WHITE PAPER

How to Increase ASICs and SOC Computational Performance with Long-Word Processors

VLIW processors execute multiple independent instructions each clock cycle and provide a tremendous performance boost per clock cycle without incurring the exponential power-consumption increase caused by clock-rate increases. However, VLIW architectures have their own problems, particularly code bloat, which causes code footprints to balloon—thus increasing memory costs.

The Xtensa LX processor uses an innovative approach to VLIW design called FLIX (Flexible Length Instruction eXtensions), which gives ASIC and SOC designers more options for cost/performance tradeoffs. FLIX technology provides the flexibility to develop DPUs (Dataplane Processing Units) that freely and modelessly intermix smaller instructions with multi-operation FLIX instructions. By packing multiple operations into a wide 32-, 64-bit or 128-bit instruction word, FLIX technology allows ASIC and SOC designers to accelerate a broader class of embedded dataplane applications while eliminating the performance and code-size drawbacks of VLIW processor architectures.

One thing's certain in ASIC and SOC design—there's never enough performance to get everything done. 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-ISA (instruction-set architecture) processor cores. This is a tried-and-true design approach. However, as ASICs and SOC get larger and larger, the job of verifying all of these custom accelerator blocks has become very cumbersome and the effort needed to verify this hand-coded hardware now dominates most projects.

*Note: This Tensilica White Paper is based on the book **Engineering the Complex SOC** by Chris Rowen, published by Prentice Hall.*

Some processor-core IP vendors offer superscalar and VLIW architectures that deliver more computing performance per clock cycle, but these architectures also have their limitations. Superscalar processors can only extract so much parallelism from existing code (generally between 2x and 3x) at great hardware cost and VLIW processors use large instruction words that lead to code bloat, which causes code

footprints to balloon—thus increasing memory costs. Yet it's desirable to find a way to allow processors to take on more of the computing load on ASICs and SOCs because processors add programmability, hence flexibility, to the chip's design which can be used in several ways.

First, programmability can get the design team out of a jam if product specifications change at the last second (which they often do). Changes can be made without revising the chip's hardware design through a firmware change. Similarly, programmability can also fix that last-minute Algorithm bug—another frequent problem in chip design. Finally, programmability allows more features to be added at a later date, which can extend the life of the hardware design—often called a "mid-life kicker." Features implemented as custom accelerator logic are not changed as easily, so there are many benefits to a more processor-centric approach to ASIC and SOC design, if there's a way to get adequate performance from a firmware-programmable processor.

A Primer on RISC and VLIW Architectures

A processor's instruction-set performance relates to the number of useful operations than can be executed per unit of time or per clock. High performance does not guarantee good flexibility, however. Instruction-set flexibility relates to the wider diversity of different 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 word.

RISC architectures generally encode one primitive operation per instruction. Long-instruction-word and very-long-instruction-word architectures encode many independent operations per instruction, with separate operand specifiers for each independent operation. Operations can be primitive, generic instructions similar to RISC instructions or they can each be more sophisticated, application-specific operations such as the custom, task-specific instructions designed into a DPU (Dataplane Processing Unit).

Figure 1 shows an example of a basic long instruction. The figure shows 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 2-bit format field on the left designates this particular grouping of sub-instructions. It can also designate the overall length of the instruction if the processor supports variable-length encoding.

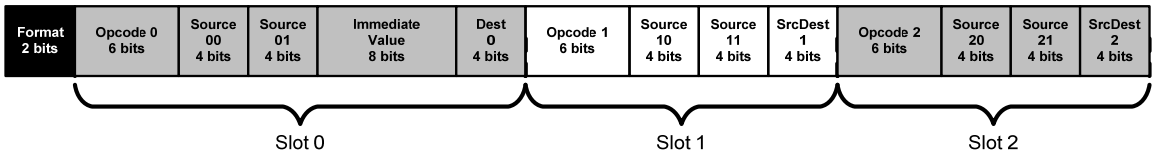

Figure 1: Example of Long-Instruction Word Encoding

Clearly there is a hardware cost associated with long instruction words. Instruction memory is wider, decode logic is bigger, and a larger number of execution units and register files (or register-file ports) must be implemented to deliver instruction parallelism. Large numbers of big hardware logic blocks are incrementally harder to optimize as a group, so the chip's maximum clock frequency can drop compared to simpler, narrower instruction encodings such as used for RISC processor designs. Nevertheless, the performance and flexibility benefits of long instruction words can be substantial, particularly for data-intensive applications with high inherent parallelism.

In some long-instruction-word architectures, 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 processor architectures, operations can share common register files and data memories and require a number of ports into common storage structures such as register files to allow effective and efficient data sharing.

