Application Report
SPRA678 - June 2000

Getting the Most Performance When Porting TMS320C62x Code to the TMS320C64x Platform

Jackie Brenner
DSP Applications

ABSTRACT

The TMS320C64x™ DSP is the next generation of the C6000™ DSP family and is object code compatible with the TMS320C62x™ DSP. Some key extensions have been made to the C62x™ VelociTI™ architecture to allow the C64x™ DSP to perform more work each clock cycle. These extensions include wider data paths, a larger register file, greater orthogonality and new instructions that support packed data processing. The fact that the C64x is object code compatible with the C62x allows you to run existing C62x code on the C64x without recompiling. However, existing code often does not take full advantage of the architectural enhancements found in the C64x. There are various techniques that the C6000 programmer can employ to gain the maximum benefit of these architectural extensions. These techniques include modifying compiler options to better suit the C64x generation, providing more information to the compiler about the alignment of pointer variables and utilizing intrinsics to fully leverage the packed data processing capabilities of the C64x platform. This application brief discusses the techniques described above and examines the code generated by the C6000 compiler to demonstrate how to get the most performance from C62x code ported to the C64x DSP.

Contents

1 Object Code Compatibility **2**

2 Leveraging the Architectural Enhancements **2**

2.1 Architectural Considerations 2
2.2 Register File Cross Paths 2
2.3 Avoiding Cross Path Stalls: Weighted Vector Sum Example 4
2.4 Using Enhanced Instructions/Avoiding Non-Aligned Accesses: Dot Product Example 7
2.5 Applying Packed Data Processing: Clear Below Threshold Example 9
2.6 Larger Register File and Increased Orthogonality 13

3 Conclusion **18**

4 References **18**

List of Figures

Figure 1. C64x Data Paths and Data Cross Paths 3

List of Tables

Table 1. Quad 8-bit and Dual 16-bit Instruction Set Extensions 10

TMS320C64x, C6000, TMS320C62x, VelociTI, C62x, and C64x are trademarks of Texas Instruments.

1 Object Code Compatibility

The C64x is the next generation of the C6000 DSP family and is object code compatible with the C62x DSP. Some key extensions have been made to the C62x VelociTI architecture to allow the C64x to perform more work each clock cycle. These extensions include wider data paths, a larger register file, greater orthogonality and new instructions that support packed data processing. The fact that the C64x is object code compatilbe with the C62x allows you to run existing C62x code on the C64x without recompiling. However, existing code does not take full advantage of the architectural enhancements found in the C64x.

2 Leveraging the Architectural Enhancements

There are various techniques that the C6000 programmer can employ to gain the maximum benefit of these architectural extensions. These techniques include modifying compiler options to better suit the C64x generation, providing more information to the C6000 compiler about the alignment of pointer variables and utilizing intrinsics to fully leverage the packed data processing capabilities of the C64x platform. These techniques can be utilized to get the most performance from C62x code ported to the C64x DSP.

2.1 Architectural Considerations

Before discussing optimization techniques, let's take a brief glimpse at the C6000 architecture.

The C6000 CPU components consist of:

- Two general-purpose register files (A and B)
- Eight functional units (.L1, .L2, .S1, .S2, .M1, .M2, .D1, and .D2)
- Two load-from-memory data paths (LD1 and LD2)
- Two store-to-memory data paths (ST1 and ST2)
- Two data address paths (DA1 and DA2)
- Two register file data cross paths (1X and 2X)

2.2 Register File Cross Paths

The functional unit is where the instructions (**ADD**, **MPY** etc.) are executed. Each functional unit reads directly from and writes directly to the register file within its own data path. That is, the .L1, .S1, .D1, and .M1 units write to register file A and the .L2, .S2, .D2, and .M2 units write to register file B.

The register files are also connected to the opposite-side register file's functional units via the 1X and 2X cross paths. These cross paths allow functional units from one data path to access a 32-bit operand from the opposite side's register file. The 1X cross path allows data path A's functional units to read their source from register file B. Similarly, the 2X cross path allows data path B's functional units to read their source from register file A.

