Application Report
SPRA476

Using the TMS320C6x in Non-Traditional DSP Applications

Mathew George, Jr. (Joe) *Digital Signal Processing Solutions*
Mohsen Khayami

Abstract

The Texas Instruments (TI™) TMS320C6x digital signal processor (DSP) architecture, with its RISC-like instruction set, flexible parallelism, and conditional execution, can be used in non-typical DSP applications from microcontroller-type to FPGA/ASIC/data flow-type tasks. This paper uses code examples to explore ways to efficiently handle bit manipulation, address manipulation, and dataflow configurations. In addition, this document includes an example table lookup benchmark and a system architecture discussion for data input/output.

Contents

Figures

Introduction

The TMS320C6x DSP is not a traditional DSP, even though it handles traditional DSP applications, such as filtering, FFTs, and vocoders, that other DSPs do. The C6x also includes a variety of additional features that make it attractive in non-standard DSP applications. These applications include, but are not limited to:

- Microcontroller-style bit manipulation (or “bit banging” as it is often called) instructions (in some cases performed even better than with microcontrollers and in a single 5-ns cycle)
- Byte addressibility
- Address manipulation (with improved results over a C3x/C4x)
- Dynamic operations (performed as well as those by a C3x/C4x)
- Efficient “conditionally execute” method on the C6x for the classic bit test/branch seen in “controller”-type housekeeping code
- Ability to replace an FPGA/ASIC with a C6x in a “dataflow” style design
- Use of innovative C6x tools to develop these operations easily
- Elegant use of four-channel C6x DMA (direct memory access) for data movement

This application report examines various aspects of the C6x and their implementation in code or hardware. Specific architectural features are described and accompanied by code examples. The document includes an application example and discusses how tools assist in the process of optimization. An elegant hardware architecture is also presented for data movement and processing. The information presented in this document should encourage an appreciation of the power of the C6x DSP.

C6x CPU/Instruction Features With Code Examples

Let us now examine various non-traditional features of the C6x architecture through its instruction set that a C6x does well and traditional DSPs often do not. A graphic example of the concept and/or application followed by a code segment is provided for each. Note that the code examples are authentic (assembled and run on a simulator) assembly code, but most are **UNOPTIMIZED** and meant for purely descriptive, academic purposes.

The features are classified into three descriptive groups: bit manipulation, address manipulation, and decision execution. All these features are important in enabling the c6x to optimally execute some of these non-traditional functions.

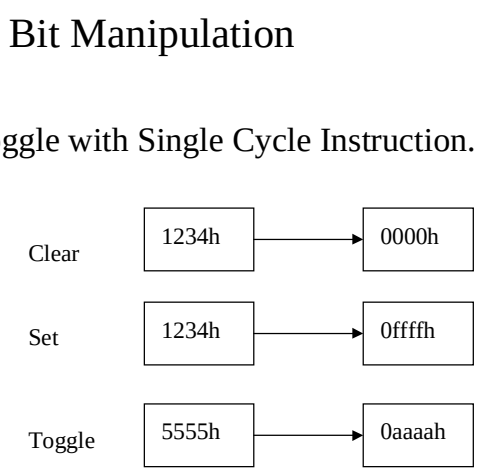
Bit Manipulation

This section examines two types of bit manipulation done in both microcontroller-type and ASIC/FPGA-type applications. In microcontroller-type applications, registers are often manipulated to control peripherals or to perform housekeeping functions. In ASIC/FPGA-type applications, fast data streams are often manipulated. Note that bit manipulation is usually done on data and not addresses (for more information, see the section, *Address Manipulation*).

Clear/Set/Toggle

Figure 1 shows a value in a register being set, cleared, and toggled, then placed in another register. This value might have been loaded from a register or as part of a data stream.

Figure 1. Clear/Set/Toggle Example



The C6x code that corresponds to each of these operations is shown in Figure 2.

Figure 2. Clear/Set/Toggle Code

Bit Manipulation

- Set/Clear/Toggle with single cycle instruction*.

```
            .text
; Typical bit banging
bitbang:  MVK     .S1     bbdata, A15      ; Initialize pointer with MVK/MVKH**
          MVKH    .S1     bbdata, A15
          LDW     .D1     *A15, A0;        Load value bbdata=12345555h
          NOP             4
                                           ; A0 = 12345555h
          CLR     .S1     A0, 24, 31, A0   ; CLEAR upper byte of upper halfword
                                           ; A0 = 00345555h
          SET     .S1     A0, 16, 23, A0   ; SET lower byte of upper halfword
                                           ; A0 = 00ff5555h
          XOR     .S1     A0, -1, A0       ; TOGGLE
                                           ; A0 = 0ff00aaaah
          .data
bbdata: .word     012345555h
```

*See TMS320C62xx CPU and Instruction Set Reference Guide pp. 3-42, 92, and 117. 5

**See TMS320C62xx CPU and Instruction Set Reference Guide pp. 3-77 to 3-80.

The three operations are shown in bold in Figure 2. Each instruction is executed in a single C6x cycle.

To set up the example, the address (pointer) of bbdata, where bbdata is located in the .data section, is loaded into register A15 using MVK/MVKH instructions. The value of bbdata is loaded into register A0 using the LDW instruction with *A15 acting as the pointer register. The NOP 4 is due to pipeline considerations.

NOTE:
Remember that most of the following code examples are **UNOPTIMIZED.**

SET/CLR is accomplished by specifying from which bit to which bit needs to be set or cleared from bits 0 to 31. The value is specified here as a constant (and fits well in the opcode because two 5-bit constants fit well in the opcode).

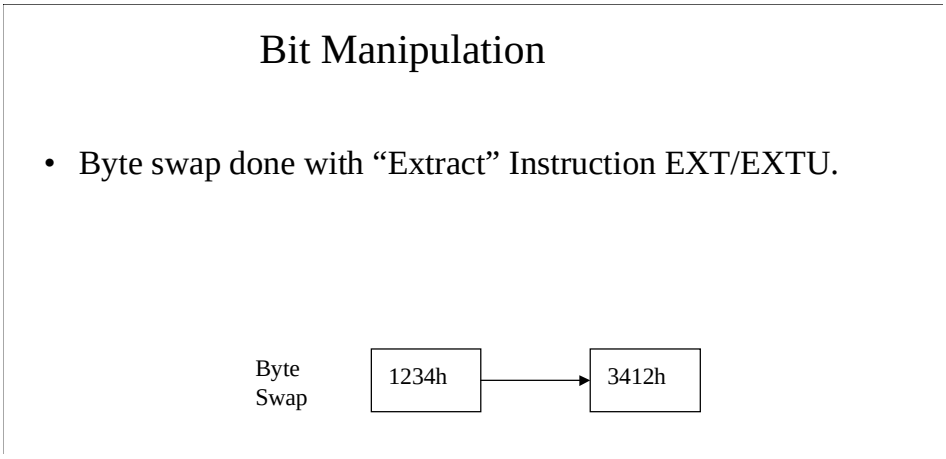
With XOR, the entire 32 bits of A0 can be toggled because the constant value “-1” is sign-extended before the operation is done. Please note that the above instructions can also be executed using a mask (and hence real-time dynamically, if needed) in a register.

These operations are standard in microcontrollers and not so well supported in our other TMS320 DSPs. C2xx requires the one accumulator and C54x requires one of the two accumulators, making them unavailable for other operations. The C5x has a PLU (parallel logic unit) that directly manipulates the data, thus off-loading the accumulator and offering an advantage over the other fixed-point processors. Even the C3x does not have a SET/CLR. In the TMS320 family, only the C8x parallel processor (PP) offers improved performance over the C6x for these types operations.

Byte-Swapping

Byte swapping is a classic operation going back to the Intel and Motorola models of big-endian and little-endian. Usually very difficult in software, shifters made the problem much easier in other TMS320 DSPs. The byte-swapping operation is shown in Figure 3.

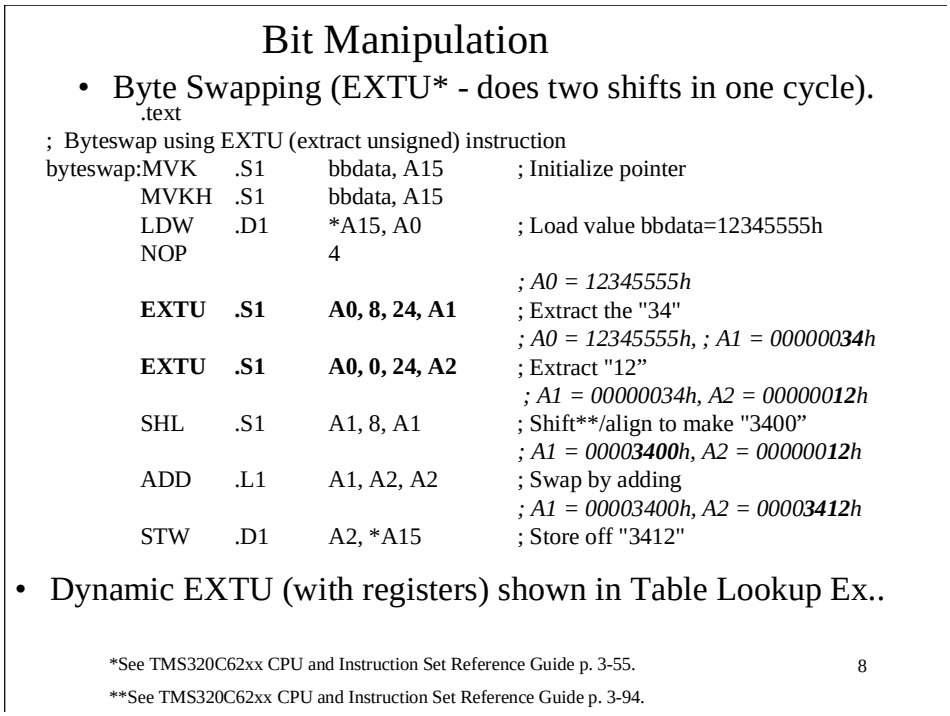
Figure 3. Byte-Swap Example



This operation must be accomplished in pure software without a shifter. Multiplication (which is slow on most microprocessor/microcontrollers) is probably required along with some masking and addition.

