# Understanding Advanced Processor Features Promotes Efficient Coding

By David Katz, Tomasz Lukasiak and Rick Gentile, Blackfin Applications Group, Analog Devices, Inc.

Today's digital signal processors (DSPs) have achieved such an attractive mix of performance, peripheral mix, power dissipation and pricing that many system designers are eager to explore their benefits over the processors with which they've traditionally designed. One potential hurdle to this is the large amount of legacy C/C++ code they've developed for their application space. Clearly, these engineers would like to leverage their existing high-level code base on a DSP platform while taking advantage of DSP architectural features that enable performance unattainable on their former platform. Moreover, they require a familiar, intuitive development environment, as well as a straightforward way to implement assembly language routines selectively for increased performance. This article discusses programming strategies and techniques for DSPs in today's development environment.

## HLL vs Assembly – A Combination May Be Best

One mandatory task when undertaking a DSP-based project is deciding what kind of programming methodology to use. The choice is usually between assembly language and a high-level language (HLL) like C or C++. The decision revolves around many factors, so it's important to understand the benefits and drawbacks each approach entails.

Benefits of C/C++ include modularity, portability and reusability. Not only do the majority of embedded programmers have experience with one of these high-level languages, but also there exists a huge code base that can be ported from an existing microcontroller or DSP domain to a new DSP platform in a relatively straightforward manner. Because assembly language is architecture-specific, reuse is typically restricted to devices in the same processor family. Also, within a development team it is often desirable to have various teams coding different system modules, and a HLL allows these cross-functional teams to be processor-agnostic.

Traditional assembly languages have long been maligned for their arcane syntax and strange acronyms. Today, however, these factors are much less of an issue in architectures where so-called "algebraic syntax" is used. Figure 1 shows an example of a typical DSP instruction in the conventional style versus that of the algebraic format. It is clear that the latter's structure is much more intuitive.

*Figure 1: A comparison between traditional assembly syntax and more modern algebraic syntax. In the examples provided, the r- registers are data registers, and the p- registers are pointer registers.*

| Type of Operation | Traditional Assembly Syntax | Algebraic Assembly Syntax |
|---|---|---|
| Moving register content | mov r7, r0 | r7 – r0 |
| Addition | add r0, r1, r2 | r0 = r1 + r2 |
| Subtraction | sub r3, r3, r1 | r3 = r3 - r1 |
| Loading from memory to a register | lw r5, p3 | r5 = [p3] |
| Storing the contents of a register into memory | sw r1, p0 | [p0] = r1 |
| Conditional jump to _equal if the input registers are equal, or to _not_equal, otherwise | beq r5, r6, _equal<br>bne r5, r6, _not_equal | cc = r5 == r6<br>if cc jump _location<br>if !cc jump _location |
| Load a register from memory and increment pointer register | lw r3, p5<br>addi p5, p5, 1 | r3 = [p5++] |

One reason assembly has been difficult to program is its focus on data flow between the DSP's actual register sets, computational units and memories. In C/C++, this manipulation typically occurs at a much more abstract level through the use of variables and function/procedure calls, making the code easier to follow.

Today's C/C++ compilers are quite resourceful, and many can do an admirable job of compiling the HLL code into tight assembly code. In fact, it's often best to just let the compiler optimizer do its job. However, the fact remains that compiler performance is tuned to a specific set of features that the tool developer considered most important. Therefore, it cannot exceed handcrafted assembly code performance in all situations.

The bottom line is that developers use assembly language only when it is necessary to optimize important processing-intensive code blocks for efficient execution on the DSP. HLL compiler optimization switches can do an admirable job, but nothing beats thoughtful, direct control of DSP data flow and computation. This is why designers often use a combination of C/C++ and assembly. The HLL is fine for the control and basic data manipulation, but the assembly shines for efficient numeric computation.