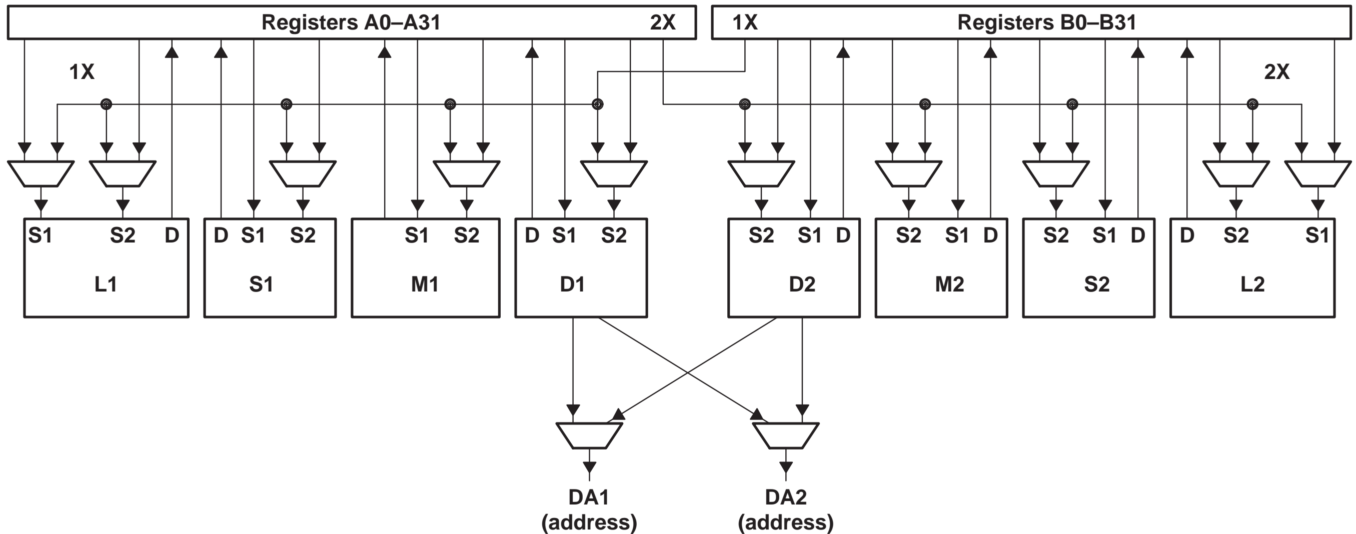

Figure 1. C64x Data Paths and Data Cross Paths

On the C64x, all eight of the functional units have access to the opposite side's register file via a cross path. Only two cross paths, 1X and 2X, exist in the C6000 architecture. Therefore, the limit is one source read from each data path's opposite register file per clock cycle, or a total of two cross-path source reads per clock cycle. The C64x pipelines data cross path accesses allowing multiple functional units per side to read the same cross-path source simultaneously. Thus, the cross path operand for one side may be used by up to two of the functional units on that side in an execute packet. In the C62x/C67x, only one functional unit per data path, per execute packet can get an operand from the opposite register file.

On the C64x, a delay clock cycle is introduced whenever an instruction attempts to read a source register via a cross path where that register was updated in the previous cycle. This is known as a cross path stall. This stall is inserted automatically by the hardware; no NOP instruction is needed. (See the *C6000 CPU and Instruction Set Reference Guide,* literature number SPRU189E, for further information). This cross path stall does not occur on the C62x/C67x. This cross path stall is necessary so that the C64x can achieve clock rate goals beyond 1GHz. It should be noted that all code written for the C62x/C67x that contains cross paths where the source register was updated in the previous cycle will contain one clock stall when running on the C64x. The code will still run correctly but it will take an additional clock cycle.

It is possible to avoid the cross path stall by scheduling instructions such that a cross path operand is not read until at least one clock cycle after the operand has been updated. With appropriate scheduling, the C64x can provide one cross path operand per data path per clock cycle with no stalls. In many cases, the TMS320C6000 Optimizing C Compiler and Assembly Optimizer automatically perform this scheduling as demonstrated in the example that follows.

2.3 Avoiding Cross Path Stalls: Weighted Vector Sum Example

Below is a C implementation of a weighted vector sum. Each value of input array ***a*** is multiplied by a constant, ***m***, and then is shifted to the right by 15 bits. This weighted input is now added to a second input array, ***b***, with the weighted sum stored in output array, ***c***.

```
int w_vec(short a[],short b[], short c[], short m, int n)
               {int i;
                       for (i=0; i<n; i++) {
                           c[i] = ((m * a[i]) >> 15) + b[i];
                                             }
               }
```

This algorithm requires two loads, a multiply, a shift, an add and a store. Only the .D units on the C6000 architecture are capable of loading/storing values from/to memory. Since there are two .D units available it would appear this algorithm would require two cycles to produce one result since three .D operations are required. Note, however, that the input and output arrays are short or 16-bit values. Both the C62x and C64x have the ability to load/store 32–bits per .D unit. (The C64x can also load/store 64 bits per .D unit). It should therefore be possible to produce two 16-bit results every two clock cycles by unrolling the loop once.

To explore this further, let's look at a partitioned linear assembly version of the weighted vector sum where data values are brought in 32 bits at a time. With linear assembly, we do not need to specify registers, functional units or delay slots. In partitioned linear assembly, we can specify what side of the machine we would like to see the instructions execute on. We can further specify the functional unit as seen below.

```
        .global _w_vec
_w_vec: .cproc  a, b, c, m
        .reg    ai_i1, bi_i1, pi, pi1, pi_i1, pi_s, pi1_s
        .reg    mask, bi, bi1, ci, ci1, c1, cntr
        MVK     -1, mask
        MVKH    0, mask       ; generate a mask = 0x0000FFFF
        MVK     50, cntr      ; load loop count with 50
        ADD     2, c, c1      ; c1 is offset by 2(16-bit values)from c

LOOP:   .trip 50              ; this loop will run a minimum of 50 times

      LDW    .D2    *a++,ai_i1            ;load 32-bits (an & an+1)
      LDW    .D1    *b++,bi_i1            ;load 32-bits (bn & bn+1)
      MPY    .M1    ai_i1, m, pi     ;multiply an by a constant ; prod0
      MPYHL  .M2    ai_i1, m, pi1    ;multiply an+1 by a constant; prod1
      SHR    .S1    pi, 15, pi_s     ;shift prod0 right by 15 -> sprod0
      SHR    .S2    pi1,15, pi1_s    ;shift prod1 right by 15 -> sprod1
      AND    .L2X   bi_i1, mask, bi  ;AND bn & bn+1 w/ mask to isolate bn
      SHR    .S1    bi_i1, 16, bi1  ;shift bn & bn+1 by 16 to isolate bn+1
      ADD    .L2X   pi_s, bi, ci          ;add sprod0 + bn
      ADD    .L1X   pi1_s, bi1, ci1       ;add sprod1 + bn+1
      STH    .D2    ci, *c++[2]          ;store 16-bits (cn)
      STH    .D1    ci1, *c1++[2]        ;store 16-bits (cn+1)
[cntr]SUB           cntr, 1, cntr        ;decrement loop count
[cntr]B             LOOP            ;branch to loop if loop count >0
        .endproc
```

