TEXAS INSTRUMENTS

Implementation of the Vector Maximum Search Benchmark on the TMS320C8x Parallel Processor

Application Report

1996 Digital Signal Processing Solutions

Texas Instruments

Printed in U.S.A., October 1996

SPRA087

Book Type
Two Lines
Volume #

Book Type
Volume #

Book Type
Two Lines

Book Type

Title
Two Lines
Subtitle
Line Two

year

Title
Two Lines
Subtitle

Title
Two Lines

Title
Subtitle
Line Two

Title
Subtitle

Title

year

Implementation of the Vector Maximum Search Benchmark on the TMS320C8x Parallel Processor

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software 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, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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.

Contents

ABSTRACT

This report uses the *Vector Maximum Search* benchmark to demonstrate the efficient performance of the TMS320C8x parallel processors (PPs). The authors describe a software implementation that uses the PP's advanced assembly language features to implement this benchmark efficiently. Results indicate that the *Vector Maximum Search* benchmark on the TMS320C8x PP exceeds all digital signal processing products currently available.

INTRODUCTION

Texas Instruments (TI™) TMS320C8x devices are a family of multi-processor digital signal processors (DSPs). Current members of the TMS320C8x family are the TMS320C80 (or ’C80) and TMS320C82 (or ’C82) multi-processor DSPs. The ’C80 has the following features:

- Four advanced DSPs called PPs, each capable of performing many operations per instruction
- A RISC processor called the master processor (MP), which has a double-precision floating-point unit (FPU)
- An advanced DMA controller called the transfer controller (TC)
- A video controller
- 50K bytes of on-chip SRAM
- The ’C82 consists of:
  - Two PPs
  - One MP
  - A TC
  - 44K bytes of on-chip SRAM

The TMS320C8x family provides powerful DSP solutions for video conferencing and telecommunication applications. These solutions are evidenced by the TMS320C8x family’s outstanding performance at a wide range of benchmarks.

This application report demonstrates the performance of the TMS320C8x PPs. Two different implementation approaches are discussed and performances are compared. After reading this report, the reader should understand how to write efficient PP assembly language code and have an improved understanding of PP architecture. The authors assume that the reader has some familiarity with the basic concepts of digital signal processing theory and the TMS320C8x architecture.

ALGORITHM

The Vector Maximum Search is a simple algorithm—for a vector of N elements, it returns the value of the largest element (V_{max}) and its location in the vector (Index).

V_{max} = Max {X(0), X(1), X(2), X(3), ... , X(N–2), X(N–1)}

Index = location of V_{max}

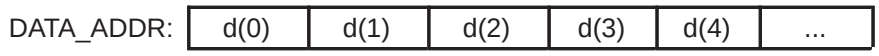
The first vector element is compared to the second vector element. The larger of the two is compared to the third element, the larger of the three is compared to the fourth, and so on. This process continues until all vector elements are compared. For each larger element found, the index is recorded—for example, if the maximum vector element is X(3), V_{max} returns the value of X(3) and its index, 3.

IMPLEMENTATION

Once the Vector Maximum Search algorithm is understood, it can be implemented as a program. The authors' approach in implementing this benchmark was to first implement it using C code. This approach serves two purposes—it helps to outline the program's flow and verifies algorithm functions. After the C code is verified to produce correct results, it is translated to PP assembly language.

The next step in the implementation process is to optimize the PP assembly code. This step demonstrates how two of the PP's advanced features, splittable ALU and multiple flags status, can be used to obtain optimal performance for the benchmark; it is intended to teach the reader some assembly coding tricks that can be applied to future PP application implementations.

Assume that a vector of N signed 16-bit (short) elements is stored in memory at the on-chip address DATA_ADDR.



The following C code implements the *Vector Maximum Search* algorithm where:

- The number of elements in the vector is **N.**
- The maximum vector element is **max.**
- The index of the maximum vector element is **index**.

```
max = DATA_ADDR[0]; /* max and index start with */
index = 0;          /* the first vector element.*/
   for (i = 1; i < N; i ++)
   {
      if (max < DATA_ADDR[i])    /* If the next element */
      {                          /* is larger than max, */
         max = DATA_ADDR[i];     /* copy its value and  */
         index = i;              /* index to max and    */
      }                          /* index, respectively.*/
   }
```