**Architectural Features for Efficient Programming**

In order for the assembly programmer to do an effective job, it is imperative to understand the types of structures that can differentiate DSPs from processors not

optimized for super-fast number crunching. These features include:

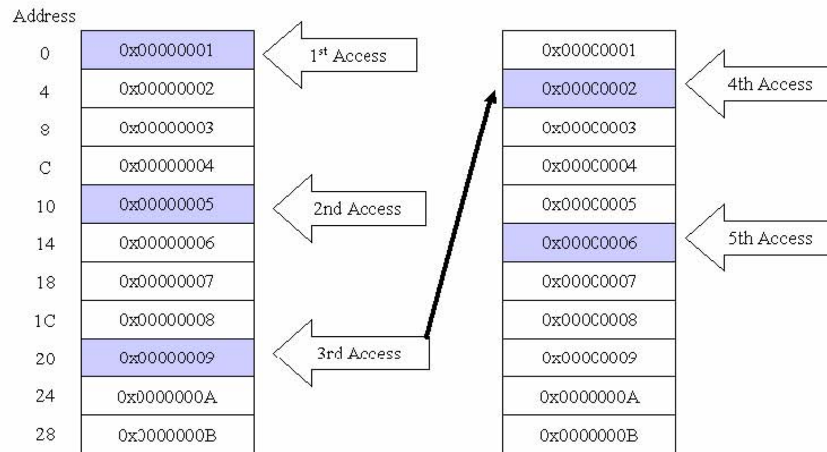
- . • Specialized addressing modes
- . • Hardware Loop Constructs
- . • Cacheable memories
- . • Multiple operations per cycle
- . • Interlocked pipeline
- . • Flexible data register file

These features can make an enormous difference in computational efficiency. Let's discuss each one in turn.

*Specialized Addressing Modes*

Allowing the processor to access multiple data words in a single cycle requires complete flexibility in address generation. In addition to the more DSP-centric access sizes along 16 and 32-bit boundaries, byte addressing is required for the most efficient processing. This is important because some common applications, including many video-based systems, operate on 8-bit data. When memory accesses are restricted to a single boundary, extra cycles may be required for the processor to mask off relevant bits.

Another beneficial addressing capability is "circular buffering". This feature must be supported directly by the processor with no special software management overhead. Circular buffering allows a programmer to define buffers in memory and stride through them automatically. Once the buffer is set up, no special software interaction is required to navigate through the data. The address generator handles non-unity strides and, more importantly, handles the "wrap-around" feature illustrated in Figure 2. Without this automated address generation, the programmer would have to manually keep track of the buffer, thus wasting valuable processing cycles.




- Base address and starting index address = 0x0
- Index address register I0 points to address 0x0
- Buffer length L = 44
  (11 data elements * 4 bytes/element)
- Modify register M0 = 16
  (4 elements * 4 bytes/element)

**Sample code:**

```
R0 = [I0++M0];   // R0 = 1 & I0 points to 0x10 after execution
R1 = [I0++M0];   // R1 = 5 & I0 points to 0x20 after execution
R2 = [I0++M0];   // R2 = 9 & I0 points to 0x04 after execution
R3 = [I0++M0];   // R3 = 2 & I0 points to 0x14 after execution
R4 = [I0++M0];   // R4 = 6 & I0 points to 0x24 after execution
```

*Figure 2: Example of circular buffering*

An essential addressing mode for efficient signal processing operations such as the FFT and DCT is bit reversal. Just as the name implies, “bit reversal” involves reversing the bits in a binary address. That is, the least significant bits are swapped in position with the most significant bits. The data ordering required by a radix-2 butterfly is in "bit-reversed" order, so bit-reversed indices are used to combine FFT stages. It is possible to calculate these bit-reversed indices in software, but this is very inefficient. An example of bit reversal address flow is shown in Figure 3.