In the implementation above, we bring in two 16-bit values at a time with the **LDW** instruction into a single 32-bit register. We multiply each 16-bit value in register ai_i1 by the short (16-bit) constant ***m***. Each 32-bit product is shifted to the right by 15 bits. The second input array is also brought in two 16-bit values at a time into a single 32-bit register, bi_i1. bi_i1 is **AND**ed with a mask that zeros the upper 16 bits of the register to create bi (a single 16-bit value). bi_i1 is also shifted to the right by 16 bits so that the upper 16-bit input value can be added to the corresponding weighted input value.

The code above is sent to the assembly optimizer with the following compiler options: –o3 –mi –mt –k and –mg. Since we did not specify a specific C6000 platform, the default is to generate code for the C62x. The –o3 option enables the highest level of the optimizer. The –mi option creates code with an interrupt threshold equal to infinity (we are saying interrupts will never occur when this code runs). The –k option keeps the assembly language file and –mt indicates we are assuming no aliasing (aliasing is where multiple pointers can point to the same object). The –mg option allows profiling to occur in the debugger for benchmarking purposes.

Below is the assembly output generated by the assembly optimizer for the weighted vector sum loop kernel:

```
LOOP:    ; PIPED LOOP KERNEL

         AND    .L2X    A3,B6,B8    ;AND bn & bn+1 with mask to isolate bn
||       SHR    .S1     A0,0xf,A0     ; shift prod0 right by 15 -> sprod0
||       MPY    .M1X    B2,A5,A0      ; multiply an by constant ; prod0
|| [ A1] B      .S2     LOOP          ; branch to loop if loop count >0
|| [ A1] ADD    .L1     0xffffffff,A1,A1  ; decrement loop count
||       LDW    .D1T1   *A7++,A3          ; load 32-bits (bn & bn+1)
||       LDW    .D2T2   *B5++,B2          ; load 32-bits (an & an+1)

   [ A2] MPYSU  .M1     2,A2,A2          ;
|| [!A2] STH    .D2T2   B1,*B4++(4)      ; store 16-bits (cn+1)
|| [!A2] STH    .D1T1   A6,*A8++(4)      ; store 16-bits (cn)
||       ADD    .L1X    A4,B0,A6         ; add sprod1 + bn+1
||       ADD    .L2X    B8,A0,B1         ; add sprod0 + bn
||       SHR    .S2     B9,0xf,B0   ; shift prod1 right by 15 -> sprod1
||       SHR    .S1     A3,0x10,A4  ; shift bn & bn+1 by 16 to isolate bn+1
||       MPYHL  .M2     B2,B7,B9    ; multiply an+1 by a constant ; prod1
```

This two-cycle loop produces two 16-bit results per loop iteration as we planned.

But what if we run the ’C62x code “as-is” on the C64x? Notice that in the first execute packet that A0 (prod0) is shifted to the right by 15 and then the result is written back into A0. In the next execute packet and therefore the next clock cycle, A0 (sprod0) is used as a cross path operand to the .L2 functional unit. If this code were run on the C64x, it would exhibit a one cycle clock stall as described above. We are updating A0 in cycle 2 and using it as a cross path operand in cycle 3. Therefore, our two-cycle loop would now take three cycles to execute.

Code Composer Studio™ (CCS) provides a simulator analysis tool that allows us to monitor cross path stalls in the debugger environment. From the debugger window we select the tools menu. Under tools we then select simulator analysis. Next we indicate which analysis event we wish to set a breakpoint on or the event we wish to count. In this case we want to count the instances of cross path forwarding stalls. A debug window opens as depicted below that will show us each time the event occurs and will keep a running sum of the total number of times the event occurred. The tool as it exists today counts the instances of the cross path stall but not the number of clock cycles it causes. The subtle difference is that in an execute packet it is possible to have two cross path stalls, one on the A side and one on the B side. This would be counted as two events but would result in only a single clock delay. Future revisions of CCS will add the capability to count clock cycles.

Simulator analysis is a good way to recognize the presence of the cross path stall but ideally we would like to avoid this stall altogether. The cross path stall can be avoided, in most cases, if we merely add the –mv6400 option to the compiler options list. This option indicates to the compiler/assembly optimizer that the code below will be run on the C64x core.

Below is the assembly output generated by the assembly optimizer for the weighted vector sum loop kernel compiled with the –mv6400 –o3 –mt –mi –k –mg options:

```
LOOP:    ; PIPED LOOP KERNEL

         STH   .D1T1   A6,*A8++(4)  ; store 16-bits (cn)
||       ADD   .L2X    B9,A16,B9    ; add bn + copy of sprod0
||       MV    .L1     A3,A16       ; copy sprod0 to another register
||       SHR   .S1     A5,0x10,A3 ; shift bn & bn+1 by 16 to isolate bn+1
|| [ B0] BDEC  .S2     LOOP,B0    ;branch to loop & decrement loop count
||       MPY   .M1X    B17,A7,A4    ; multiply an by a constant ; prod0
||       MPYHL .M2     B17,B4,B16   ; multiply an+1 by a constant ; prod1
||       LDW   .D2T2   *B6++,B17    ; load 32-bits (an & an+1)

         STH   .D2T2   B9,*B7++(4)  ; store 16-bits (cn+1)
||       ADD   .L1X    A3,B8,A6     ; add bn+1 + sprod1
||       AND   .L2X    A5,B5,B9   ; AND bn & bn+1 with mask to isolate bn
||       SHR   .S2     B16,0xf,B8 ; shift prod1 right by 15 -> sprod1
||       SHR   .S1     A4,0xf,A3  ; shift prod0 right by 15 -> sprod0
||       LDW   .D1T1   *A9++,A5     ; load 32-bits (bn & bn+1)
```