The instruction length used in various long-instruction-word processor architectures varies widely. For high-end processors aimed at workstations and servers such as Intel's Itanium family and for high-end embedded processors such as Texas Instruments' TMS320C6400 DSP family, the instruction word is very "long" indeed—hundreds of bits. For embedded applications, which are always more cost- and power-sensitive, "long" might be just 64 bits. The essential processor architectural principles are largely the same, however, no matter the definition of "long." The essential characteristic is that multiple independent operations are packed into each instruction word.

Code Size and Long Instructions

One common liability of long-instruction-word processor architectures is large code size, compared to architectures that encode one independent operation per instruction. This is a common problem for VLIW architectures—it is frequently called "code bloat"—but 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 can require two to five times more code storage. Figure 2 compares the total code size of a VLIW DSP (TI TMS320C6203) with Tensilica's Xtensa

processor for the EEMBC Telecom suite. Note that the results for both processors reflect straight compilation from unmodified C source code and with optimized C code. No assembly code was used.

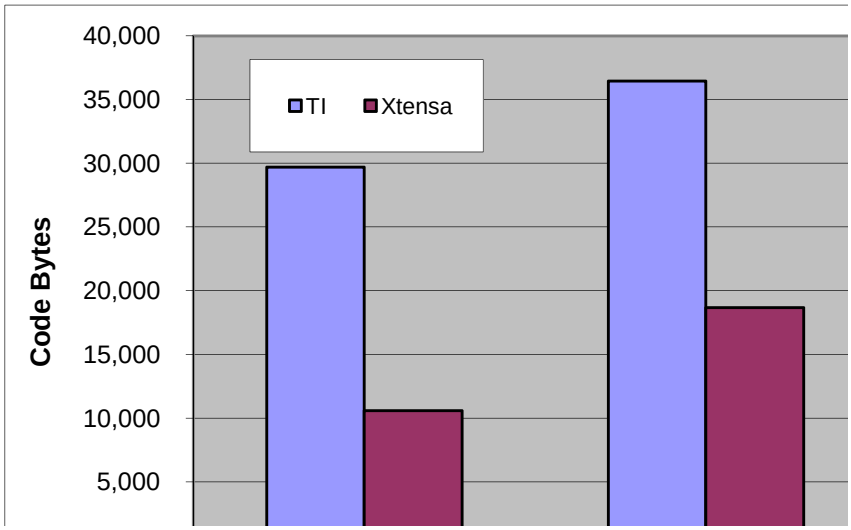

Figure 2: EEMBC Telecom Code Size Comparison

VLIW code bloat stems, in part, from instruction-length inflexibility. For example, if the VLIW compiler's scheduler finds only one operation whose source operands and execution units are ready, it will be forced to encode several operation fields in the VLIW instruction as NOPs (no operation). This mechanism is a key factor in VLIW code bloat. The cost of on-chip instruction storage is already a major portion of ASIC and SOC silicon area, so code expansion translates into higher cost, diminished cache performance, or both.

A second source of VLIW code bloat is the loose encoding of frequent operations commonly implemented in VLIW processors. The TI TMS320C6203 DSP, for example, requires 32 bits in the instruction word to specify a 16-bit multiplication and another 32 bits to specify a 16-bit add, so the common multiply/accumulate (MAC) operation requires at least 64 bits in the processor's instruction word. If a loop containing many MACs is unrolled four times (to amortize the cost of branch and address calculations), the resulting eight MAC operations require 512 bits of instruction storage, not counting the additional bits for associated loads, stores, branches, and address-calculation instructions.

However, long instructions need not lead to VLIW code bloat. For example, a long-instruction-word implementation of Tensilica's Vectra LX DSP architecture using Tensilica's flexible-length instruction extensions (FLIX) long-word technology uses about 20 bits within the instruction stream to specify eight 16-bit MACs executing in

SIMD fashion, not counting the additional bits needed for associated loads, stores, branches, or address-calculation instructions.

Thus one attractive method to avoid VLIW code bloat is to use 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 traditional 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) and therefore reduces the power dissipation associated with instruction fetches.

Tensilica's Xtensa LX processor incorporates the ability to use flexible-length instruction extensions (FLIX). This architectural approach addresses the code size challenge by offering 16-bit, 24-bit, and a choice of either 32-, 64- or 128-bit instruction lengths. Designer-defined instructions can use the 24-, 32-, 64- and 128-bit instruction formats.

Long instructions allow more encoding freedom, where dozens of independent operation slots can be defined, although three to ten 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 above in Figure 1. Large slots (20-30 bits) can accommodate a wide variety of opcodes, relatively deep register files (16-32 entries), and three or four register-operand specifiers. ASIC and SOC developers should consider creating processors with large operation slots for applications that exhibit modest degrees of parallelism but a strong need for flexibility and generality within the application domain.
Small operation slots (8-16 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, fewer explicit operand specifiers (perhaps increasing the need for more implied operands), and shallower register files. ASIC and SOC developers should consider creating processors with many small slots for applications with a high degree parallelism among many specialized function units.