| Address LSB | Input buffer | Bit-reversed buffer | Address LSB |
|---|---|---|---|
| 000 | 0x00000000 | 0x00000000 | 000 |
| 001 | 0x00000001 | 0x00000004 | 100 |
| 010 | 0x00000002 | 0x00000002 | 010 |
| 011 | 0x00000003 | 0x00000006 | 110 |
| 100 | 0x00000004 | 0x00000001 | 001 |
| 101 | 0x00000005 | 0x00000005 | 101 |
| 110 | 0x00000006 | 0x00000003 | 011 |
| 111 | 0x00000007 | 0x00000007 | 111 |

**Sample code:**

```
LSETUP(start,end) LC0 = P0;              //Loop count  P0= 8
start: R0 = [I0] || I0 += M0 (BREV)  ;   // I0 points to input buffer, automatically incremented in bit-reversed progression
end:[I2++] = R0;                          // I2 points to bit-reversed buffer
```

*Figure 3: Hardware Bit-Reversal Mechanism*

*Hardware Loop Constructs*

Looping is a critical feature in communications processing algorithms. There are two key looping-related features that can improve performance on a wide variety of algorithms. The first is referred to as a "zero-overhead hardware loop". As with the addressing capabilities, the looping constructs are implemented in hardware. Again, while this function could be accomplished in software, the associated overhead would cut into the real-time processing budget. Zero overhead loops allow programmers to initialize loops by setting up a count value and defining the loop bounds. The processor will continue to execute this loop until the count has been reached.

Zero-overhead loops are part of most processors, but "hardware loop buffers" can really add increased performance in looping constructs. They act as a type of cache for instructions being executed in the loop. For example, after the first time through a loop, the instructions can be kept in the loop buffer, eliminating the need to "re-fetch" the same instructions over and over again each time through the loop. This can produce a significant savings in cycles by keeping the loop instructions in a buffer where they can be accessed in a single cycle. This feature requires no additional setup by the programmer but it is important to know the size of this buffer so that loop sizes can be selected intelligently.