In the case of the C6x, the EXT instruction makes the job even easier by letting the programmer actually **pick out the contiguous bits** he or she wants to manipulate. This powerful feature is implemented in an interesting way with two shifts in one cycle. (See Figure 4 for the C6x code.)

Figure 4. Byte-Swap Code



Bit Manipulation

- Byte Swapping (EXTU* - does two shifts in one cycle).

```
        .text
;  Byteswap using EXTU (extract unsigned) instruction
byteswap:MVK    .S1     bbdata, A15     ; Initialize pointer
         MVKH   .S1     bbdata, A15
         LDW    .D1     *A15, A0        ; Load value bbdata=12345555h
         NOP            4
                                        ; A0 = 12345555h
         EXTU   .S1     A0, 8, 24, A1   ; Extract the "34"
                                        ; A0 = 12345555h, ; A1 = 00000034h
         EXTU   .S1     A0, 0, 24, A2   ; Extract "12"
                                         ; A1 = 00000034h, A2 = 00000012h
         SHL    .S1     A1, 8, A1       ; Shift**/align to make "3400"
                                        ; A1 = 00003400h, A2 = 00000012h
         ADD    .L1     A1, A2, A2      ; Swap by adding
                                        ; A1 = 00003400h, A2 = 00003412h
         STW    .D1     A2, *A15        ; Store off "3412"
```

- Dynamic EXTU (with registers) shown in Table Lookup Ex..

*See TMS320C62xx CPU and Instruction Set Reference Guide p. 3-55. 8

**See TMS320C62xx CPU and Instruction Set Reference Guide p. 3-94.

After loading the pointer and value as in the last two example codes, EXTU (the “U” means unsigned) pulls out the appropriate 8 bits specified in this case as constants. The first bolded EXTU pulls out the “34” and saves in a register while the second bolded EXTU pulls out the “12” and saves in a register. The “34” is then left-shifted to make “3400” and added to the “12”. In other processors without an EXT instruction, the values must be masked off.

Using the EXT instruction, the first number denotes how many bits on the **left** to throw out. A slight wrench in the system is the fact that the second number denotes how many bits on the **right** to throw out **PLUS** how many bits on the **left** are already thrown out.

That’s right. You must specify the bits to the left twice. This is because the operation is accomplished with 2 shifts in one cycle. After you shift left to throw out the bits to the left, you must shift right the same distance to return to the original position to start shifting right, if you want a right-justified answer in the destination register. (In some operations, such as an optimized byte-swap, you might not want the answer right justified.) This instruction is graphically explained in the *TMS320C62xx CPU and Instruction Set Reference Guide*, p. 3-55. Again, note that the value can by dynamic in a register (this is shown in the *Application Example* section).

Again, remember that this example is UNOPTIMIZED. (This operation can be accomplished in 2 cycles instead of 4 cycles by parallelizing the EXTU with one of them NOT right justifying “34” in the first cycle and then adding in the second cycle.)

But data is not the only item that might need to be manipulated. Addresses also often need some manipulation, as discussed in the following section.

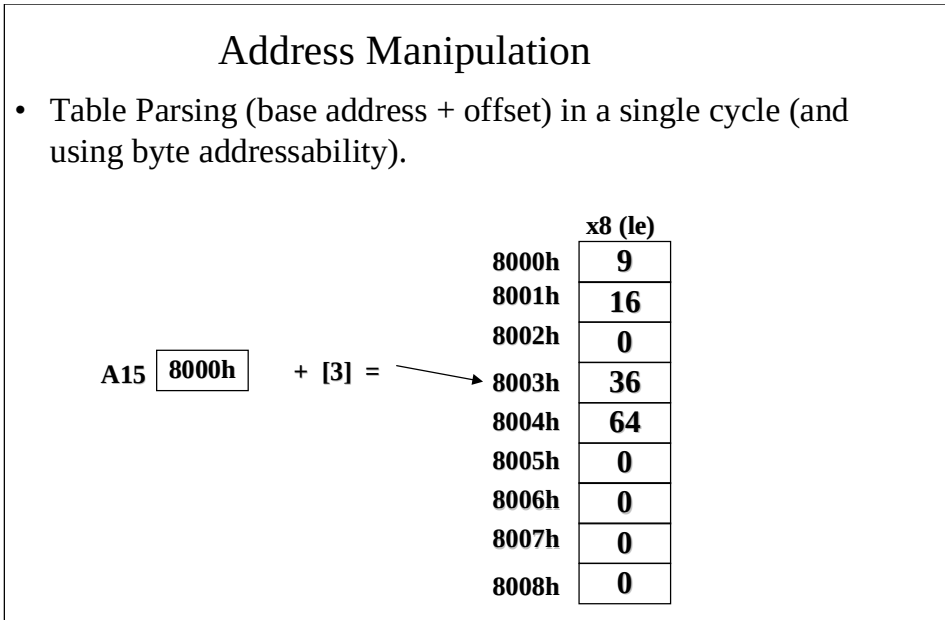
Address Manipulation

In the *Bit Manipulation* section, we said that bit manipulation is often performed on data. Theoretically, you can perform bit manipulation on addresses also, treating it as data. This section describes how the C6x CPU treats addresses. Most processors provide a specific, separate set of address registers to allow “pointer”-type address manipulation. In contrast, on the C6x, ALL 32 C6x “general-purpose” registers can be used as address/pointer registers.

Table Parsing

Table parsing/lookup is important to allow a base-pointer register setup, from which offsets can be applied to jump through a table. The C3x/C4x does this well, but only the C54x came close for fixed-point processors and that was a constant (or immediate) modify usually only good for stacks. Figure 5 shows a contrived example of table lookup for summing some Pythagorean triples. A15 is the base register with the address “8000h” and the offset is indicated by the value in the “[]”.

Figure 5. Table Parsing Example



Although this example is not complex, it shows the capability for byte addressing that DSPs other than the C8x (and C32 for dynamic memory only) do not support. The corresponding code itself is shown in Figure 6:

Figure 6. Table Parsing Code

Address Manipulation

- Table Parsing (base address + offset)* in a single cycle.

```
            .text
; Table look up (base address + offset) for Pythagorean Triples c^2=a^2 + b^2 calculation
tablel:     MVK     .S1     table, A15          ; Initialize pointer
            MVKH    .S1     table, A15

            LDB     .D1     *A15[0], A0         ; Load a^2
            LDB     .D1     *A15[1], A1         ; Load b^2
            NOP             4
                                                ; A0 = 00000009h, ; A1 = 00000010h
            ADD     .L1     A0, A1, A1          ; Calculate c^2
                                                ; A1 = 000019h
            STW     .D1     A1, *A15[2]         ; Store c^2
                                                ; table:   .word    9, 16, 25

            .data
table:      .byte   9, 16, 0            ; (3^2) + (4^2) = (5^2)
            .byte   36, 64, 0           ; (6^2) + (8^2) = (10^2)
```

- Dynamic parsing/addressing available with registers. 10

*See TMS320C62xx CPU and Instruction Set Reference Guide p. 3-20.

As in previous examples to set up the pointer, bbdata is loaded into register A15 using MVK/MVKH instructions. The value of bbdata is loaded into register A0 using the LDB instruction (to show byte addressibility), with *A15 acting as the pointer register. The NOP 4 is due to pipeline considerations. Remember that the code examples are UNOPTIMIZED**.**

This contrived example reads the squares of a table as bytes and adds them together. The resulting value is then written over the initialized “zero”, again as a byte. The index for each element is denoted by the “[]” in the LDB instruction. Remember that in the C6x pure load/store architecture, only LD and ST instructions can perform address accesses. Thus, you see a “*”with only one of these two instructions.

This method also works well for manipulating the registers dedicated to a peripheral (such as the C6x McBSP or DMA). The main peripheral control register often comes first in the memory map, which can be used as the base. Other secondary peripheral registers are used as the offsets.

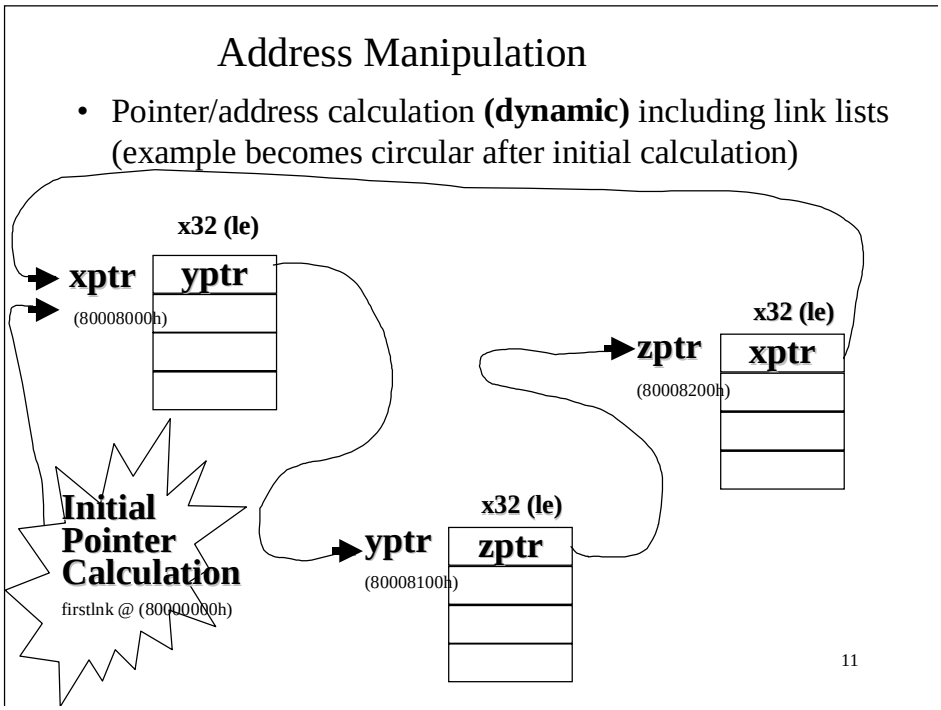
In this example, the offsets are the constant (immediate) offsets that we derided in the beginning of this section. The following section shows not only a dynamic example but also two other C6x features.