The C code shows that the basic operations of this algorithm are comparisons and conditional moves. Because PP assembly language supports conditional operations, this C code can be easily translated to PP assembly language. The following shows this code translated to PP assembly language:

```
La_D_ptr  .reg   la
Ga_count  .reg   ga
Gx_one    .reg   gx
index     .reg   d
data      .reg   d
d_count   .reg   d
max       .reg   d
test      .reg   d
    .global DATA_ADDR
    data =h *(La_D_ptr = xba + DATA_ADDR)
    index = 0
      || max =h *La_D_ptr
    Gx_one = 1
    lrs2 = N – 2
    le2 = LOOP_MAX_END
    Ga_count = 0
LOOP_MAX:
    test = max – data
      || d_count =&*(Ga_count += Gx_one)
      || data =h *++La_D_ptr
LOOP_MAX_END:
    max =[lt] data
      || index =[lt] d_count
    .cjump LOOP_MAX


.
.
.
```

The preceding PP assembly code requires a two-cycle (two instructions) tight loop to test if the current element is larger than the previous element. In the first instruction, two vector elements are compared (by calculating the difference) in the PP's ALU, the iteration counter is incremented in the global address using address unit arithmetic, and the next signed 16-bit vector element to be checked is loaded in the local address unit. The difference calculation in this instruction sets status bits in the PP's status register (SR). Based on the status of the first instruction, the second instruction conditionally selects the larger vector element and its corresponding location index using the **less than** [lt] condition code. This tight loop is iterated N times. After the Nth iteration, **max** has the value of the largest vector element and **index** has its corresponding location. Refer to the *TMS320C80 Parallel Processor User's Guide*, (literature number SPRU110A) section 8.9, for more information on conditional operations.

In the preceding code, the PP performs a subtraction, an increment of a counter, two conditional register-to-register moves, and a register load from memory in two instruction cycles. Most DSPs would require five or more instruction cycles to perform these operations. At first glance, this code looks quite efficient; however, the PP has advanced features that allow further optimization of this code.

The PP has a 32-bit ALU that is *splittable*, meaning it can perform multiple arithmetic operations—two 16-bit adds/subtracts or four 8-bit adds/subtracts. The PP also has the ability to save the status of the split ALU results in a special register called the multiple flags (mf) register.

The mf expander, which is a special input to the ALU, takes the mf register and creates a mask based on the mf bits that were set by the split ALU operation. This mask can be used to control multiple byte or halfword operations. To save on loads, the PP can load two 16-bit values in a single 32-bit load. Using these features, it will be shown how two vectors can be processed in a three-instruction-cycle loop, thereby reducing the processing from 2 cycles per vector to 1.5 cycles per vector—a 33% reduction in cycles.

The following *tight loop* code processes two 16-bit vectors per iteration:

```
.
.
.
LOOP_MAX:
    test =me max – data
    max = (max & ~@mf) | (data & @mf)
      || d_count = &*(La_count += Lx_two_two)
LOOP_MAX_END:
    index = (index & ~@mf) | (d_count & @mf)
      || data = *++La_D_addr
.
.
.
```

The following example illustrates how the preceding code uses the expanded mf register to control the selection of larger numbers from A, B, C, D and to record the appropriate indexes. Assume A is in the upper half and B in the lower half of register **max**, and C is in the upper half and D is in the lower half of register **data.** The PP's status register has been set to enable the PP to perform halfword, multiple-arithmetic operations and to set mf based on the sign of the split ALU result.

A = 0x1, B = 0x7, C = 0x3, D = 0x4;

First Instruction

	A	B
max	0000 0000 0000 0001	0000 0000 0000 0111
	C	D
data	0000 0000 0000 0011	0000 0000 0000 0100
test = me max – data	1111 1111 1111 1110	0000 0000 0000 0011
Sign out	1	0
mf register	xxxx xxxx xxxx xxxx	xxxx xxxx xxxx xx10

Second Instruction

Expanded mf (@mf)	1111 1111 1111 1111	0000 0000 0000 0000
	C	B
max = (max & ~@mf) | (data & @mf)	0000 0000 0000 0011	0000 0000 0000 0111

Third Instruction

Index	UpperIndex	LowerIndex
Count	UpperCount	LowerCount
Expanded mf (@mf)	1111 1111 1111 1111	0000 0000 0000 0000
index = (index & ~@mf) | (count & @mf)	UpperCount	LowerIndex