In this case, the assembly optimizer has created a two-cycle loop without a cross path stall. The loop count decrement instruction and the conditional branch to loop based on the value of loop count instruction have been replaced with a single **BDEC** instruction. In the instruction slot created by combining these two instructions into one, a **MV** instruction has been placed. The **MV** instruction copies the value in the source register to the destination register. In this case, the value in A3 (sprod0) is placed into A16. A16 is then used as a cross path operand to the .L2 functional unit. A16 is updated every two cycles. For example, A16 is updated in cycles 2, 4, 6, 8 etc. Furthermore, the value of A16 from the previous loop iteration is used as the cross path operand to the .L2 unit in cycles 2, 4, 6, 8 etc. This rescheduling prevents the cross path stall. So we again have a two-cycle loop with two 16-bit results produced per loop iteration. Further optimization of this algorithm can be achieved by unrolling the loop one more time.

Next we will look at an example that allows us to take advantage of the additional multiply capacity found on the C64x.

Code Composer Studio is a trademark of Texas Instruments.

2.4 Using Enhanced Instructions/Avoiding Non-Aligned Accesses: Dot Product Example

One of the fundamental building blocks of any DSP algorithm be it convolution or filtering is the sum of products equation.

$$Y = \sum_{n=1}^{N} a_n * x_n \tag{1}$$

The two basic instructions in this sum of products equation are multiply and add. We want to multiply an element in the ***a*** array with the corresponding element in the ***x*** array and we want to add that product to a running sum of products as we process the next elements in the arrays.

Here is a C implementation of this algorithm where the number of elements in the arrays is 80.

```
/* Main Code */
main()
{
  y = DotP(a, x, 80);
}

int DotP(short * restrict m, short * restrict n, int count)
{ int i;
  int sum = 0;

    for (i=0; i < count; i++)
    {
      sum += m[i] * n[i];
    }
return(sum);
}
```

Here is the output of the compiler for the loop kernel for the example above. This is using version 4.0 of the C6000 compiler generating C62x code by default. The compiler options used were:

–k –o3 –mt –mi –mg

```
LOOP:    ; PIPED LOOP KERNEL

   [!A1] ADD   .L2    B7,B5,B5       ; running sum 2
|| [!A1] ADD   .L1    A6,A0,A0       ; running sum 3
||       MPY   .M2X   B8,A4,B7       ; 1 16x16 multiply ; prod 0
||       MPYH  .M1X   B8,A4,A6       ; 1 16x16 multiply ; prod 1
|| [ B0] B     .S1    LOOP           ; branch to loop if loop count >0
||       LDW   .D1T1  *+A5(4),A4     ; load a 32-bit value
||       LDW   .D2T2  *+B4(4),B8     ; load a 32-bit value

   [ A1] SUB   .L1    A1,1,A1        ; running sum count
|| [!A1] ADD   .S2    B7,B6,B6       ; running sum 0
|| [!A1] ADD   .S1    A6,A3,A3       ; running sum 1
||       MPY   .M2X   B8,A4,B7       ; 1 16x16 multiply ; prod 2
||       MPYH  .M1X   B8,A4,A6       ; 1 16x16 multiply ; prod 3
|| [ B0] SUB   .L2    B0,1,B0        ; decrement loop count
||       LDW   .D1T1  *++A5(8),A4    ; load a 32-bit value
||       LDW   .D2T2  *++B4(8),B8    ; load a 32-bit value
```

The compiler created a two-cycle loop. In each loop iteration four 16x16 bit multiplies are taking place in two cycles. The compiler is using four **LDW** instructions to bring in eight 16-bit values per loop iteration. Each of the two multipliers are performing two multiplies per loop iteration and we have four running sums. This two cycle loop takes a total of 15 instructions to implement.

Now let's add the –mv6400 option to the list of compiler options and examine the assembly code generated by the compiler.

```
LOOP:    ; PIPED LOOP KERNEL

   [!A0] ADD   .S1   A17,A16,A16  ; running sum 0
||       MV    .L1   A5,A17    ;copy A5–>A17 source to .M1 in next cycle
|| [ B0] BDEC  .S2   LOOP,B0      ;decrement loop counter & branch if >0
||       DOTP2 .M1   A6,A4,A17 ; 2 16x16 multiplies + add; prod0 & prod1
||       LDNDW .D1T1 *A3++,A5:A4  ; load a 64–bit non–aligned value

   [ A0] SUB   .L1   A0,1,A0      ; running sum counter
|| [!A0] ADD   .S1   A6,A9,A9     ; running sum 1
||       DOTP2 .M1   A7,A17,A6 ; 2 16x16 multiplies + add; prod2 & prod3
||       LDNDW .D1T1 *A8++,A7:A6  ; load a 64–bit non–aligned value
```

The compiler again created a two-cycle loop with four 16x16 bit multiplies taking place per loop iteration. This implementation takes only 9 instructions; this represents a 40% code size reduction as compared to the C62x implementation. Each **DOTP2** instruction returns the dot product between two pairs of signed packed 16-bit values residing in two 32-bit registers. The compiler is using two **LDNDW** instructions to bring in eight 16-bit values per loop iteration. **LDNDW** stands for Load Non-Aligned Double Word. **LDNDW** brings in a 64-bit value that can be aligned at any byte boundary. Note that the memory accesses are not being done in parallel. Non-aligned memory accesses cannot occur in parallel. Because of this the compiler is scheduling the instructions so that only one side of the machine is being utilized.