Link Lists

Dynamically calculating pointer addresses often constitute bad programming practice but are often used extensively in real-time processor code. This example first calculates the initial address of a linked list (dynamically). It then shows how the pointer access is accomplished using the same register (a C6x feature mentioned in the *Address Manipulation* section.). And finally, the example shows a subtle feature of how the link list can be circular with **one** instruction on the C6x.

Figure 7 shows the example.

Figure 7. Link List Example



The value for xptr is initially dynamically calculated (and forced to 80008000h) and then a link list points to the next location in a circular fashion. Note that each “ptr” could be an arbitrary place in memory that just points to the next “ptr” in an arbitrary place in memory. Figure 8 shows the corresponding code:

Figure 8. Link List Code

Address Manipulation

- Link lists - using the fact that all 32 registers can be used for both calculation/general purpose and pointer/address functions.

```
                .text
; Circular three element link list load
llcirc:     MVK     .S1     firstlnk, A15       ; Initialize firstlnk pointer
            MVKH    .S1     firstlnk, A15
            MVK     .S1     08000h, A1          ; Hand calc xptr offset
            MVKH    .S1     0, A1               ; Clear upper 16 bits
                                                ;A1=8000h, A15=80000000h
            ADD     .S1     A1, A15, A15        ; Add to firstlnk (bad programming practice)
                                                ; n=0 and A15 = xptr=80008000h
circ:       LDW     .D1     *A15, A15           ; Load next link
            NOP             4
; n=1 and A15 = yptr,  n=2 and A15 = zptr,  n=3 and A15 = xptr,  n=4 and A15 = yptr….
            B       .S1     circ                ; Repeat infinitely
            NOP             5

            .data
firstlnk    .word   firstlnk                    ; at 80000000h
            .sect "ptrs"
xptr        .word   yptr                        ; at 80008000h
yptr        .word   zptr                        ; at 80008100h
zptr        .word   xptr                        ; at 80008200h
```

12

The pointer “firstlnk” is initialized in data memory (at 80000000h to push the example) and loaded with MVK/MVKH. Then the address that the pointer “firstlink” points to has 8000h added to it to get the hard-coded address (as hard-coded in the linker command file) of xptr. (There technically should be a separate “ptrs” section with a .sect directive for **EACH** pointer to be accurate to the addresses shown in the comments in the .sect directive above).

Then the LDW overwrites the present pointer with the next one in a circular endless-loop fashion.

This single instruction pointer update/overwrite is possible because all 32 registers on the C6x (A0–A15 and B0–B15) can be **BOTH** calculation (or general-purpose) **AND** address (or auxiliary) registers. No other TMS320 can do this. Thus, cycles are not wasted moving the value from general-purpose register/accumulator to an address/auxiliary register.

But address and data manipulation are not the only features that the C6x does well. The execution of code, especially decision execution, should next be examined.

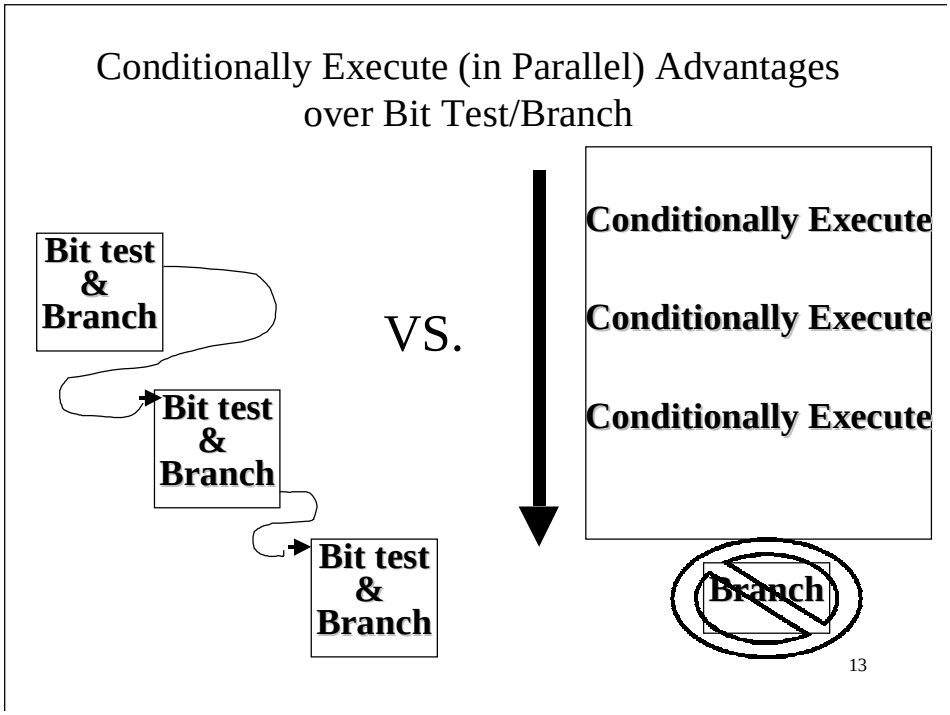
Decision Execution (Conditionally Execute Advantages Over Bit Test/Branch)

Much non-traditional DSP code involves the “controller”, housekeeping-type functions that often involve decision trees with bit testing and branches. The disadvantage of this on many DSPs (and other microprocessors) is the branching overhead caused by deep pipelines. Previous TMS320 DSPs needed 3–5 cycles of overhead per branch (sometimes the overhead was reduced with a delayed branch instruction).

The C6x overhead can be 5 cycles in the traditional sense, if no delay slots are used. (Microcontrollers often have shorter pipelines but much slower cycle times, so the overall execution speed is much worse than with a DSP.) Often the delay slots do not help when there are tight data dependencies; that is, when the next decision is based on very few operations following the results of the last decision. Such a configuration is inherently inefficient.

One option to optimally execute decisions on the C6x with tight data dependencies uses the C6x feature in which every instruction can be conditionally executed. This option presents a linear, non-branching method of achieving these decision trees. The concept is shown in Figure 9.

Figure 9. Decision Execution Concepts



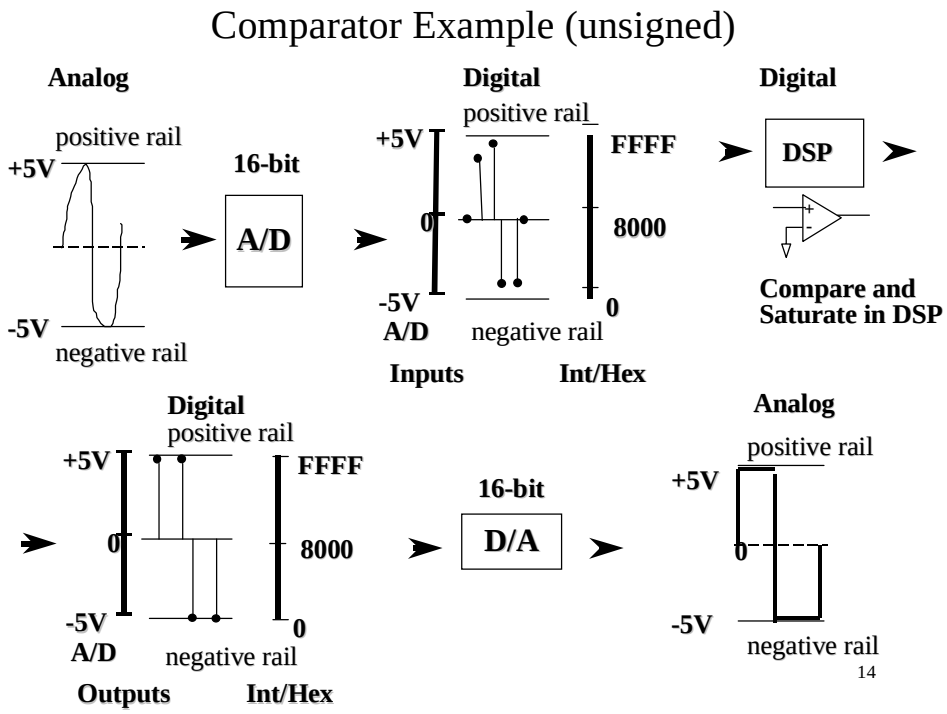
Instead of the classic “bit test and branch” that flushes the pipeline on each decision, as shown on the left side of the diagram, either execute or NOP the instruction based on a condition. This method avoids branching overhead.

Other microprocessors, often RISCs, use this methodology. Some, such as the Intel IA-64, go as far as to execute both legs of a branch ahead of time until it is determined which leg will be used, at which point the other one is voided. Of course, this method is expensive in hardware. The C6x method is software-based and less expensive in hardware.

Comparator Example

The real-world example we use to illustrate the concept is the saturation of an input signal seen in Figure 10 using a comparator function on the C6x.

Figure 10. Decision Execution Example (Comparator)



In this example system, the analog signal is converted to digital, resulting in a 16-bit unsigned value.

If the voltage is > 0 V (that is, > 8000h), it will be saturated to the maximum positive unsigned value of 0ffffh by the C6x.

If the voltage is < 0 V (that is, > 8000h) it will be saturated to the minimum negative unsigned value of 0ffffh by the C6x.

The digital signal is then converted to analog.

This example is nothing fancy but does allow us to compare the two styles of decision execution.

The more classical “bit test and branch” is shown in Figure 11, implemented in C6x assembly with a conditional branch instruction (called BCND on other TMS320 DSPs).