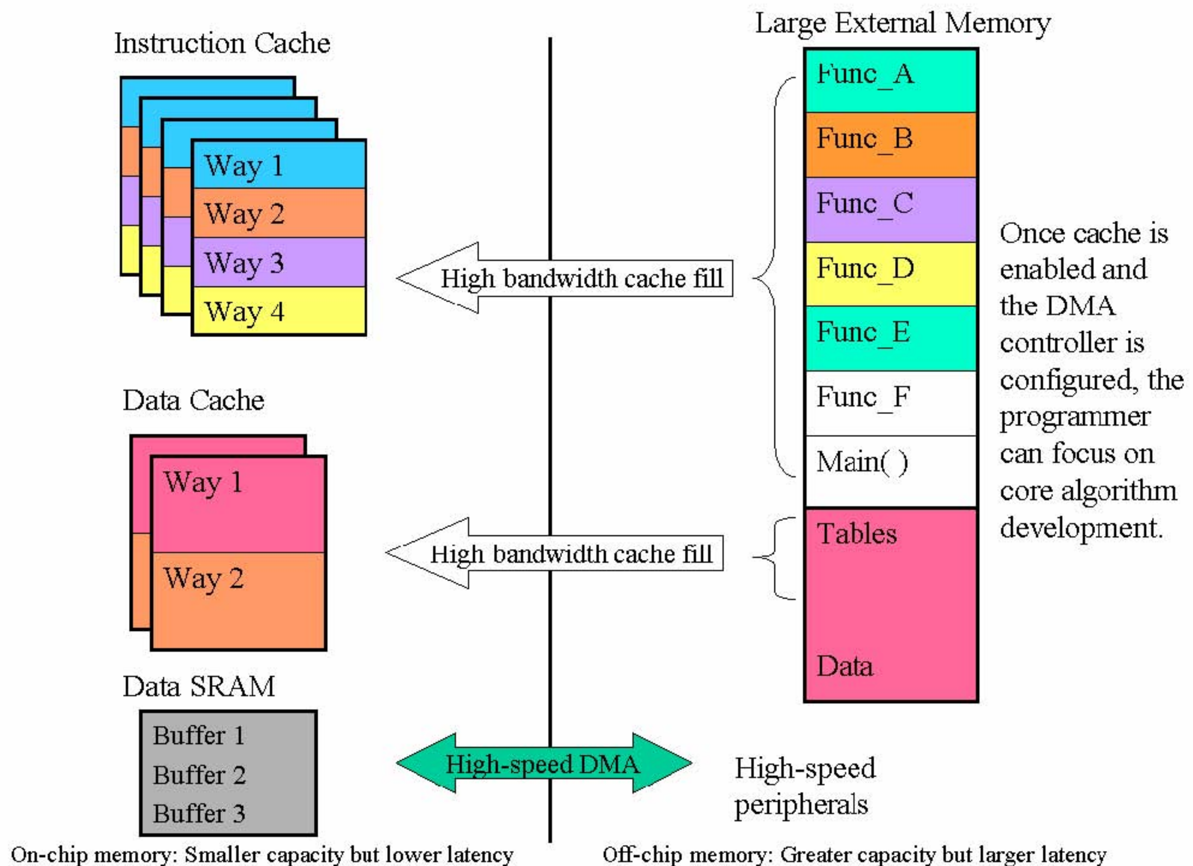
*Cacheable Memories*

Typical DSPs usually have a small amount of fast, on-chip memory. Microcontrollers usually have access to large external memories. A hierarchical memory architecture combines the best of both approaches, providing several levels of memory with different performance levels. For applications that require the most determinism, on-chip SRAM can be accessed in a single core clock cycle. For systems with larger code sizes, large, higher-latency on-chip and off-chip memory is available.

By itself, this hierarchy is only moderately useful, since today's high-speed processors would effectively run at much slower speeds because larger applications would only fit in slower external memory. Additionally, programmers would be forced to manually move key code in and out of internal SRAM. However, by adding data and instruction caches into the architecture, external memory becomes much more manageable. The cache reduces the manual movement of instructions and data into the processor core. This greatly simplifies the programming model by eliminating the need to worry about managing the flow of data and instructions into the core.

Figure 4 demonstrates a typical memory configuration where instructions are brought in from external memory as they are needed. Instruction cache usually operates with some type of Least Recently Used (LRU) algorithm, insuring that instructions that run more often get replaced less often. The figure also illustrates that having the ability to configure some on-chip data memory as cache and some as SRAM can optimize performance. DMA controllers can feed the core directly, while data from tables can be brought in to the data cache as they are needed.

Figure 4: Configurable Cache and Memory Structures Can Optimize Data Movement




*Multiple Operations per Cycle*

Processors are often benchmarked by how many millions of instructions they can execute per second (MIPS). However, for modern processors this can be misleading because of the confusion surrounding what actually constitutes an instruction. For example, multi-issue instructions, which were once reserved for use in higher-cost parallel processors, are now also available in low-cost, fixed-point processors. In addition to performing multiple ALU/MAC operations each core processor cycle, additional data loads and stores can also be completed in the same cycle. The memory is typically portioned into sub-banks that can be dual-accessed by the core and optionally by a DMA controller. Factoring in the hardware-based address calculations described above, it is apparent that a lot can happen in a single cycle.

An example of a multi-operation instruction is shown in Figure 5. As shown, in addition to 2 separate MAC operations, a data fetch and data store can also be accomplished in the same processor clock cycle.