Can this algorithm be improved? If we tell the compiler that the data is aligned on a double word boundary, then the compiler will be able to perform parallel memory accesses and perhaps improve the partition of the algorithm. We can specify the data alignment with a DATA_ALIGN pragma. **#pragma DATA_ALIGN(a,8)** tells the compiler that the data in ***a*** is aligned on a double word boundary. Let's modify our C code accordingly:

```
main()

{
 #pragma DATA_ALIGN(a,8);
 #pragma DATA_ALIGN(x,8);

  y = dotp(a, x, 80);
 }

int dotp(short * restrict m, short * restrict n, short count)
{ int i;
  int sum=0;

  for (i=0; i < count; i++)
  {
   sum += m[i] * n[i];
   }
   return(sum);
 }
```

Now let's compile the modified source with the same options as before –mv6400 –mi –mt –mg –k –o3 and examine the assembly output of the compiler.

```
LOOP:    ; PIPED LOOP KERNEL

   [ B0] SUB   .L2     B0,1,B0    ; decrement running sum counter
|| [!B0] ADD   .S2     B8,B6,B6   ; running sum 1
|| [!B0] ADD   .L1     A7,A6,A6   ; running sum 0
||       DOTP2 .M2X    B4,A4,B8 ;2 16x16 multiplies + add; prod2 & prod3
||       DOTP2 .M1X    B5,A5,A7 ;2 16x16 multiplies + add; prod0 & prod1
|| [ A0] BDEC  .S1     LOOP,A0    ;decrement loop counter & branch if >0
||       LDDW  .D1T1   *A3++,A5:A4 ; load a 64-bit aligned value
||       LDDW  .D2T2   *B7++,B5:B4 ; load a 64-bit aligned value
```

The compiler has created a single cycle loop with four 16x16 bit multiplies occurring every cycle. As we mentioned before by specifying that the data be aligned on double word boundaries, the compiler can perform parallel memory accesses. The parallel memory accesses allow the compiler to use both .M units. And since each .M unit on the C64x is capable of performing two 16x16 bit multiplies per clock cycle, this is twice the performance possible on the C62x! All that was necessary was to tell the compiler that the data was aligned on double word boundaries and to use the –mv6400 compiler option. No other modifications to the source code were necessary to get 2x the performance.

2.5 Applying Packed Data Processing: Clear Below Threshold Example

Packed data processing is a type of processing where a single instruction applies the same operation to multiple independent pieces of data. For example, we have already seen the **DOTP2** instruction that returns the dot product between two pairs of signed packed 16-bit values residing in two 32-bit registers. Another example is the **ADD2** instruction that performs two independent 16-bit additions between two pairs of 16-bit values. This produces a pair of 16-bit results. In this case, a single instruction, **ADD2**, is operating on multiple sets of data, the two independent pairs of addends.

In the C64x, instructions have been added that that operate directly on packed data (both 8-bit and 16-bit) to streamline data flow and increase instruction set efficiency. An extensive collection of pack and unpack instructions simplifies the manipulation of packed data types. Packed data types can be visualized as 8-bit or 16-bit partitions inside a 32-bit register. These partitions are merely logical partitions. How data in a register is interpreted is determined entirely by the instruction that is using the data. The C64x has a comprehensive collection of 8-bit and 16-bit instruction set extensions. They are included in Table 1.

Table 1. Quad 8-bit and Dual 16-bit Instruction Set Extensions

Operation	Quad 8-bit	Dual 16-bit
Multiply	X	X
Multiply with Saturation		X
Addition/Subtraction	X	X†
Addition with Saturation	X	X
Absolute Value		X
Subtract with Absolute Value	X	
Compare	X	X
Shift		X
Data Pack/Unpack	X	X
Data Pack with Saturation	X	X
Dot product with optional negate	X‡	X
Min/Max/Average	X	X
Bit-expansion (Mask generation)	X	X

† The C62x provides support for 16-bit data with the ADD2/SUB2 instructions. The C64x extends this support to include 8-bit data.
‡ Dot product with negate is not available for 8-bit data

The following code is an example that could benefit by applying packed data processing techniques. The Clear Below Threshold kernel scans an image of 8-bit unsigned pixels, and sets all pixels that are below a certain threshold value to zero.

```
void clear_below_thresh(unsigned char *restrict image, int count,
                        unsigned char threshold)
{int i;
    for (i = 0; i < count; i++)
        {
         if (image[i] <= threshold)
         image[i] = 0;
        }
}
```

Here is the output of the compiler for the loop kernel for the example above. The first time we will compile this for the C62x using compiler options –k –o3 –mt –mi –mg.

```
LOOP:    ; PIPED LOOP KERNEL

   [!A1] STB   .D2T2   B4,*B5    ; store 0 if input < threshold
||       ADD   .L2     1,B5,B5   ; increment output value ptr. by 1 byte
||       CMPGT .L1     A0,A4,A1  ; compare input value to threshold
|| [ A2] B     .S2     LOOP      ; branch to loop if loop count >0
|| [ A2] SUB   .S1     A2,1,A2   ; decrement loop count
||       LDBU  .D1T1   *++A3,A0  ; load 8-bit input value
```

The compiler has created a single cycle loop. One 8-bit value is brought in per loop iteration, one compare is performed and zero is output if the input is less than the threshold.