After execution of instructions one, two, and three, the larger number from the lower half of **max** (B) and the larger number from the upper half of **data** (C) are the new max values. The upper half of **index** is updated with the new count because a new max value for the upper half has been found; the lower half of **index** remains unchanged as a new max for the lower half was not found. Note that La_count and Lx_two_two were initialized to the appropriate values such that the upper half of **count** indexes the number of even vectors, and the lower half indexes the number of odd vectors. Also, two 16-bit vector elements are loaded in a single 32-bit load. This three-instruction loop is iterated N/2 times for an N-element vector. After N/2 iterations, a comparison is performed to find the maximum vector element in **max**—that is, to

determine if the upper 16 bits or lower 16 bits are larger. This comparison can be done using the following instructions:

```
max = mc max – max\\16
  || max = max
  || d0 = &*(16)
max = (max & ~@mf) | (max\\d0 & @mf)
  ; The max vector element is in both halves of max.
index = (index & ~@mf) | (index\\d0 & @mf)
  ; The max index is in both halves of index
  || max =h1 max
  ; max now is only in the lower half.
index = index >> 16
  ; index now is only in the lower half.
```

After these instructions are executed, **max** contains the value of the largest vector element, and **index** contains its corresponding position. The setup code for the loop and instructions is not shown; however, Appendix A contains a complete source code listing of the *Vector Maximum Search* benchmark.

PERFORMANCE AND MEMORY REQUIREMENTS

This section outlines the performance, register usage, memory requirements, and restrictions of the *Vector Maximum Search* benchmark shown in Appendix A.

Table 1 shows performance data.

Table 1. Benchmark Performance Data

	PP INSTRUCTION CYCLES	MEMORY (BYTES)
Benchmark pre-amble code	8	64
Benchmark kernel code	3N/2	24
Benchmark post-amble code	7	56
Total	15 + 3N/2	144

Note: N is the number of elements in a vector.

Register Usage

- 5 of 8 data registers
- 1 of 8 global address unit registers
- 3 of 8 local address unit registers
- 4 of 13 loop control registers

Data Memory Requirements (in bytes)

- PP data RAM 2*N elements

Restrictions

- DATA_ADDR must be 16-bit signed data.
- DATA_ADDR must be 32-bit aligned.

SOFTWARE USAGE

The *Vector Maximum Search* software in Appendix A is C-callable PP assembly code that executes on a TMS320C8x PP. A main server program written in C can be used to call the V_{max} PP code. This server program passes three arguments to the PP, and the PP returns two values: the maximum vector and its location. Table 2 shows the prototype of this V_{max} function.

Table 2. Prototype of V_{max} Function

SYNTAX	short vector_maximum (short *arg1, short arg2, vmax *arg3);
PARAMETERS	arg1 /* Pointer vector address */ arg2 /* Number of vector elements */ arg3 /* Pointer to structure of return values where: typedef struct { short max; short index; } vmax; */
RETURN VALUE	max /* Value of maximum vector element */ index /* Location of maximum vector element */
DESCRIPTION	This function is called to do the *Vector Maximum Search.*

CONCLUSION

The *Vector Maximum Search* benchmark of the TMS320C8x PP exceeds all DSP products currently available. This report emphasizes the use of multiple arithmetic and the *multiple flags* register for efficient TMS320C8x programming. Additionally, the advanced TMS320C8x architecture has many other features that can be used to write highly efficient assembly code.