Figure 11. Decision Execution Code (Bit Test/Branch)

Bit Test/Branch

- Every instruction is conditional
  - **Instead** of the typical "bit test and branch" with much pipeline overhead (using registers for >5 bit constants):

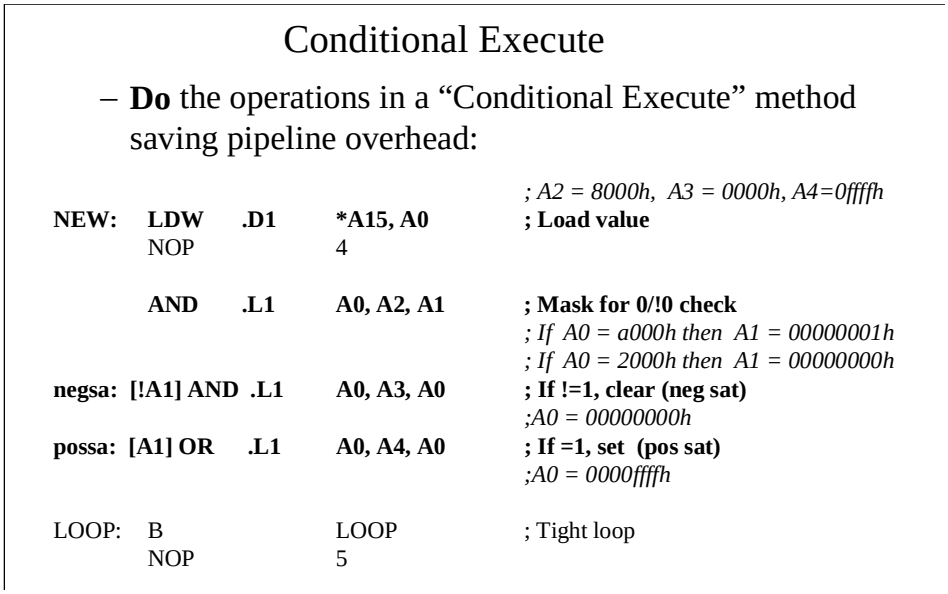
```
                                                ; A2 = 8000h,  A3 = 0000h, A4=0ffffh
OLD:    LDW     .D1     *A15, A0        ; Load value
        NOP             4
        CMPGT   .L1     A0, A2, A1      ; Test if greater than A2 (8000h)
                                        ; If  A0 = a000h then  A1 = 00000001h
                                        ; If  A0 = 2000h then  A1 = 00000000h
[A1]    B               possat          ; If so, branch to set (pos sat)
        NOP             5

negsat: AND     .L1     A0, A3, A0      ; If not, fall thru to clear (neg sat)
        B               LOOP            ;A0 = 00000000h
        NOP             5

possat: OR      .L1     A0, A4, A0      ; If so, set  (pos sat)
        B               LOOP            ;A0 = 0000ffffh
        NOP             5                                       15
```

Note that some values have been pre-loaded into A2, A3, and A4 so that the register operation may be used for 16 bits (For brevity we have omitted every MVK/MVKH seen in previous examples). The data is LDWed (into register A0) and tested (with register A2) to see if it is > 8000h. The result is written to the A1 register and the branch is conditioned on [A1] (other TMS320 DSPs have a specific "branch conditional" instruction).

If the value is > 8000h, the code branches to possat: and positively saturates the value. If the value is ≤ 8000h, it falls through the branch to negsat: and negatively saturates the value. Often one can design the code so the condition that statistically may happen more often will fall through, although this is not so applicable, if you are comparing sine waves.

You can count the cycles to execute the loop once as 1(LDW)+ 4(NOP) + 1(AND) + 6(B) + 1(AND/OR) = 13 cycles. Can we improve on this number? The "conditionally execute" method is more conducive to the C6x and is shown in Figure 12.

Figure 12. Decision Execution Code (Conditional Execute)



Note that in this code example also, some values have been pre-loaded into A2, A3, and A4 so that register operation may be used for 16 bits. (Again, for brevity we have omitted every MVK/MVKH seen in previous examples).

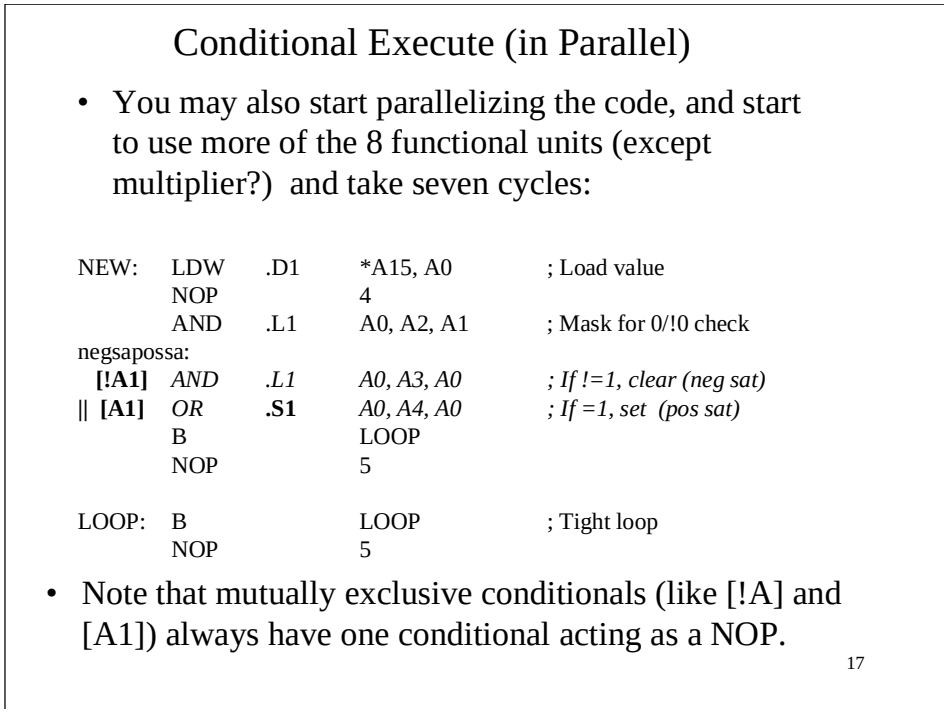
Again the data is LDWed and this time tested by doing an AND operation with the value 8000h that will result in either 0 or !0 in the A1 register. (The reason for using the "and" along with other optimization methods is discussed in the section, *Optimization Methods/Rationales→ASIC/FPGA*. CMPGT would have been just as valid). Then A1 is used as the conditional test for negative or positive saturation, identical to Figure 11. The conditions are mutually exclusive; thus, one is executed while the other one becomes a NOP.

But no branches are needed (The tight loop is just meant to give an end to the example). This code is equivalent to that seen in Figure 11. Let us now think about benchmarking the number of cycles to execute the code. You can count the cycles to execute the loop once as 1(LDW) + 4(NOP) + 1(AND) + 2(AND/OR) = 8 cycles. Again, can we improve on this number?

Optimizing the Code (Parallelism and Unit Utilization)

Examining the code, we see that the positive and negative saturation instructions have no data dependencies between them (for more information on data dependencies, see the *TMS320C6000 Programmer's Guide*, literature number SPRU198). Thus nothing prevents us from executing them at the same time. So we may start optimizing the code now by adding the "||" in the code to perform the negative and positive saturation in the same cycle. Again note that the conditions are mutually exclusive; thus, one is executed while the other becomes a NOP in parallel, as shown in Figure 13.

Figure 13. Decision Execution Code (Conditional Execute in Parallel)

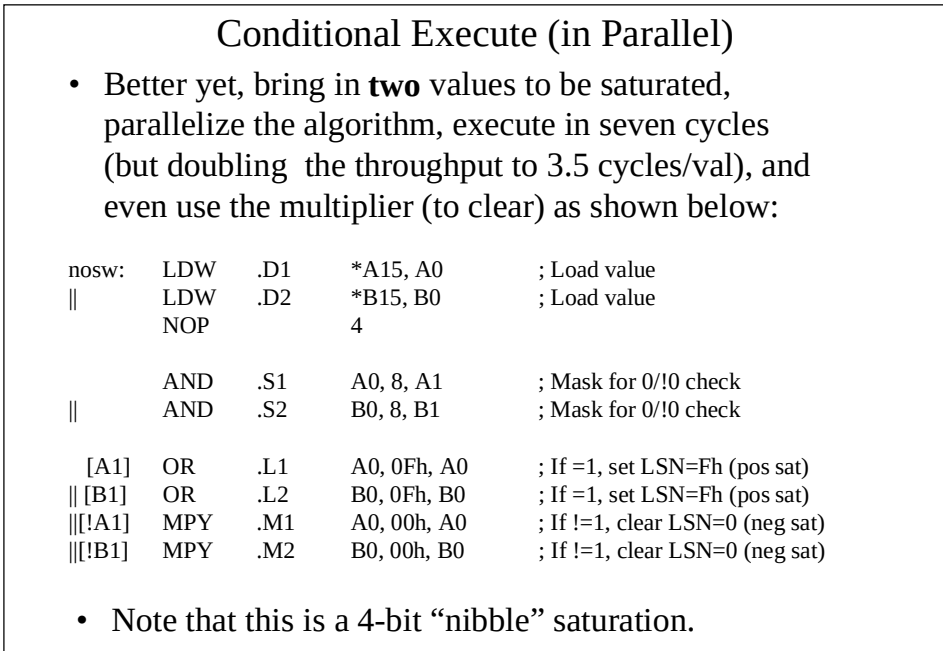


A second issue to be concerned about is unit resources. Because the .L1 unit cannot be used twice in the same cycle, we must also use the .S1 unit, as shown in Figure 13. You can count the cycles to execute the loop once as 1(LDW) + 4(NOP) + 1(AND) + 1(AND/OR) = 7 cycles.