Now let's add the –mv6400 option to the list of compiler options and examine the assembly code generated by the compiler.

```
LOOP:    ; PIPED LOOP KERNEL

   [!A0] STB   .D2T2   B4,*B5   ; store 0 if input < threshold
||       ADD   .L2     1,B5,B5  ; increment output value ptr. by 1 byte
||       CMPGT .L1     A3,A5,A0 ; compare input value to threshold
|| [ B0] BDEC  .S2     LOOP,B0  ;decrement loop counter & branch if >0
||       LDBU  .D1T1   *++A4,A3 ; load 8-bit input value
```

The output is nearly identical to the C62x assembly code. We again get a single cycle loop with one 8-bit value brought in, one compare performed and a zero output if the input is less than the threshold. We know that the C64x can bring in up to eight 8-bit values per clock cycle per side with an **LDDW** instruction. We also know that the C64x has quad 8-bit compare instructions (in this case we want a **CMPGTU4** instruction) and quad 8-bit bit-expansion instructions (**XPND4**). How can we take advantage of these new instructions from the C language?

CMPGTU4 and **XPND4** are available to the compiler as intrinsics. An intrinsic function is similar to the mathematic functions available in the Run-Time Support Library. An intrinsic allows your C code to directly access the hardware while preserving the C environment. Intrinsic functions have a leading underscore with the function in lower case. For example, the intrinsic for **CMPGTU4** is _cmpgtu4.

We will now apply packed data processing techniques to the Clear Below Threshold kernel. (See the *C6000 Programmer's Guide* Chapter 8 for a full description of packed data processing techniques). The modified code is shown below. The _cmpgtu4() intrinsic compares four 8-bit data values with the threshold value, and the _xpnd4() intrinsic generates a mask for setting pixels to 0 when they are below the threshold value. Note that the new code has the restriction that the input image must be double word aligned, and must contain a multiple of 8 pixels.

```
void clear_below_thresh(unsigned char *restrict image, int count,
                        unsigned char threshold)
  {  int i;
     unsigned temp;
     unsigned t3_t2_t1_t0; /* Threshold (replicated) */
     unsigned p7_p6_p5_p4, p3_p2_p1_p0; /* Pixels */
     unsigned c7_c6_c5_c4, c3_c2_c1_c0; /* Comparison results */
     unsigned x7_x6_x5_x4, x3_x2_x1_x0; /* Expanded masks */

     /* Replicate the threshold value four times in a single word */

     temp = _pack2(threshold, threshold);
     t3_t2_t1_t0 = _packl4(temp, temp);

      for (i = 0; i < count; i += 8)
     {
       /* Load 8 pixels from input image (one double-word). */
          p7_p6_p5_p4 = _hi(*(double*) &image[i]);
          p3_p2_p1_p0 = _lo(*(double*) &image[i]);

       /* Compare each of the pixels to the threshold. */
          c7_c6_c5_c4 = _cmpgtu4(p7_p6_p5_p4, t3_t2_t1_t0);
          c3_c2_c1_c0 = _cmpgtu4(p3_p2_p1_p0, t3_t2_t1_t0);

       /* Expand the comparison results to generate a bitmask. */
          x7_x6_x5_x4 = _xpnd4(c7_c6_c5_c4);
          x3_x2_x1_x0 = _xpnd4(c3_c2_c1_c0);

     /* Apply mask to the pixels. Pixels that were less than or */
     /* equal to the threshold will be forced to 0 because the */
     /* corresponding mask bits will be all 0s. The pixels that */
     /* were greater will not be modified, because their mask */
     /* bits will be all 1s. */
        p7_p6_p5_p4 = p7_p6_p5_p4 & x7_x6_x5_x4;
        p3_p2_p1_p0 = p3_p2_p1_p0 & x3_x2_x1_x0;

     /* Store the thresholded pixels back to the image. */
      *(double*) &image[i] = _itod(p7_p6_p5_p4, p3_p2_p1_p0);
     }
}
```

Now let's compile the modified source with the same options as before –mv6400 –mi –mt –mg –k –o3 and examine the assembly output of the compiler.

```
LOOP:    ; PIPED LOOP KERNEL

         AND     .D1   A9,A5,A4 ;AND upper 4 8-bit input values w/ mask
||       MV      .L1   A4,A9    ;Copy upper 4 8-bit input values to A4
||       MV      .L2   B6,B5    ;Copy lower 4 8-bit input values to B5
||       CMPGTU4 .S1   A4,A8,A5 ;Compare upper 4 input values to thres.
|| [ B0] BDEC    .S2   LOOP,B0  ;Decrement loop counter & branch if >0
||       XPND4   .M2   B5,B7 ;Expand compare upper inputs w/ thres.

         MV      .L1   A4,A5 ;Copy result upper input values AND to A5
||       MV      .S1X  B7,A4 ;Copy result lower input values AND to A4
||       XPND4   .M1   A5,A5 ;Expand compare lower inputs w/ threshold
||       MV      .L2X  A6,B6      ; Copy lower four 8-bit values to B6
||       LDDW    .D1T1 *A3++,A7:A6  ; Load eight 8-bit input values

   [ A0] SUB     .L1   A0,1,A0     ;
|| [!A0] STDW    .D2T1 A5:A4,*B4++ ; Store 8 8-bit values
||       AND     .L2   B5,B7,B7  ; AND lower four 8-bit values w/ mask
||       MV      .S1   A7,A4      ; Copy upper four 8-bit values to A4
||       CMPGTU4 .S2X  B6,A8,B5   ; Compare upper four inputs to thres.
```

The compiler has created a 3-cycle loop where eight 8-bit values are brought in per loop iteration, eight comparisons with the threshold are made every loop iteration and eight outputs are created per loop iteration. The values output are 0 if the input is less than the threshold and the input itself if the input is greater than the threshold. This represents a 2.67x performance increase from our original implementation.

