduced.

Example: A 20% increase in power dissipated per clock cycle, offset by a 3X speed up in task execution, actually reduces energy consumption by 60%. This reduction in required task-execution cycles allows the system either to spend much more time in a low-power sleep state, or to reduce the processor's clock frequency and operating voltage, leading to further reductions in both dynamic and leakage power.

See *The What, Why, and How of Customizable Processors* white paper for additional information.

Conclusion

Processors can be flexible and versatile, but one of their chief liabilities has been an inability to achieve I/O transfer rates equal to that of RTL accelerators. The chief bottleneck has been the conventional microprocessor bus. However, Xtensa processors offer I/O options that can directly connect to other parts of the system—avoiding the bus bottleneck altogether—and provide better flexibility than hand-coded RTL blocks through firmware programmability while delivering equivalent throughput. These characteristics make custom processors attractive alternatives when designing functional blocks for ASICs and SoCs.

With Xtensa processors, you can increase computational performance using a variety of methods, such as defining TIE instructions that perform multiple operations within a single instruction, or executing multiple independent instructions in parallel using FLIX. By customizing the Xtensa processor, you can optimize it to run your application more efficiently, significantly reducing cycle counts, improving performance and reducing energy consumption.

Additional Information

For more information on the unique abilities and features of the Cadence Tensilica Xtensa processors, see ip.cadence.com.

Cadence Design Systems enables global electronic design innovation and plays an essential role in the creation of today's electronics. Customers use Cadence software, hardware, IP, and expertise to design and verify today's mobile, cloud and connectivity applications. www.cadence.com

© 2015 Cadence Design Systems, Inc. All rights reserved worldwide. Cadence, the Cadence logo, Tensilica, and Xtensa are registered trademarks of Cadence Design Systems, Inc. in the United States and other countries. All other trademarks are the property of their respective owners.
5241 09/15 SC/DM/PDF