It is interesting to note that the cost of not branching is that one unit becomes a NOP. Thus, you could almost say that instead of losing 6 cycles from Figure 11 (8 units * 6 cycles = 48 potential units), you "lose" only one unit instead of forty-eight.

Now we consider how to parallelize by using more of the units. This is accomplished by bringing in two values at a time, keeping them separate on A and B sides, and executing in parallel. Each of the two .D units can load a value into A0 and B0 respectively in the first cycle and wait the appropriate NOPs. Each of the two .S units can test each of the values and write the result into A1 and B1 registers, respectively, in the sixth cycle. Then we conditionally positively saturate the values using the .L units, and even use a trick to conditionally negatively saturate the values using the .M units (multiply a value by "0" and get "0") in the seventh cycle. See the code in Figure 14.

Figure 14. Decision Execution Code (Conditional Execute in Parallel—II)



Conditional Execute (in Parallel)

- Better yet, bring in **two** values to be saturated, parallelize the algorithm, execute in seven cycles (but doubling the throughput to 3.5 cycles/val), and even use the multiplier (to clear) as shown below:

```
nosw:   LDW     .D1     *A15, A0        ; Load value
||      LDW     .D2     *B15, B0        ; Load value
        NOP             4

        AND     .S1     A0, 8, A1       ; Mask for 0/!0 check
||      AND     .S2     B0, 8, B1       ; Mask for 0/!0 check

  [A1]  OR      .L1     A0, 0Fh, A0     ; If =1, set LSN=Fh (pos sat)
|| [B1]  OR      .L2     B0, 0Fh, B0     ; If =1, set LSN=Fh (pos sat)
||[!A1]  MPY     .M1     A0, 00h, A0     ; If !=1, clear LSN=0 (neg sat)
||[!B1]  MPY     .M2     B0, 00h, B0     ; If !=1, clear LSN=0 (neg sat)
```

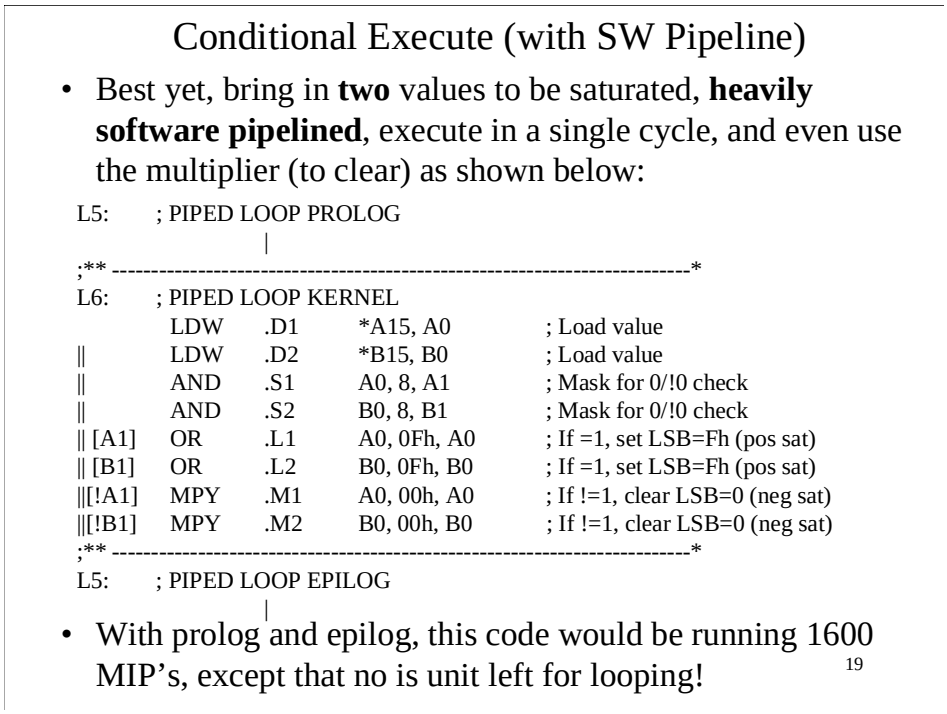
- Note that this is a 4-bit “nibble” saturation.

Technically, the .M units have a latency of 1, so negatively saturated values would not be ready until the eighth cycle. Nevertheless, by counting the “||” combinations, the number of cycles comes to 7 for two values, thus averaging to 7/2 = 3.5 cycles per value.

Note that the example is simplified to doing **nibbles** so we could stick with constants. Using registers is possible, but resource conflicts will start to appear in the Figure 15, if you do not spread accesses among registers.

Finally, if you use software pipelining, the kernel shown in Figure 15 is possible (for more information, see the *TMS320C6000 Programmer’s Guide*, literature number SPRU198).

Figure 15. Decision Execution Code (Conditional Execute—Software Pipelined)



After some prolog to initialize the SW pipeline, the above kernel uses all eight units to execute two samples per cycle. Then some epilog code is often needed to gracefully exit from the kernel.

This method allows two values to be loaded, compared, and saturated in a single cycle, assuming, of course, appropriate prolog and epilog code. This eliminates the "NOP 4" following the "LDW" seen in the previous code examples. Thus, in 50 cycles a theoretical maximum of 100 values could be processed, but with prolog/epilog overhead it is probably more like 55–60 cycles. Thus, the effective benchmark is 1 cycle per 2 values or 0.5 cycles per value.

No .S unit in Figure 15 is available for looping. Thus, there are two ways to repeat this instruction, for example, 100 times. One method is to use a dual-cycle loop that will cause it to take 105–110 cycles (for more information, see the *TMS320C6000 Programmer's Guide*, literature number SPRU198). The second method is to unroll the loop. In other words, repeat/copy it 100 times, if you have the available code space. Thus, the "loop" benchmark remains within 55–60 cycles with a classic code size for speed tradeoff.

Optimization Methods/Rationales→ASIC/FPGA

In the section, *Optimizing the Code (Parallelism and Unit Utilization)*, Figure 11 shows the 8000h test performed using the CMPGT instruction. In Figure 12 through Figure 15, the equivalent test could be and was done using the AND instruction. Such a method was chosen to allow flexibility in later unit allocation for instructions because the CMPGT is only available on the .L units. Because the AND is available on the 2 .L units and the 2 .S units, using this equivalent test makes later flexibility in allocation of units possible.

The C compiler uses a similar trick when it tests for a value being equal to something. Say (if i==5) could be tested by subtracting 5 from the variable for i and testing for 0!/0. Because this operation is available on 6 of the 8 units (.L, .S, and .D's), it gives greater flexibility in unit allocation.

The feature of the "conditionally executes" allowing for NOP (such as mutually exclusive conditions) when an operation is not to be performed and the feature of parallelism in the C6x architecture offers an interesting observation. This architecture allows operation of the C6x in a sequential execution mode with very little branches and many conditionals, similar to the dataflow seen in an FPGA or ASIC. These functions could run in lockstep at very fast speeds but are now much easier to program/route than when implemented on an FPGA/ASIC.

Decision Execution Cycle Summary

Thus, to summarize the cycle savings in Table 1 (please bear with the relative levels of optimization that were presented to academically get the concepts across):

Table 1. Execution Decision Cycle Summary

Coding Style	Cycles
Bit test and branch (Figure 11)	13
Conditionally execute (Figure 12)	8
Conditionally execute with parallel saturate (Figure 13)	7
Dual value conditionally execute with parallel saturate (Figure 14)	3.5
Software pipelined dual value conditionally execute with parallel saturate (Figure 15)	~0.5

Now that we have seen specifics of the heart of the C6x architecture in assembly, let us see how the advanced C6x tools can help make using this architecture easier.

Application Example

In the section, *C6x CPU/Instruction Features With Code Examples*, we examined various specific features of the C6x architecture, albeit all written in assembly. Often a programmer, especially starting out, does not want to get involved in the intricacies of a certain CPU's assembly language. Thus, they write in ANSI C to produce portable, general code.

There are various code optimization levels between ANSI C and pure assembly (intrinsics, C callable assembly, etc.) that will be fully explored with benchmarks in a future application report with code and benchmarks.

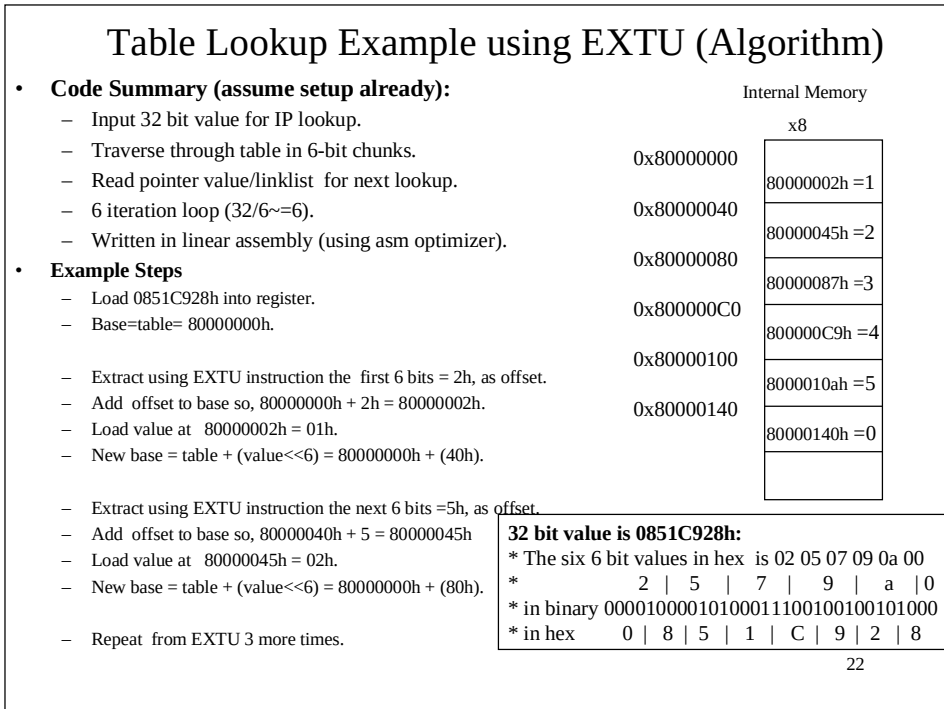
In this section, we write in something unique to the C6x called "linear assembly" and run through a code-generation tool called the "assembly optimizer" (for more information, see the *TMS320C6000 Optimizing C Compiler User's Guide*, literature number SPR187). The presented example is just a first pass of a non-traditional application.

