

# Increasing Processor Computational Performance with More Instruction Parallelism

While most software is written with the assumption that processors execute a task's operations one at a time in sequence, some processors can simultaneously perform multiple operations without functionality changes. Executing multiple operations at a time reduces a task's total cycle count at the expense of additional logic. The amount of parallelism that can be achieved depends on the task itself and the ability to detect what can safely be executed at the same time. The Cadence<sup>®</sup> Tensilica<sup>®</sup> Xtensa<sup>®</sup> processor uses a flexible very long instruction word (VLIW) approach that allows the designer to choose the amount of parallelism they want to add to the processor, using an advanced C/C++ compiler to automatically bundle and schedule operations to create the optimal tradeoff between performance and area.

## Contents

Introduction .....	1
Long Instructions.....	2
Xtensa Processor—A Primer on RISC and VLIW Architectures.....	2
FLIX and the Xtensa Processor Generator (XPG).....	3
A FLIX Example—Issuing Multiple Operations in Parallel to Improve Application Performance .....	4
Defining a FLIX Instruction Format ....	4
Interleaving FLIX Instructions.....	5
Defining Custom Operations and Slot Sizes .....	5
Conclusion .....	6
Additional Information .....	6

## Introduction

ASIC and SoC designers routinely turn to hand-coded RTL acceleration hardware to compensate for the lack of performance they experience with general-purpose, fixed-instruction-set architecture (ISA) processors—a tried-and-true design approach. However, as ASICs and SoCs get larger and larger, the job of verifying all of these custom accelerator blocks becomes more cumbersome, and the effort needed to verify this hand-coded hardware dominates most projects.

By allowing processors to take on more of the computing load for ASICs and SoCs, designers avoid long RTL verification time and add flexibility via programmability to the chip's design. Programmability enables design modifications to address product specification changes late in the design process or fix last-minute algorithm bugs. Modifications can be made by a simpler firmware change without revising the chip's hardware design and re-running verification. In addition, programmability allows more features to be added at a later date, which can extend the life of the hardware design.

To get more computing performance from a processor, i.e., high throughput per clock cycle of instruction processing, some processor IP vendors offer superscalar or VLIW architectures, which execute (or "issue") multiple independent operations in a single cycle. These multi-issue architectures are larger than the single-issue versions due to the duplication of functional blocks, extra register-file ports, and other logic that makes simultaneous execution possible.

Superscalar is typically used for general-purpose control code processing. The processor uses specially designed hardware to analyze the code in real time and determine which instructions to execute in parallel, then routes the instructions to the appropriate functional blocks for processing. Superscalar processors do not require any pre-processing to determine what to issue at the

same time, so any scalar code targeting that processor will achieve the benefits of instruction-level parallelism from this “dynamic scheduling” approach. This means that superscalar processors can only extract limited parallelism from the code, and require significant silicon overhead.

VLIW is commonly used for DSP algorithm processing, where the compiler finds, bundles, and schedules multiple operations for parallel execution (static scheduling). The primary advantage of VLIW processors is the ability to offload the choice of what is executed to the software development process—reducing hardware cost and increasing the amount of parallelism that can be achieved. By detecting parallelism before run-time, the entire application can be analyzed to seek hard-to-find cases in the dynamic scheduling of superscalar architectures. Historically, VLIW processors used long instruction words that led to larger code footprints—thus increasing memory costs—but modern VLIW processors avoid this. The performance and flexibility benefits of long instruction words can be substantial, particularly for data-intensive applications with high inherent parallelism.

In this white paper, we discuss how the Xtensa processor improves performance and cost for both control and data-intensive processing with its flexible VLIW architecture.

## Long Instructions

RISC architectures generally encode one primitive operation per instruction. Long- instruction-word and VLIW architectures encode many independent operations per instruction, each in its own slot, with separate operand specifiers. Operations can be generic instructions similar to RISC instructions, or they can each be more sophisticated operations such as custom task-specific instructions designed to run a particular algorithm more efficiently.