Long Instruction Words and Automatic Processor Generation

Automatic generation of processor hardware and software can easily accommodate long-instruction-word architectures. High-level instruction descriptions can specify operations that fit into each slot. From these descriptions, an automated processor generator can determine encoding requirements for each field in each slot, assign opcodes, and create instruction-decoding hardware for all necessary instruction formats. At the same time, the processor generator can also create the corresponding compiler and assembler for the long-word processor. The advantage of all this automated generation is that the hardware will be correct by construction and the required software-development tools are available simultaneously with the processor's hardware description.

For long-instruction-word architectures, hand packing of independent operations into long instructions is a very complex task. However, an appropriate assembler can automatically handle this packing, so that assembly source code programs written by programmers need only specify the individual operations, giving less attention to packing constraints. An appropriate compiler will generate code that is already aligned with operation-slot availability to maximize performance and minimize code size. Such a compiler will generally pack operations into long instructions.

Figure 3 shows a short but complete example of a very simple long-instruction word processor described in the TIE (Tensilica Instruction Extension) language using FLIX constructs. This TIE description relies entirely on built-in definitions of 32-bit integer operations. It defines no new operations; it merely bundles existing ones. Although quite short, this TIE description creates a highly parallel VLIW processor that delivers high performance even for applications written purely in terms of standard C integer operations and data types.

```
1: length ml64 64 {InstBuf[3:0] == 15}
2: format format1 ml64 {base slot, ldst slot, alu slot}
3: slot opcodes base slot {ADD.N, ADDX2, ADDX4, SUB, SUBX2, SUBX4, ADDI.N, AND, OR, XOR,
    BEQZ.N, BNEZ.N, BGEZ, BEQI, BNEI, BGEI, BNEI, BLTI, BEQ, BNE, BGE, BLT, BGEU, BLTU,
    L32I.N, L32R, L16UI, L16SI, L8UI, S32I.N, S16I, S8I, SLLI, SRLI, SRAI, J, JX, MOVI.N }
4: slot opcodes ldst slot { ADD.N, SUB, ADDI.N, L32I.N, L32R, L16UI, L16SI, L8UI, S32I.N,
    S16I, S8I, MOVI.N }
5: slot opcodes alu slot {ADD.N, ADDX2, ADDX4, SUB, SUBX2, SUBX4, ADDI.N, AND, OR, XOR,
    SLLI, SRLI, SRAI, MOVI.N }
```

Figure 3: A Simple 32-bit Multi-Slot Architecture Description

This TIE description creates a VLIW processor with three operation slots in its 64-bit instruction word. The first of the three operation slots supports all the commonly used integer operations including ALU operations, loads, stores, jumps and branches. The second operation slot offers loads and stores, plus the most common ALU operations. The third slot offers a full complement of ALU operations, but no loads and stores.

The first line of the example shown in Figure 4 declares a new instruction length (64 bits) and specifies the encoding of the first four instruction bits, which determine the instruction length. The second line of TIE code declares a format for that 64-bit instruction, called format1, which contains three operation slots called base_slot, ldst_slot, and alu_slot. This line of TIE code also names the three slots within the new format. The fourth line of TIE code specifies all of the operations that can be packed into the base_slot. In this case, all the instructions happen to be existing Xtensa LX instructions. However, it's also possible to invent new instructions and then assign them to this slot. The Xtensa processor generator also creates a NOP (no operation) 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. Lines 4 and 5 assign other instruction subsets to the other two slots.

Figure 4 defines a long-instruction-word architecture with a mix of built-in 32-bit operations and new 128-bit operations. Line 2 defines one 64-bit instruction format with three operation slots (base_slot, ldst_slot, and alu_slot). This description takes advantage of the Xtensa processor's predefined RISC instructions, but it also defines a large new register file and three new ALU operations for the new register file:

```
1: length ml64 64 {InstBuf[3:0] == 15}
2: format format1 ml64 {base slot, ldst slot, alu slot}
3: slot_opcodes base_slot {ADD.N, ADDX2, ADDX4, SUB, SUBX2, SUBX4, ADDI.N, AND, OR, XOR,
    BEQZ.N, BNEZ.N, BGEZ, BEQI, BNEI, BGEI, BNEI, BLTI, BEQ, BNE, BGE, BLT, BGEU, BLTU,
    L32I.N, L32R, L16UI, L16SI, L8UI, S32I.N, S16I, S8I, SLLI, SRLI, SRAI, J, JX, MOVI.N }
4: regfile x 128 32 x
5: slot_opcodes ldst_slot {loadx, storex}    /* slot does 128b load/store*/
6: immediate_range sim8 -128 127 1 /*8 bit signed offset field */
7: operation loadx {in x *a, in sim8 off, out x d} {out VAddr, in MemDataIn128}{
8: assign VAddr = a + off; assign d = MemDataIn128;}
9: operation storex {in x *a, in sim8 off, in x s} {out VAddr,out MemDataOut128}{
10: assign VAddr = a + off; assign MemDataOut128 = s;}
11: slot_opcodes alu_slot {addx, andx, orx} /* two new ALU operations on x regs */
12: operation addx {in x a, in x b, out x c} {} {assign c = a + b;}
13: operation andx {in x a, in x b, out x c} {} { assign c = a & b;}
14: operation orx {in x a, in x b, out x c} {} { assign c = a | b;}
```