**R1.H=(A1+=R0.H*R2.H), R1.L=(A0+=R0.L*R2.L) || R2 = [I0--] || [I1++] = R1;**

R1.H=(A1+=R0.H*R2.H), R1.L=(A0+=R0.L*R2.L)
- multiplication R0.H*R2.H
  - accumulation to A1
  - store to R1.H
- multiplication R0.L*R2.L
  - accumulation to A0
  - store to R1.L

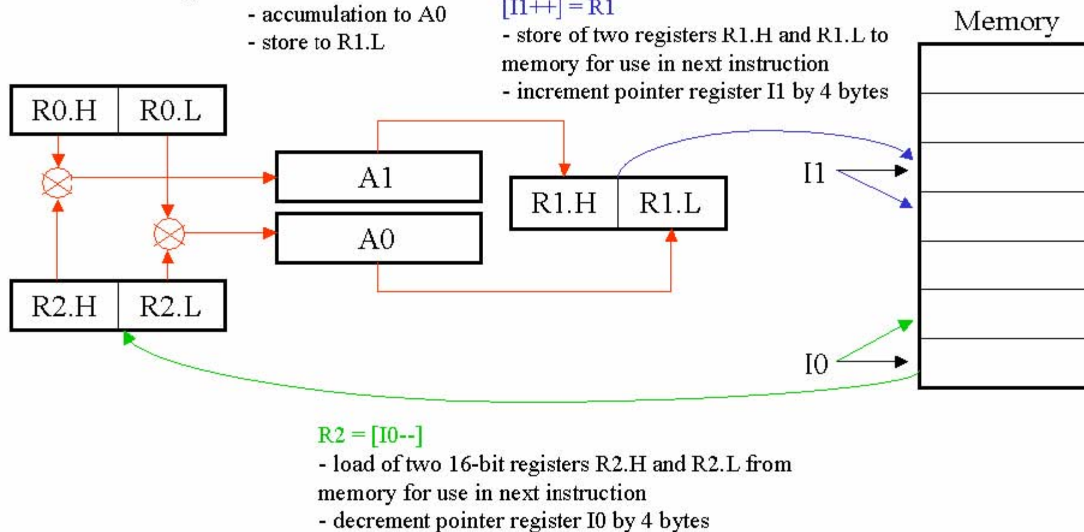



*Figure 5: A Blackfin Multi-Issue Instruction Performs Several Operations in a Single Cycle*

*Interlocked Pipeline*

As processors increase in speed, it is necessary for the processing pipeline to become deeper in terms of overall stages. This is important to understand because when assembly programming is required, the pipeline can make programming more challenging. Some processors, however, have an "interlocked" pipeline. This means that when assembly programming is performed, the programmer does not have to manually schedule or keep track of data and instructions moving through the pipe. The processor automatically handles stalls and bubbles.

*Flexible Data Register File*

Finally, another complementary feature is a versatile data register set. In traditional fixed-point DSPs, word sizes are usually fixed. However, there is an advantage to having data registers that can be treated as either a 32-bit word (e.g., R0) or two 16-bit words (R0.L and R0.H, for the low and high halves, respectively). In a dual-MAC system, this allows operation on four pieces of 16-bit data in a single cycle.

## Code Comparison and Analysis

The architectural framework described above is the foundation of efficient DSP programming. Many ubiquitous number-crunching algorithms can be executed extremely fast if the programmer utilizes the full potential of the processor's features. Below is a selection of a few common algorithms with a description of how they should be executed on a DSP. Note that, while the code efficiency needs to be examined at the assembly level, modern optimizing DSP compilers are designed to utilize many of the same rules that are at the disposal of an assembly programmer. For illustration, Blackfin processor assembly language is used in the examples.

Dot Product The dot product, or scalar product, is an operation useful in measuring orthogonality of two vectors. Most C programmers would be familiar with the following implementation of a dot product:

```
short dot(short a[], short b[], int size) { int i; int output = 0;

  for(i=0; i<size; i++) {
    output += (a[i] * b[i]);
  }
  return output;
```

Below is the main portion of the assembly code:

```
//P0=loop count, I0 & P1 are address registersA1 = A0 = 0; // A0 & A1 are accumulatorsLSETUP (loop1,loop1) LC0 = P0 ; // Set up hardware loop starting at label loop1:loop1: A1 += R1.H * R0.H , A0 += R1.L * R0.L || R1 = [ P1 ++ ] || R0 = [ I0 ++ ] ;
```

The following points illustrate DSP architectural features that facilitate this tight coding.