A processor’s performance relates to the number of useful operations than can be executed per unit of time (operations per clock cycle and clock speed). Turning up the clock speed will yield a limited performance increase before requiring a deeper pipeline architecture and the addition of compensating logic area, such as branch prediction. Executing more operations in parallel to increase performance is limited by the inherent algorithm and the compiler capabilities. In addition, high performance does not guarantee flexibility. Instruction-set flexibility relates to the wider diversity of applications whose computations can be efficiently encoded in the instruction stream. A longer instruction word generally allows more operations and operand specifiers to be encoded in each instruction word.

One historical liability of long-instruction-word processor architectures is large code size, compared to architectures that encode one independent operation per instruction. This problem, frequently called “code bloat,” was common in VLIW architectures, and it is a critical design consideration for ASIC and SoC designs because instruction memories often consume a significant fraction of the total chip’s silicon area. Compared to code compiled for code-efficient architectures, VLIW code could require two to five times more code storage.

VLIW code bloat stemmed in part from instruction-length inflexibility. For example, if the VLIW compiler’s scheduler finds that only one operation can be executed, it will be forced to encode several operation fields in the VLIW instruction as “no operation” (NOP).

Modern architectures such as the Xtensa processor avoid VLIW code bloat by using more flexible instruction lengths. If the processor accommodates multiple instruction lengths, including short instructions that encode single operations, the compiler can achieve a significantly smaller code footprint, which boosts instruction-storage efficiency compared to the earlier VLIW processor designs with fixed-length instruction words. Reducing code size for long-instruction-word processors also tends to decrease bus-bandwidth requirements because fewer instruction bits are fetched.

## Xtensa Processor—A Primer on RISC and VLIW Architectures

The Xtensa processor uses flexible length instruction extensions (FLIX) that freely intermix smaller instructions with multi-operation VLIW instructions. This architectural approach addresses the code size challenge by offering 16-bit to 128-bit instruction lengths, in multiples of 8 bits, as well as variable slot widths in the instruction definitions. By packing multiple operations into a wide instruction word, the FLIX technology allows designers to accelerate a broader class of embedded applications with ultra-flexible instruction sizes and slot assignments.

Figure 1 shows an example of a FLIX basic long instruction implementation of a 64-bit instruction word with three independent operation slots, each of which specifies an operation and its operands. The first operation (slot 0) has an opcode and four operand specifiers—two source registers, an immediate field, and one destination register. The second and third operations (slots 1 and 2) have an opcode and three operand specifiers—two source registers and one source/destination register. The two-bit format field on the left designates the length for the variable size instruction, e.g., 24-bit (default), 16-bit, or 64-bit.

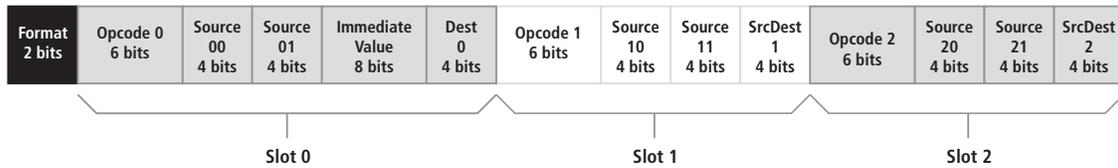


Figure 1: Example of FLIX Long-Instruction-Word Encoding

In some FLIX implementations, each operation’s resources can be almost completely independent of the other operation’s resources. Each operation can have dedicated execution units, dedicated register files, and dedicated data memories. In other FLIX implementations, operations can share common register files and data memories and require a number of ports for common storage structures such as register files to allow effective and efficient data sharing. The Xtensa processor allows the designer to define a FLIX implementation that is optimized for the application.

### FLIX and the Xtensa Processor Generator (XPG)

VLIW architectures sometimes need assembly coding to optimize best slot usage, where hand-packing independent operations into long instructions is a complex task. However, the Xtensa compiler generates code that is already aligned with operation-slot availability to maximize performance and minimize code size, and generally packs operations efficiently into long instructions, so no hand coding (assembly) is required.