2.6 Larger Register File and Increased Orthogonality

So far we have seen the performance benefits of the wider data paths, increased multiply capability and using the packed data processing features of the C64x architecture. Let's look at one final example that demonstrates the benefits of having a larger register file and increased orthogonality.

Consider a C program with the following structure:

```
for  ( i=0; i<n; i++ ) {
  /* inner loop set up code goes here */
  for ( j=0; j<4; j++ ){
    if(cond1)
      if(cond2)
        if(cond3)
     /* segment0 processing goes here */
        else
     /* segment1 processing goes here */
        else
           if(cond4)
          /* segment2 processing goes here */
           else
          /* segment3 processing goes here */
           else
             if (cond5)
               if (cond6)
             /* segment4 processing goes here */
              else
             /* segment5 processing goes here */
              else
                if(cond7)
             /* segment6 processing goes here */
              else
            /* segment7 processing goes here */
      }
  }
```

Each segment0-7 contains code to do 8 **XOR**s and 24 **ADD**s on 32 different values. Here is an example of a segment showing segment0.

```
   /* segment0 */
 {
   a_0=a_0^a_7; a_1=a_1^a_7; a_2=a_2^a_7; a_3=a_3^a_7;
   a_4=a_4^a_7; a_5=a_5^a_7; a_6=a_6+a_7; a_7=a_7+a_7;
   b_0=b_0+b_7; b_1=b_1+b_7; b_2=b_2+b_7; b_3=b_3+b_7;
   b_4=b_4+b_7; b_5=b_5+b_7; b_6=b_6+b_7; b_7=b_7+b_7;
   c_0=c_0+c_7; c_1=c_1+c_7; c_2=c_2+c_7; c_3=c_3+c_7;
   c_4=c_4+c_7; c_5=c_5+c_7; c_6=c_6+c_7; c_7=c_7+c_7;
   d_0=d_0+d_7; d_1=d_1+d_7; d_2=d_2+d_7; d_3=d_3+d_7;
   d_4=d_4^d_7; d_5=d_5^d_7; d_6=d_6+d_7; d_7=d_7+d_7;
   }
```

Because each segment has so many operations and variables, it is not possible to keep each of these in a register during the loop on a 32 register C62x. Also, it is not profitable for the compiler to execute all these instructions conditionally on an effectively 8-way switch statement. Thus, the compiler creates eight different blocks of code for each of the segments and then branches to the appropriate block based on the triply nested if control flow.

Here is the output of the compiler for the segment for the example above. The first time we will compile this for the C62x using compiler options –k –o3 –mt –mi –mg.

```
segment0:
            LDW     .D2T2   *+SP(72),B2
||          XOR     .L1X    A5,B13,A5

            STW     .D2T1   A5,*+SP(44)
            LDW     .D2T1   *+SP(96),A5
            LDW     .D2T1   *+SP(64),A1
            LDW     .D2T1   *+SP(36),A11
            LDW     .D2T2   *+SP(60),B0
            LDW     .D2T1   *+SP(40),A13

            LDW     .D2T1   *+SP(60),A5
||          ADD     .L1X    A5,B9,A10

            LDW     .D2T1   *+SP(64),A12
||          MV      .L2X    A7,B5

            LDW     .D2T1   *+SP(72),A1
||          ADD     .L1X    A7,B6,A7
||          ADD     .L2X    B2,A1,B2

            STW     .D2T1   A7,*+SP(56)
            LDW     .D2T1   *+SP(40),A7

            LDW     .D2T1   *+SP(40),A5
||          ADD     .L2X    A5,B10,B4
||          ADD     .L1X    A5,B7,A8

            XOR     .L2X    A11,B13,B13
            ADD     .L1X    A1,B12,A1

            STW     .D2T1   A1,*+SP(84)
||          MV      .L1     A11,A1

            ADD     .D1     A1,A13,A12
||          XOR     .L1     A0,A12,A13
||          STW     .D2T2   B4,*+SP(68)
||          ADD     .S2     B0,B3,B4
||          ADD     .L2     B5,B8,B0
||          XOR     .S1     A7,A3,A7

            STW     .D2T2   B0,*+SP(48)
||          XOR     .L1     A5,A6,A5
||          XOR     .S1     A1,A3,A3

            XOR     .L1     A1,A6,A6
||          LDW     .D2T1   *+SP(72),A11
||          SHL     .S1     A1,1,A1
```

```
            STW     .D2T1   A1,*+SP(36)
            LDW     .D2T2   *+SP(60),B0

            B       .S1     L10
||          LDW     .D2T1   *+SP(72),A1

            XOR     .L2X    A0,B12,B12
||          ADD     .S2     B1,B6,B6

            STW     .D2T1   A13,*+SP(64)
||          ADD     .L2     B11,B7,B7
||          ADD     .S2     B1,B9,B9

            STW     .D2T2   B4,*+SP(52)
||          ADD     .L2     B1,B8,B8
||          ADD     .S2     B11,B10,B10

            SHL     .S2     B1,1,B1
||          ADD     .L2     B11,B0,B0
||          ADD     .D2     B1,B5,B5
||          ADD     .L1     A11,A4,A11
            SHL     .S1     A0,1,A0
||          ADD     .L1     A0,A1,A1
||          STW     .D2T2   B0,*+SP(60)
||          SHL     .S2     B11,1,B11
||          ADD     .L2     B11,B3,B3
||          ADD     .D1     A0,A4,A4
```