Figure 4: Mixed 32-bit/128-bit Multi-slot Architecture Description

The first three lines shown in Figure 5 are identical to those of Figure . The fourth line declares a new 32-entry register file. Each entry is 128-bits wide. The fifth line declares and assigns the two new load and store instructions (loadx and storex) that work with the new register file. These new load instructions (defined in lines 7 through 10) are assigned to the long instruction word's second operation slot. The sixth line defines a new immediate range, an 8-bit signed value, to be used as the offset range for the new 128-bit load and store instructions.

Lines 7-10 fully define the new load and store instructions in terms of basic interface signals Vaddr (the address used to access local data memory), MemDataIn128 (the data being returned from local data memory), and MemDataOut128 (the data to be sent to the local data memory). The use of 128-bit memory data signals also guarantees that the local data memory will be at least 128 bits wide. Line 11 lists the three new ALU operations that can be put in the third operation slot of the long instruction word. Lines 12-14 fully define those operations for the 128-bit wide register file: add, bit-wise AND, and bit-wise OR.

With this example, any combination of the 39 instructions (including NOP) in the first operation slot, three instructions in the second operation slot (loadx, storex, and NOP), and four instruction in the third operation slot can be combined to form legal

instructions—resulting in a total of 468 combinations. This simplified example nearly specifies enough instructions to densely populate a long instruction word. The first slot needs about 21 bits, the second slot only needs about 19 bits, the third slot needs about 17 bits, and the format/length field required four bits—for a total of roughly 62 bits.

This example shows the potential power of instruction-level parallelism that arises from the ability to independently specify operations within a long instruction word. Moreover, all of the techniques associated with configurable processors that improve the performance of individual instructions—especially compound instructions (instruction fusion) and SIMD instructions—can be readily applied to the operations encoded in each operation slot.

Conclusion

VLIW processor architectures can deliver tremendous gains in processing power for ASIC and SOC designs. Code bloat, the major liability associated with VLIW architectures, can be avoided if the processor can handle variable-length instruction words. The use of customized, task-specific instructions in one or more VLIW operations slots further extends the processor's ability to quickly execute tasks without the need to increase clock rate. Finally, automated generation of the VLIW processor is critical to the practical application of such processors in ASIC and SOC designs.

Note: If you would like help creating long-instruction-word processors to boost the performance of your next ASIC or SOC design, contact Tensilica for a consultation.

US Sales Offices:

Santa Clara, CA office:
3255-6 Scott Blvd.
Santa Clara, CA 95054
Tel: 408-986-8000
Fax: 408-986-8919

San Diego, CA office:
1902 Wright Place, Suite 200
Carlsbad, CA 92008
Tel: 760-918-5654
Fax: 760-918-5505

Boston, MA office:
25 Mall Road, Suite 300
Burlington, MA 01803
Tel: 781-238-6702 x8352
Fax: 781-820-7128

International Sales Offices:

Yokohama office (Japan):
Xte Shin-Yokohama Building 2F
3-12-4, Shin-Yokohama, Kohoku-ku,
Yokohama
222-0033, Japan
Tel: 045-477-3373 (+81-45-477-3373)
Fax: 045-477-3375 (+81-45-477-3375)

UK office (Europe HQ):
Asmec Centre
Eagle House
The Ring
Bracknell
Berkshire
RG12 1HB
Tel : +44 1344 38 20 41
Fax : +44 1344 30 31 92

Israel:
Amos Technologies
Moshe Stein
moshe@amost.co.il

Beijing office (China HQ):
Room 1109, B Building, Bo Tai Guo Ji,
122th Building of Nan Hu Dong Yuan, Wang Jing,
Chao Yang District, Beijing, PRC
Postcode: 100102
Tel: (86)-10-84714323
Fax: (86)-10-84724103

Taiwan office:
7F-6, No. 16, JiHe Road, ShihLin Dist,
Taipei 111, Taiwan ROC
Tel: 886-2-2772-2269
Fax: 886-2-66104328

Seoul, Korea office:
27th FL., Korea World Trade Center,
159-1, Samsung-dong, Kangnam-gu,
Seoul 135-729, Korea
Tel: 82-2-6007-2745
Fax: 82-2-6007-2746