APPENDIX A

```
*---------------------------------------------------------------
* Copyright (C) 1996 Texas Instruments Inc.  All Rights Reserved.
*---------------------------------------------------------------
* vmaxs.p —— source code for a C–callable, signed vector
* maximum search routine designed to run on one of the C8x's
* parallel processors (PPs).
*
* Environment:
*    —— This code executes on a PP (TMS320C8x device).
*****************************************************************
* C PROTOTYPE
*
* void vector_maximum(short *vector_addr, short n, VMAX *vmaxs)
*
*  typedef struct {
*   short max;   /* Value of largest vector element */
*   short index; /* Location (index) of largest vector element */
*  } VMAX;
*
* vector_maximum returns the value and index of the largest
* element of an n length, signed, 16–bit vector pointed to
* by vector_addr.
*****************************************************************
* DESCRIPTION
*
*  Arguments description:
*
*   vector_addr  Address of vector
*   n            Number of elements in the vector
*   vmaxs        Pointer to a structure of type VMAX
*
*  Memory map
*
*   vector_addr | d(0) | d(1) | d(2) | d(3) | d(4) | d(5) |
```

```
*****************************************************************
* IMPLEMENTATION
*
* The following is the equivalent C code that has been
* implemented in PP assembly:
*
*   void vector_maximum(short *vector_addr, short n, VMAX *vmaxs)
*   {
*       max->vmaxs = vector_addr[0];
*       index->vmaxs = 0;
*       for (i = 1; i < n; i++)
*       {
*            if (max->vmaxs < vector_addr[i])
*            {
*                 max->vmaxs = vector_addr[i];
*                 index->vmaxs = i;
*            }
*       }
*   }
*****************************************************************
* ASSUMPTIONS
*
*       (1) Vector elements are 16-bit signed values
*       (2) Overflow is not checked
*       (3) Vector starting address must be 32-bit aligned
*       (4) n <= 1024
*****************************************************************
* PERFORMANCE
*
*      Number of cycles = 8 + (3*n)/2
*            where:  n is the vector length
*           (excluding ICACHE misses)
*****************************************************************
* MEMORY
*
*  Program memory:      144 bytes (18 PP instruction words)
*  On-chip data RAM:    n*2 bytes (Input vector)
*  Parameter RAM:       0 bytes
*****************************************************************
```

```
arg1      .set   d1
arg2      .set   d2
arg3      .set   d3

; Define data register names
d_index   .reg   d
index     .reg   d
d         .reg   d
max       .reg   d
d_count   .reg   d
test      .reg   d
d_two     .reg   d
loop_cnt  .reg   d

; Define local address unit register names
La_D_addr .reg      la
La_count  .reg      la
Lx_two_two.reg      lx

; Define global address unit register names
Ga_struct .reg   ga

; Define constants
MAX       .set   0
INDEX     .set   1
   .global $vector_maximum
   .entry arg1, arg2, arg3
   .lock d6, d7, d0, a4, a12
$vector_maximum:
     loop_cnt = arg2>>1
       || La_D_addr = arg1
     *--sp = sr  ; Push contents of sr onto stack
     sr = 0x12d  ; Set up sr for halfword arithmetic
     max = *La_D_addr
         ; Initial maxs are the first vector element

         ; max = | max1 | max2 |

       || Ga_struct = arg3
     le2 = LOOP_MAX_END    ; Load the loop end register
     lrs2 = loop_cnt – 2   ; Initialize the loop count register
```

```
      d_two = 2             ; Load a temporary d reg. with 2
        || La_count = &*(1)
           ; Register used to keep track of the current vector count;
           ; the upper half keeps track of the d1 count, and the lower half
           ; keeps track of the d2 count

           ; La_count =  | 0x0 | 0x1 |

        || d = *++La_D_addr
           ; Read the next two vectors

           ; d = | d1 | d2 |

       Lx_two_two =rh d_two
           ; Register used to increment the vector count

           ; La_two_two = | 0x2 | 0x2 |

        || index = &*(1)
           ; Initial index

           ; index = | 0x0 | 0x1 |

LOOP_MAX:
           ;************************************
           ; Process two 16 vectors in this loop
           ;************************************
       test = me max – d
           ; Test to see if the d elements > max elements
           ; Set mf on carries: if max1 > d1 set mf bit 1, if
           ; max2 > d2 set mf bit 0

           ; test = | d1 – max1 | d2 – max2 |

        max = (max & ~@mf) | (d & @mf)
           ; Uses the mf expand operator to merge d1 and/or
           ; d2 if they are larger than their corresponding
           ; maxs

           ; max = | max1 | max2 |

          || d_count = &*(La_count += Lx_two_two)
           ; Update vector count
           ; Increment in twos because are processing
```

```
          ; Two vectors in each loop
          ; d_count = 0:1, 2:3, 4:5, .... , N–2:N–1
LOOP_MAX_END:
     index = (index & ~@mf) | (d_count & @mf)
          ; If one of the d's, or both, is a new max,
          ; merge the corresponding d_count into
          ; the corresponding index
       || d = *++La_D_addr
```

; d = 


```
          ; Determine if max1 or max2 is larger
     max = me max – max\\16
       || max = max
       || d0 = &*(16)
     max = (max & ~@mf) | (max\\d0 & @mf)
          ; The max vector element is in both halves of max
     index = (index & ~@mf) | (index\\d0 & @mf)
          ; The max index is in both halves of index
       || max = h1 max
          ; max now is only in the lower half
     index = index >> 16
          ; index now is only in the lower half
       || sr = *sp++    ; Restore sr and sp
     br = iprs
     *(Ga_struct + [MAX]) = h max
     *(Ga_struct + [INDEX]) = h index
```