Table Lookup Example Description

We examine a certain networking lookup algorithm implemented on a C6x as it is implemented on a TNETX15VE address lookup engine. Of course, additional optimizations are possible in hand assembly, but we use the assembly optimizer tool to accomplish some of the functions we have mentioned.

The algorithm is explained in Figure 16. The full code is listed in Appendix A.

Figure 16. Table Lookup Example Description



The code summary gives an overview of what the code does, while the example steps go through the contrived actual data value used. The actual data value is displayed in the lower right-hand box in hex, binary, and 6-bit values coded in hex. The table, hard-coded in internal memory, is displayed on the upper right side of the graphic. The specific initialized values (along with their addresses) used in this contrived example are displayed in the boxes and not to scale.

This example shows the use of the EXTU instruction. It is assumed that the lookup table is built, and the code and benchmarks apply to processing of one 32-bit value.

The algorithm ended up being a six-iteration loop. Loops are obviously good for DSPs. More iterations would be helpful but would require buffering up of much more data on the system level (up to 2K bytes per IP packet). Or in other words, 2K bytes of buffer space per six iterations of the loop are needed. Thus, to do a thousand iterations, you would need (1000/6) * 2K = 333K bytes, which may be prohibitive on some systems.

Table Lookup Example Code

The initialization code was written in C (as all initialization code should be) and the actual lookup function could be ANSI C, C with intrinsics, linear assembly, or pure assembly. Figure 17 shows both the main C code and the beginning of the called linear assembly function named “iploop” (The code in Figure 17 actually resides in two separate files. The C code is in a “.c” file. The linear assembly is in a “.sa” file that stands for “serial assembly”.)

Figure 17. Table Lookup Example Code Initialization

```
main()
{

// Init pointer and data
int *llptr;
int data = 0x0851C928;
//Assign to 0x80000000 (reserved in linker - bad programming practice) and call .SA
llptr = (int *) 0x80000000;
ipploop (llptr, data);

} /* end main */

*****************************************************************
_ipploop:.cprocllptr, data

.regcount, cstal, cstbr, cstfinal
.reg base, offset

mvk    06, count; init cnount
mvk0,  cstal; init shift
mvk    26,  cstbr; val for EXT
mvk0, base ; init base
```

A called linear assembly function from the calling C function resembles any C function with passable parameters and return value. The top half shows C code that hard-codes a pointer at internal memory location 0x80000000 (and allocates memory using the linker) with a pointer. Then the function is called, as any C function, with passed parameter of the pointer and the data value.

When using .cproc, called linear assembly function understands passed parameters from the C calling function for use in the linear assembly function. The bottom half shows the linear assembly function in a .sa file and how the parameters are received and used as it was a C function along with some initializations.

Figure 18 shows the iploop() function written in linear assembly that appears in the same .sa file shown in Figure 17 (for more information, see the *TMS320C6000 Optimizing C Compiler User's Guide*, literature number SPR187).

Figure 18. Primary Lookup Table Loop in Linear Assembly

```
                    ; build EXT par
loop:               shl     cstal, 5, cstfinal
                    add     cstal, cstfinal, cstfinal; annoying for CPU
                    add     cstbr, cstfinal, cstfinal
                    add     llptr, base, base     ; add new base with llptr

                    extu    data, cstfinal, offset; get offset
                    add     base, offset, base    ; add to base
                    ldb     *base, offset                 ; next offset

                    ; update base
                    shl     offset, 6, base       ; new offset->base

                    ;increment cstal and cstbr
                    add     cstal, 6, cstal
                    sub     cstbr, 6, cstbr

              [count] sub   count, 1, count
              [count] b     loop

                    .return count
```

Linear assembly allows the use of C6x mnemonics with symbolic (including C passed parameter) values. It was written as “you think it” without optimization or software pipelining. Figure 18 shows the meat of the code. The EXTU instruction is the meat of the loop. It extracts the 6 bits from the data value as the offset and looks up the new base for the next table location. EXTU is used dynamically and cstal and cstbr variables specify which bits to extract. They are pasted together and put into register cstfinal at the beginning of the loop and updated toward the end. A loop counter operation is needed and seen as the last two lines of code before the return. Note that the return value merely confirms that the loop was executed.

Now after the code is run through the assembly optimizer, pure assembly is automatically generated. Figure 19 shows the kernel of optimized assembly that would reside in an “.asm” file. Note that epilog and prolog have been omitted for brevity and that little time was spent optimizing any of this by looking at data dependencies.

Figure 19. Primary Lookup Table Loop in Pure Assembly

```
L6:      ; PIPED LOOP KERNEL

         ADD    .L1    A0,A5,A5    ; add new base with llptr
||       EXTU   .S1    A4,A6,A6    ; get offset

  [ B0]  SUB    .L2    B0,0x1,B0   ;
||       ADD    .L1    A5,A6,A5    ; add to base

  [ B0]  B      .S2    L6          ;
||       LDB    .D1    *A5,B4      ; next offset

         NOP          1
         ADD    .L1    0x6,A7,A7   ;
         SHL    .S1    A7,0x5,A6   ;@

         SUB    .S1    A3,0x6,A3   ;
||       ADD    .L1    A7,A6,A6    ;@ annoying for CPU

         SHL    .S1X   B4,0x6,A5   ; new offset->base
||       ADD    .L1    A3,A6,A6    ;@
```

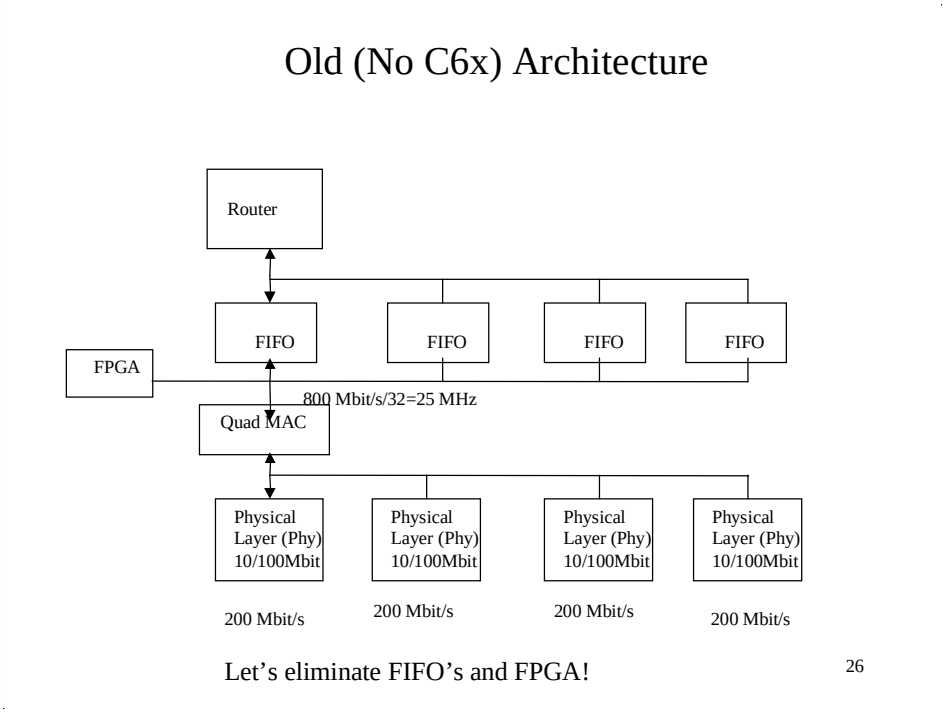
Thus, Figure 19 shows the assembly optimizer generated assembly code as a software-pipelined kernel. You did not have to think about software pipelining or optimization because it was done for you. The code clearly shows the number of cycles required. You can count the 8 cycles and see the data dependencies follow the sets of parallel bars.

System Discussion—C6x DMAs for Data I/O (Eliminate Components)

As mentioned earlier, the C6x CPU has certain architectural features that make it powerful for operation in “dataflow” applications. In addition, the C6x DMA provides an efficient configuration for bringing data on-chip and taking data off-chip without much CPU overhead. Also, C6x internal memory allows the elimination of expensive external device I/Os, such as FIFOs.

A networking data mover is a typical example shown in Figure 20.