This segment takes 27 cycles to execute and 58 instructions to implement. As previously mentioned 32 registers are insufficient to handle the number of variables being handled per segment. For this reason the compiler must pop values off the software stack to read in the data (this is shown with the **LDW** *+SP(offset), register). The **XOR** and **ADD** operations are performed and the results are placed back on the stack (shown with the **STW** Register**,+***SP).

Now let's compile this same code and include the –mv6400 option and examine the result.

```
segment0:
            B       .S2      L10
||          MV      .L1X     B6,A3
||          ADD     .L2      B19,B9,B21
||          XOR     .S1      A9,A5,A22
||          XOR     .D1      A16,A5,A21

            MV      .L2X     A18,B17
||          ADD     .S2      B4,B16,B8
||          ADD     .L1      A4,A6,A19
||          ADD     .D2      B7,B9,B5
||          XOR     .S1      A7,A5,A17
||          ADD     .D1X     B1,A6,A23

            XOR     .L2X     A26,B1,B1
||          ADD     .S2      B20,B6,B6
||          ADD     .D2      B20,B19,B19
||          XOR     .L1      A8,A9,A9
||          XOR     .S1      A8,A16,A16
||          ADD     .D1X     A25,B16,A24

            XOR     .L1      A26,A18,A18
||          ADD     .L2X     B17,A6,B18
||          ADD     .S2      B23,B4,B4
||          ADD     .S1      A26,A4,A4
||          ADD     .D2      B20,B7,B7
||          ADD     .D1      A8,A7,A7

            SHL     .S2      B20,1,B20
||          SHL     .S1      A8,1,A8
||          ADD     .L2      B22,B16,B17
||          ADD     .D2      B20,B9,B9
||          XOR     .L1      A8,A5,A5
||          ADD     .D1X     A3,B9,A29

            SHL     .S2      B23,1,B23
||          SHL     .S1      A26,1,A26
||          ADD     .L2      B23,B16,B16
||          ADD     .D2      B23,B22,B22
||          ADD     .L1      A26,A6,A6
||          ADD     .D1X     B23,A25,A25
```

This segment takes 6 cycles to execute and is implemented in 35 instructions! With the larger register file, variables no longer need to be popped and pushed off the software stack. On the C64x, the .D functional units as well as the .S and .L functional units can execute an **XOR.** Also on the C64x, the .D functional unit can now use a data cross path operand as well as the .L, .M and .S units. This allows the compiler to generate a much more favorable schedule due to the larger register file and greater orthogonality of resources. This example shows a 4.5x boost in performance per segment and a reduction in code size of 40%. All that is needed to see this dramatic increase in performance and reduction in code size is to use the –mv6400 compiler option. No modification of the source code is necessary.

3 Conclusion

The C64x is the next generation of the C6000 DSP family and is object code compatible with the C62x DSP. Some key extensions have been made to the C62x VelociTI™ architecture to allow the C64x to complete more work each clock cycle. These extensions include wider data paths, a larger register file, greater orthogonality and new instructions that support packed data processing.

There are various techniques that the C6000 programmer can employ to gain the maximum benefit of these architectural extensions. One technique we applied was providing more information to the C6000 compiler about the alignment of pointer variables as seen in the dot product example. In this case we saw a 2x performance improvement and 46.7% code size reduction as compared to the C62x as we were able to take advantage of the additional multiply capability found on the C64x. We examined another technique that utilized intrinsics to fully leverage the packed data processing capabilities of the C64x platform as seen in the clear below threshold example. This example represents a 2.67x performance increase from our original C62x implementation. One last technique discussed to use –mv6400 compiler option. We saw this in the weighted vector sum example as the C6000 code generation tools were able to schedule around the cross path stall. In the segment code example we achieved a performance gain of 4.5x and a code size reduction of 40% as compared to the C62x by simply using the mv6400 switch.

TI's C6000 Compile Tools were co-developed with the architecture to offer best in class performance. Over the last three years very significant improvements have been made to the compiler by focusing on C6000 specific back end optimizations. Our latest compiler release 4.0 offers 35% better performance and 30% smaller code size than our original compiler. All of these improvements benefit not only the existing C62x and C67x devices but also the C64x. Compiler performance will be further enhanced as the advantages of the C64x VelociTI.2™ extensions are more fully leveraged in subsequent releases.

The C6000 Compile tools and the techniques described in this application brief can be utilized to get the most performance when porting code from the C62x to the C64x DSP platform.

4 References

1. Texas Instruments Inc., *TMS320C6000 CPU and Instructions Set Reference Guide*, Texas Instruments Inc.
2. Texas Instruments Inc., *TMS320C6000 Optimizing C Compiler User's Guide*, Texas Instruments Inc.
3. Texas Instruments, Inc., *TMS320C6000 Programmer's Guide*, Texas Instruments Inc.

VelociTI and VelociTI.2 are trademarks of Texas Instruments.

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 acknowledgment, 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.

Customers are responsible for their applications using TI components.

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.

Copyright © 2000, Texas Instruments Incorporated