*Hardware loop buffers and loop counters eliminate the need for jump instructions at the end of each iteration.* Since a dot product is a summation of products, it is implemented in a loop. Many RISC microcontrollers use a jump instruction at the end of each iteration in order to process the next iteration of the loop. The assembly program shows the LSETUP instruction, which is the only instruction needed to implement a loop.

*Multi-issue instructions allow the execution of instructions and two data accesses in the same cycle.* In each iteration, the values a[i] and b[i] must be read, then multiplied, and finally written back to the running summation in the variable output. On many microcontroller platforms, this effectively amounts to four instructions. The last line of the assembly code shows that all of these operations can be executed in one cycle.

*Parallel ALU operations allow two 16-bit instructions to be executed simultaneously.* The assembly code shows two accumulator units (A0 and A1) being used in each iteration. This reduces the number of iterations by 50%, effectively halving the original execution time.

FIR The finite impulse response filter is a very common filter structure equivalent to the convolution operation. A straightforward C implementation looks very similar to the dot product:

```
// sample the signal into a circular buffer
x[cur] = sampling_function();
cur = (cur+1)%TAPS; // advance the cur pointer in a circular fashion


// perform the multiply-
addition
y = 0;
for (k=0; k<TAPS; k++)
{

  y += h[k] *
x[(cur+k)%TAPS];
}
```

The essential part of an FIR kernel written in assembly shows a format similar to that of the dot
product. In fact, the same DSP features were used to deliver maximum performance to the
algorithm's execution. In this specific example, the samples are stored in the R0 register, while the coefficients are stored in the R1 register.

```
// P0 holds # of filter taps R0=[I0++] || R1=[I1++]; // set initial
values for R0 and R1 A1=A0=0;   // zero the accumulators LSETUP (loop1, loop1) LC0 = P0; // configure inner
looploop1: A1+=R0.L*R1.L, A0+=R0.H*R1.H || R0 = [I0++] || R1 = [I1++]; // compute
```

Besides the features described for the dot product, the FIR algorithm shown above exploits circular buffering.

*Circular buffers eliminate the need for explicit modulus arithmetic.* In the C code snippet, the % (modulus) operator provides a mechanism for circular buffering. As shown in the assembly kernel, this modulus operator does not get translated into an additional instruction inside the loop. Instead, the Data Address Generator registers I0 and I1 are configured outside the loop to automatically wrap around to the beginning upon hitting the coefficient buffer boundary.

FFT A Fast Fourier Transform is an integral part of many signal-processing algorithms. One of its peculiarities is that the input vector is in sequential time order, but the output comes out in bit-reversed order. Most traditional general-purpose processors require the programmer to implement a separate routine to unscramble the bit-reversed output. On a DSP platform, bit reversal is designed into the addressing engine.

*Bit-reversed addressing eliminates the need for a separate bit-reversing procedure in an FFT implementation.* Allowing the hardware to automatically bit-reverse the output of an FFT algorithm relieves the programmer from writing additional utilities, and thus improves performance.

In addition to the instruction constructs shown above, some processors also include an additional set of dedicated instructions to support a wide range of applications. The purpose of these instructions is to further extend the processing capabilities to algorithms such as Viterbi, Huffman coding and many other bit manipulation routines.
Clearly, there is much to consider when defining a programming strategy for a DSP-based application. Using C or C++ with a strong compiler/optimizer can produce robust results much of the time, but handcrafted assembly is often the best way to gain extra performance out of a processor. However, this effort must be undertaken only after gaining a thorough understanding of the architectural blocks that promote efficient coding.