As Xtensa processor hardware and tools are automatically generated, it is easy to accommodate custom FLIX architectures. High-level instruction descriptions can specify operations that fit into each slot. From these descriptions, the automated XPG determines encoding requirements for each field in each slot, assigns opcodes, and creates instruction-decoding hardware for all necessary instruction formats. At the same time, the XPG creates the corresponding compiler and other tool components as required. The advantages of this proven automated generation are that processor optimization and evaluation is quick, the processor does not need to be verified, and the software-development tools are available immediately.

The Xtensa base ISA instructions are either 16- or 24-bits wide. Through the Tensilica Instruction Extension (TIE) language, designers can define instructions of 24 bits up to 128 bits, in 8-bit increments. Instructions of different sizes can be freely intermixed without cycle penalties. The processor supports modeless intermixing of 16-bit, 24-bit, and wide instructions. Each instruction’s size is encoded into the instruction word itself. The Xtensa processor identifies, decodes, and executes any mix of instruction sizes from the incoming instruction stream. The wide instructions (instructions between 32 bits and 128 bits) can be divided into slots with independent operations placed in all or some of the slots. FLIX operation slots need not be equally sized. Any combination of the operations is allowed in each slot, and can occupy a single FLIX instruction word. All operations with a FLIX instruction are issued and executed in parallel.

## A FLIX Example—Issuing Multiple Operations in Parallel to Improve Application Performance

Figure 2 shows code for a simple C for-loop that is extracted from a program

```
for (i=0; i<n; i++)
    c[i]= (a[i]+b[i])>>2;
```

Figure 2: Simple C For-Loop

Compiling this code on the Xtensa processor without the FLIX implementation results in the disassembly code shown in Figure 3.

```
loop:
...
    addi a9, a9, 4;
    addi a11, a11, 4;
    l32i a8, a9, 0;
    l32i a10, a11, 0;
    add a12, a10, a8;
    srai a12, a12, 2 ;
    addi a13, a13, 4;
    s32i a12, a13, 0;
...
```

Figure 3: Disassembly of For-Loop Code Compiled Without FLIX

The compiler executes the for-loop code one instruction at a time without FLIX, thus taking eight cycles to execute.

Adding FLIX (using TIE) to your Xtensa processor improves the performance of your specific application by issuing multiple operations in parallel. With some additional hardware for parallel processing, a FLIX implementation can greatly reduce the overall execution cycles of your application program.

Compiling the same for-loop code on the Xtensa processor with a 64-bit FLIX implementation, we are able to bundle three operations in a single instruction, resulting in the disassembly code as shown in Figure 4.

```
loop:
{ addi ; add ; l32i }
{ addi ; srai ; l32i }
{ addi ; nop ; s32i }
```

Figure 4: Disassembly of For-Loop Code Compiled With FLIX

With FLIX, the compiler executes the for-loop code in three cycles, which is a tremendous performance improvement versus the eight cycles without FLIX. In this case, FLIX bundles a maximum of three operations in a single instruction word where possible for parallel processing. Note that a NOP (no operation) is inserted in a slot when an operation cannot be placed into a slot for parallel execution. The compiler can choose to use FLIX or not from one instruction to the next without penalty so that it does not need to insert NOPs each time.

## Defining a FLIX Instruction Format

Figure 5 shows a short but complete example of a simple multi-slot FLIX architecture description using TIE. This TIE description creates three operation slots in its 64-bit instruction word.

```
1: format myflix1 64 {slot_a, slot_b, slot_c}
2: slot_opcodes slot_a {L32I, S32I}
3: slot_opcodes slot_b {ADDI}
4: slot_opcodes slot_c {ADD, SRAI}
```

Figure 5: A Simple Multi-Slot Architecture Description

In the first line of Figure 5, the TIE *format* construct defines the *myflix1* instruction format with a width of 64 bits, and three instruction slots as *slot\_a*, *slot\_b*, and *slot\_c*.