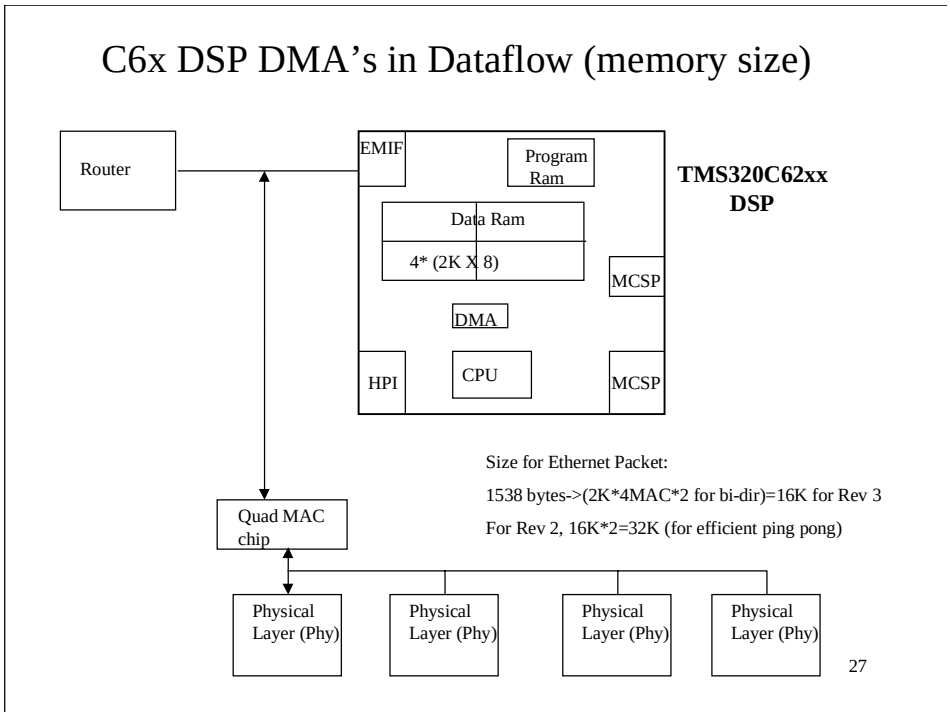
Figure 20. Old (No C6x) Architecture



In this networking example, the physical layer (PHY) is akin to a speech codec in a typical DSP system. The media access controller (or MAC) receives the digital data from the Ethernet wire (as the DSP would) and has it sent to the router (which you could imagine is like a host, but much faster) through the FIFOs by the FPGA. Everything is running fast with many parts in a bi-directional manner.

The maximum size for an Ethernet packet is 1538 bytes. Figure 21 shows how a C6x can be substituted for the FIFOs in Figure 20.

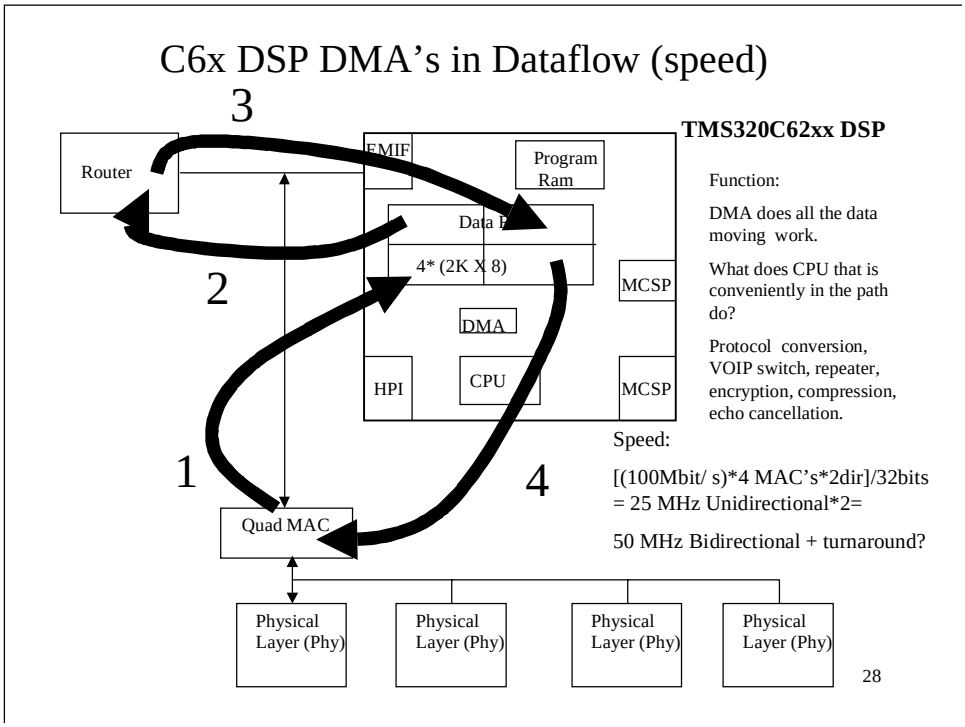
Figure 21. C6x Architecture (Size)



The C6x internal memory is able to replace the FIFOs. Size-wise there is easily enough internal memory for an entire maximum Ethernet packet of size 1538 bytes to fit into each direction (discussed in Figure 22) for a total of 16K for C6201B silicon. Because the internal memory is not as well partitioned on the C6201B silicon, doubling the buffer size with a ping-pong approach would cause less CPU/DMA conflicts.

Figure 22 shows how the DMA/EMIF replaces the FPGA and addresses the speeds and bandwidths necessary for the system operation.

Figure 22. C6x Architecture (Speed)

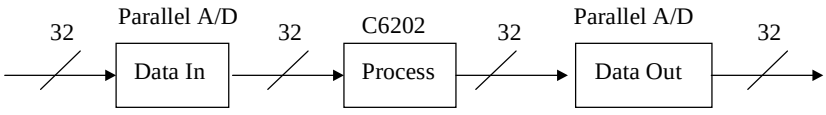


The four DMA channels give an elegant solution for each direction to each of the two "ports" that the C6x is hooking up to. Speed-wise, the C6x might have trouble keeping up due to presently uncharacterized "bus turnaround" issues in a bi-directional manner.

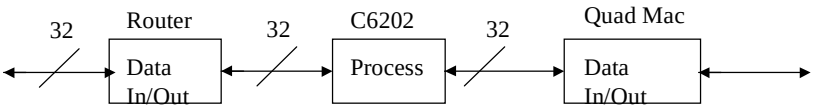
To enhance the discussion, if we modify the C6x to have a second bus as we have in the C6202, some enhancements to the system architecture can be made, as shown in Figure 23.

Figure 23. C6202 Architecture With a Second Bus

- Second parallel bus (to the EMIF) would speed up **uni-directional systems** by eliminating any "bus turnaround" overhead. Each bus handles a direction.



- Second parallel bus (to the EMIF) would simplify **bi-directional systems** interface logic providing a second "port" for parallel access.



A second parallel bus now adds some major advantages to system interfacing not only in reducing the bandwidths by two but also

- On the simpler uni-directional system, there is no bus turnaround overhead because one side is writing and one side is reading.
- For the more complex bi-directional system, the second bus provides a second “port” for parallel access and simpler decode (see Figure 23).

The latter looks like the router described in this section.

Conclusion

The TMS320C6x CPU/architecture has a variety of features attractive for non-typical DSP functions, especially in a dataflow/”virtual FPGA”-type architecture. It may be preferable to write much of this code in linear assembly because the C6x C compiler does not yet comprehend all these features. The C6x four-channel DMA provides an attractive architecture for such dataflow applications (the second parallel bus is appropriate for uni- and bi-directional applications).

Appendix A. Table Lookup Code

The following code is used in the example described in the section, *Application Example*.

The following command lines were used to invoke the C6x tools:

```
cl6x -s -g -k  -ml ipp.c
cl6x  -g -k ipploop.sa
asm6x -s -g  ipptab.asm
lnk6x ipp.cmd
```

ipp.c

```
#include <c:\dsp\c6x\c6xc\include\stdio.h>
#include <c:\dsp\c6x\c6xc\include\stdlib.h>
#include <c:\dsp\c6x\c6xc\include\string.h>
#include <c:\dsp\c6x\c6xc\include\ctype.h>

extern void ipploop();

main()
{
                  int *llptr;
                  int data = 0x0851C928;
                  /* Could hack by making this 0x80000000 in simulator */
                  /* But this works */
                  llptr = (int *) 0x80000000;
                  ipploop (llptr, data);
} /* end main */
```

iploop.sa

```
*****************************************************************
* Texas Instruments, Inc.
*
* Linear Assembly to perform the IP Packet Parsing
*
* Executive Author: David A. Alter, PhD.
*
```

```
* Author: Joe George
*
* Date: 02/02/98
*
* Description:
*                  Parse 32 bit IP header in 6 bit chunks
*
* Requirements:
*                  Table to parse
*
*
* Parameters:
*                     llptr
*
* Return:
*                     1 if it finds it; 0 if it doesn't
*
*****************************************************************

                     .def  _ipploop

_ipploop:          .cproc llptr, data

                     .reg  count, cstal, cstbr, cstfinal
                     .reg   base, offset

                     mvk    06, count   ; init cnount
                     mvk0,  cstal ; init shift
                     mvk    26,  cstbr  ; val for EXT
                     mvk0, base   ; init base


                     ; build EXT par
loop:                shlcstal, 5, cstfinal
                     addcstal, cstfinal, cstfinal; annoying for CPU
                     addcstbr, cstfinal, cstfinal
                     addllptr, base, base  ; add new base with llptr
```

```
                    extu  data, cstfinal, offset; get offset
                    addbase, offset, base ; add to base
                    ldb*base, offset      ; next offset

                    ; update base
                    shloffset, 6, base    ; new offset->base

                    ;increment cstal and cstbr
                    addcstal, 6, cstal
                    subcstbr, 6, cstbr

   [count]          subcount, 1, count
   [count]          b  loop

                    .return count

                    .endproc
```

ipp.cmd

```
/*****************************************************************/
/*  lnk.cmd   v1.00                                              */
/*  Copyright (c) 1996-1997  Texas Instruments Incorporated       */
/*****************************************************************/
-c
-heap  0x2000
-stack 0x0800

/* Link Command file for EVM test code          */

-o ipp.out
-m ipp.map

ipp.obj ipploop.obj ipptab.obj
```

```
-l c:\dsp\c6x\c6xc\lib\rts6201.lib

/* Map 1 */
MEMORY
{
    VECS: o = 00000000h l = 00400h /* reset & interrupt vectors   */
    PMEM: o = 00000400h l = 0FC00h /* intended for initialization */
    LTABLE0: o = 80000000h l = 0003Fh /* 1/2 the DM for table    */
    LTABLE1: o = 80000040h l = 0003Fh /* 1/2 the DM for table     */
    LTABLE2: o = 80000080h   l = 0003Fh /* 1/2 the DM for table   */
    LTABLE3: o = 800000C0h   l = 0003Fh /* 1/2 the DM for table    */
    LTABLE4: o = 80000100h   l = 0003Fh /* 1/2 the DM for table    */
    LTABLE5: o = 80000140h   l = 0003Fh /* 1/2 the DM for table    */
    BMEM: o = 80008000h l = 08000h /*.bss, .system, .stack, cinit */
}

SECTIONS
{
    vectors     >       VECS
    .text       >       PMEM
    lnktable0  >       LTABLE0
    lnktable1  >       LTABLE1
    lnktable2  >       LTABLE2
    lnktable3  >       LTABLE3
    lnktable4  >       LTABLE4
    lnktable5  >       LTABLE5
    .tables     >       BMEM
    .data       >       BMEM
    .stack         >  BMEM
    .bss           >  BMEM
    .sysmem        >  BMEM
    .cinit         >  BMEM
    .const         >  BMEM
    .cio        >       BMEM
    .far           >  BMEM
      }
```