In lines two, three, and four, the TIE *slot\_opcodes* construct defines a set of all operations that can be packed into each slot: *L32i*, and *S32i* in *slot\_a*, *ADDI* in *slot\_b*, and *ADD* and *SRAI* in *slot\_c*.

In this case, the instructions are existing Xtensa instructions. However, designers can also define new instructions and then assign them to these slots. A NOP is always created for each slot so the software development tools can always bundle three operations into a complete 64-bit instruction, even when no other operations for that slot are available for bundling due to resource-scheduling conflicts.

### Interleaving FLIX Instructions

The compiler automatically schedules and bundles FLIX instructions for maximum parallelism. Figure 6 shows execution of FLIX long instruction words within the loop and subsequent standard ISA instructions.

```

loop
  { l32i   ; add   ; addi  }
  { l32i   ; addi  ; srai  }
  { s32i   ; addi  ; nop   }
_loop_end:
add.n
srai
s32i.n
    
```

Figure 6: Interleaving of ISA and FLIX Instructions

### Defining Custom Operations and Slot Sizes

Long instructions allow more encoding freedom, where dozens of independent operation slots can be defined, although 3 to 10 independent slots are typical. The number of operation slots depends on the operational richness required in each slot. In addition, operation slots need not be equally sized as shown in Figure 7. Large slots (20 to 30 bits) can accommodate a wide variety of opcodes with greater than 2 or 3 register-operand specifiers addressing relatively deep register files (16 to 32 entries). ASIC and SoC developers should consider creating processors with wide operation slots for applications that exhibit modest degrees of parallelism but a strong need for flexibility and generality within the application domain.

Op A	Op B	Op C	Op D	Op E	Op F	Op G	Op H	Op I	Format Bits
------	------	------	------	------	------	------	------	------	-------------

Figure 7: FLIX Instruction Format with Custom Operations and Widths

Small operation slots (less than 16 bits) lend themselves to direct specification of movement among small register sets and allow a large number of independent slots to be packed into a long instruction word. Each of the smaller slots offers a more limited range of operations, shallower register files, and fewer explicit operand specifiers, perhaps increasing the need for more implied operands. ASIC and SoC developers should consider creating processors with many narrow slots for applications with a high degree of parallelism among many specialized function units. For example, when controlling multiple I/O ports and queues, it is advantageous to define several narrow slots consisting of only a few bits to encode a small number of the port and queue operations. Thus, a single long instruction word can execute a large number of simultaneous operations.

The FLIX format in Figure 7 shows the potential power of the Xtensa processor’s FLIX capabilities, offering instruction-level parallelism with the ability to independently specify unrelated operations within a single long instruction word. Moreover, all of the techniques associated with configurable and extendable processors that improve the performance of individual instructions—especially compound instructions and SIMD instructions—can be readily applied to the operations encoded in each operation slot.

For more information on FLIX options, see the *Xtensa LX Microprocessor Data Book*.

## Conclusion

The Xtensa processor uses flexible instruction lengths and slot widths with the FLIX technology. The processor naturally accommodates variable-length instructions, including short instructions that encode single operations and wide instructions with multiple operation slots, without penalty. The advanced compiler automatically bundles operations together and schedules them to achieve significant levels of parallelism across the whole application. Designers can accelerate a broader class of embedded applications for both control and DSP tasks using FLIX.

With this approach, designers can optimize the Xtensa processor for their exact requirements, testing out different configurations and options to see what provides the best combination of area, and performance, without touching the actual processor RTL. Instead, the TIE instructions are fed into the Xtensa Processor Generator and a complete new RTL description of the optimized processor is automatically created, along with all the matching software development tools. Our automated process ensures the new core will be correct-by-construction—we guarantee it.

## Additional Information

For more information on the unique abilities and features of Cadence Tensilica Xtensa processors, see [ip.cadence.com](http://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](http://www.cadence.com)