ipploop.asm (tool generated)

```
;****************************************************************
;* TMS320C6x ANSI C Codegen                                  Version
1.10 *
;* Date/Time created: Mon Feb 23 14:12:41 1998                   *
;****************************************************************

;****************************************************************
;* GLOBAL FILE PARAMETERS                                      *
;*                                                             *
;*   Architecture    : TMS320C6200                               *
;*   Endian          : Little                                    *
;*   Memory Model    : Small                                      *
;*   Redundant Loops : Enabled                                   *
;*   Pipelining      : Enabled                                   *
;*   Debug Info      : Debug                                     *
;*                                                             *
;****************************************************************

FP                .set  A15
DP                .set  B14
SP                .set  B15

                  .file "ipploop.sa"



*****************************************************************
* Texas Instruments, Inc.     *
* Linear Assembly to perform the IP Packet Parsing
*
* Executive Author: David A. Alter, PhD.
*
* Author: Joe George
*
* Date: 02/02/98
*
```

```
* Description:
*                  Parse 32 bit IP header in 6 bit chunks
*
* Requirements:
*                  Table to parse
*
*
* Parameters:
*                     llptr
*
* Return:
*                     1 if it finds it; 0 if it doesn't
*
*****************************************************************

                     .def  _ipploop
                  .sect ".text"
                  .align 32
                  .sym  _ipploop,_ipploop,36,2,0
                  .func 30

;*****************************************************************
;* FUNCTION NAME: _ipploop                                       *
;*                                                               *
;*   Regs Modified     : A0,A1,A3,A4,A5,A6,A7,B0,B4,B5           *
;*   Regs Used         : A0,A1,A3,A4,A5,A6,A7,B0,B3,B4,B5
*
;*****************************************************************
_ipploop:
;** --------------------------------------------------------------*
;
; _ipploop:        .cproc llptr, data
;                     .reg  count, cstal, cstbr, cstfinal
;                     .reg   base, offset
                  .sym  llptr,0,4,4,32
                  .sym  data,4,4,4,32
                  .line 1
```

```
           MV      .L1X    B4,A4
||         MV      .S1     A4,A0

                   .sym  count,16,4,4,32
                   .sym  cstal,7,4,4,32
                   .sym  cstbr,3,4,4,32
                   .sym  cstfinal,6,4,4,32
                   .sym  base,5,4,4,32
                   .sym  offset,6,4,4,32
                   .line 7
           MVK     .S2     0x6,B0       ; init cnount
                   .line 8
           MVK     .S1     0x0,A7       ; init shift
                   .line 9
           MVK     .S1     0x1a,A3      ; val for EXT
                   .line 10
           MVK     .S1     0x0,A5       ; init base
           CMPGTU  .L1X    B0,1,A1
   [ A1]   B       .S1     L4
           NOP             5
           ; BRANCH OCCURS
;** --------------------------------------------------------------*
loop:
                   .line 14
           SHL     .S1     A7,0x5,A6
                   .line 15
           ADD     .L1     A7,A6,A6     ; annoying for CPU
                   .line 16
           ADD     .L1     A3,A6,A6
                   .line 17
           ADD     .L1     A0,A5,A5     ; add new base with llptr
                   .line 19
           EXTU    .S1     A4,A6,A6     ; get offset
                   .line 20
           ADD     .L1     A5,A6,A5     ; add to base
                   .line 21
           LDB     .D1     *A5,A6       ; next offset
```

```
           NOP             4
                   .line 24
           SHL     .S1     A6,0x6,A5    ; new offset->base
                   .line 27
           ADD     .L1     0x6,A7,A7
                   .line 28
           SUB     .L1     A3,0x6,A3
                   .line 30
   [ B0]   SUB     .L2     B0,0x1,B0
                   .line 31
   [ B0]   B       .S1     loop
           NOP             5
           ; BRANCH OCCURS
;** --------------------------------------------------------------*
           B       .S1     L10
           NOP             5
           ; BRANCH OCCURS
;** --------------------------------------------------------------*
L4:
           MVC     .S2     CSR,B5
           AND     .L2     -2,B5,B4

           MVC     .S2     B4,CSR
||         SUB     .L2     B0,1,B0

;** --------------------------------------------------------------*
L5:        ; PIPED LOOP PROLOG
           SHL     .S1     A7,0x5,A6    ;
           ADD     .L1     A7,A6,A6     ; annoying for CPU
           ADD     .L1     A3,A6,A6     ;
;** --------------------------------------------------------------*
L6:        ; PIPED LOOP KERNEL

           ADD     .L1     A0,A5,A5     ; add new base with llptr
||         EXTU    .S1     A4,A6,A6     ; get offset

   [ B0]   SUB     .L2     B0,0x1,B0    ;
```

```
||         ADD     .L1     A5,A6,A5     ; add to base

   [ B0]   B       .S2     L6           ;
||         LDB     .D1     *A5,B4       ; next offset

           NOP             1
           ADD     .L1     0x6,A7,A7    ;
           SHL     .S1     A7,0x5,A6    ;@

           SUB     .S1     A3,0x6,A3    ;
||         ADD     .L1     A7,A6,A6     ;@ annoying for CPU

           SHL     .S1X    B4,0x6,A5    ; new offset->base
||         ADD     .L1     A3,A6,A6     ;@

;** --------------------------------------------------------------------*
L7:        ; PIPED LOOP EPILOG

           ADD     .L1     A0,A5,A5     ;@ add new base with llptr
||         EXTU    .S1     A4,A6,A6     ;@ get offset

           ADD     .L1     A5,A6,A5     ;@ add to base
           LDB     .D1     *A5,B4       ;@ next offset
           NOP             1
           ADD     .L1     0x6,A7,A7    ;@
           NOP             1
           SUB     .S1     A3,0x6,A3    ;@
           SHL     .S1X    B4,0x6,A5    ;@ new offset->base
;** --------------------------------------------------------------------*
           MVC     .S2     B5,CSR
                  .line 33
           B       .S1     L10
           NOP             5
           ; BRANCH OCCURS
;** --------------------------------------------------------------------*
;** --------------------------------------------------------------------*
L10:
```

```
                .line 35
          B     .S2     B3
          NOP           4
          MV    .L1X    B0,A4
          ; BRANCH OCCURS
                .endfunc  64,000000000h,0


;               .endproc
```

ipptab.asm

```
*===============================================================
*
*               TEXAS INSTRUMENTS, INC.
*
*
*               Revision Data: 04/22/97
*
*               USAGE This table is IP Packet
*
*
*               Table
*
*===============================================================
                .global ippacket
                .sect data

ippacket:

*  As 6 bit values in hex the value is 02 05 07 09 0a 00?
*                    2  |  5  |  7  |  9  |a  | 3?
p1:             .word 00001000010100011100100100101000
* in hex          0   8 5   1 C   9 2   8

                .sect "lnktable0"
t000:           .byte 00h
                .byte 00h
```

```
                .byte 01h; Packet val at 2
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h

                .sect "lnktable1"
t001:           .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 02h; Packet val at 5
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h
                .byte 00h

                .sect "lnktable2"
t002:           .byte 00h
                .byte 00h
```

```
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 03h; Packet val at 7
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h

                    .sect "lnktable3"
t003:               .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 04h; Packet val at 9
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h

                    .sect "lnktable4"
t004:               .byte 00h
                    .byte 00h
                    .byte 00h
```

```
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 05h; Packet val at A
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h

                    .sect "lnktable5"
t005:               .byte 00h; Packet val at
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h
                    .byte 00h

                    .end
```

TI Contact Numbers

INTERNET

TI Semiconductor Home Page
www.ti.com/sc

TI Distributors
www.ti.com/sc/docs/distmenu.htm

PRODUCT INFORMATION CENTERS

Americas
Phone +1(972) 644-5580
Fax +1(972) 480-7800
Email sc-infomaster@ti.com

Europe, Middle East, and Africa
Phone
Deutsch +49-(0) 8161 80 3311
English +44-(0) 1604 66 3399
Español +34-(0) 90 23 54 0 28
Francais +33-(0) 1-30 70 11 64
Italiano +33-(0) 1-30 70 11 67
Fax +44-(0) 1604 66 33 34
Email epic@ti.com

Japan
Phone
International +81-3-3344-5311
Domestic 0120-81-0026
Fax
International +81-3-3344-5317
Domestic 0120-81-0036
Email pic-japan@ti.com

Asia
Phone
International +886-2-23786800
Domestic
Australia 1-800-881-011
TI Number -800-800-1450
China 10810
TI Number -800-800-1450
Hong Kong 800-96-1111
TI Number -800-800-1450
India 000-117
TI Number -800-800-1450
Indonesia 001-801-10
TI Number -800-800-1450
Korea 080-551-2804
Malaysia 1-800-800-011
TI Number -800-800-1450
New Zealand 000-911
TI Number -800-800-1450
Philippines 105-11
TI Number -800-800-1450
Singapore 800-0111-111
TI Number -800-800-1450
Taiwan 080-006800
Thailand 0019-991-1111
TI Number -800-800-1450
Fax 886-2-2378-6808
Email tiasia@ti.com

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE (“CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty, or endorsement thereof.