IMPLEMENTATION OF A DIGITAL COPIER USING TMS320C6414 VLIW DSP PROCESSOR

Taeksang Hwang

Sindoricoh Co., Ltd.
277-22, 2Ka, Sungsu-dong, Sungdong-gu,
Seoul, KOREA 133-705
E-mail: taeksanghwang@sr.sindo.com

Wonyong Sung

School of Electrical Engineering
Seoul National University
San 56-1 Shilim-dong, Gwanak-gu,
Seoul, KOREA 151-742
E-mail: wysung@dsp.snu.ac.kr

ABSTRACT

In this paper, we developed real-time image processing programs for a digital copier using TMS320C6414 CPU. The CPU is good for real-time image processing because of multiple and packed-data processing functional units. However, it needs careful programming to exploit deep pipelining, multiple functional units and packed-data instructions. All the critical functions for the implementation of a digital copier, which include shading correction, X-zoom, 2D filtering, and halftoning, are implemented through assembly programming. Programs using linear assembly programming followed by the assembly optimization in software are compared with the manual assembly coded versions. The results show that explicit disambiguation of memory dependency is most critical for the assembly optimization. The cache miss effects are also evaluated.

1. INTRODUCTION

Digital copiers are now becoming popular due to its versatile ability to edit, enhance, store and transmit scanned images. Currently, digital copiers are mostly implemented using hardwired image processing circuits because of the demand for high throughput. However, the hardware based circuits are disadvantageous for implementing complex functions, such as editing and compression. The Texas Instruments' digital signal processor TMS320C6414 can achieve a very high processing rate due to its VLIW architecture and packed data processing support [1]. The CPU can process up to 8 instructions at each clock because of its pipelined and VLIW characteristics. It can also process up to four pixels of 8-bit data at each instruction because of the packed data processing capability. The prototype digital copier developed performs the basic image processing steps depicted in Fig. 1 [2].

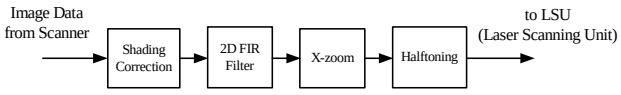

Fig. 1. Image processing flow for a digital copier

2. TMS320C6414 ARCHITECTURE AND PROGRAM DEVELOPMENT ENVIRONMENTS

2.1. TMS320C6414 Architecture

The C6000 family employs a very long word instruction (VLIW) set architecture to increase the performance while providing the flexibility in programming. We can also expect fairly good high-level language oriented development environments since the advance of VLIW compiler is quite dramatic in recent years [3]. The C64x architecture supports packed data processing (sub-word parallelism) so that four 8-bit operations can be conducted using one 32-bit ALU. It also supports double word (64 bit) load and store from non-aligned memory, contains more number of registers, and allows the access of any registers at the other data-path as operands. For example, the **dotpsu4** instruction can perform the dot-product of four 8-bit data using M1 or M2 functional unit. Besides this instruction, the C64x architecture supports ADD, SUB, MPY, PACK, and UNPK of four 8-bit data using a functional unit in the data-paths.

2.2. Program Development Environments

There are basically three different application development methods using C6x architecture. A direct way is to conduct manual assembly programming. Obviously, this is a very difficult method to apply for the C64x architecture because of the deep pipeline, multiple functional units and the need of packed data processing. The second method is using the VLIW compiler. This is the most convenient method to a programmer. However, the performance for time-critical operations is yet to be satisfactory in many cases. Especially, the compiler can hardly choose instructions using packed data processing. The third approach is using the "assembly optimizer." A programmer develops an application code using C6x instructions in a serial manner. The, so called, linear assembly programming is much easier since a programmer does not have to consider the pipelining fill or utilization of multiple functional units. The assembly optimizer produces an optimized, in other words parallelized and pipelined, version from this linear assembly code.

3. IMPLEMENTATION OF IMAGE PROCESSING ALGORITHMS

3.1. Shading Correction

The shading correction process compensates for the non-constant intensity of the light source of the scanning device along a line. This is rather a simple step and just needs to multiply the correction coefficients to the scanned line data. Note that the coefficients and the line data are all represented in 8-bit. In the non packed-data linear assembly coding for the shading correction, one pixel data represented in 8-bit is loaded by the **ldbu** (load byte unsigned) instruction and only one pixel is executed for each loop. However, in the packed-data linear assembly version, four pixels are loaded and executed at each loop. The results are saved to a pair of registers in 16-bit format. Note that **mpyu4** (multiply unsigned by unsigned packed 8-bit) instruction is used for the multiplication with the coefficients. However, there are some other extra works, such as **shru2** for the shift of packed data, **cmpgt2** for clamping and **packl4** for packing to four 8-bit data. Figure 2-(a) shows the initially optimized linear assembly code with packed-data for shading correction. This code requires 17 cycles for each loop, which translates 4.25 cycles for each pixel.

```
loop:   ; PIPED LOOP KERNEL

           LDW     .D2T2   *B5++,B7        ; ^ |23|
||         LDW     .D1T1   *A3++,A4        ; ^ |24|

           NOP             4
           MPYU4   .M1X    B7,A4,A5:A4     ; ^ |25|
           NOP             3
           SHRU2   .S1     A4,0x7,A4       ; ^ |29|

           SHRU2   .S2X    A5,0x7,B7       ; ^ |28|
||         CMPGT2  .S1     A4,A8,A9        ; ^ |33|

           MV      .D1     A4,A5           ; |31|
|| [ A1]  BDEC    .S1     loop,A1         ;
||         AND     .L1     0x1,A9,A0       ; ^ |36|
||         CMPGT2  .S2     B7,B6,B8        ; ^ |32|

           AND     .D2     0x2,B8,B1       ; |35|
||         MV      .S2     B7,B8           ; |30|
||         AND     .D1     0x2,A9,A0       ; |37|
||         AND     .L2     0x1,B8,B0       ; ^ |34|
|| [ A0]  OR      .S1     A4,A7,A5        ; ^ |40|

   [ B0]  OR      .D2X    B7,A7,B8        ; ^ |38|
|| [ A0]  OR      .D1     A4,A6,A5        ; ^ |41|

   [ B1]  OR      .D2X    B7,A6,B8        ; ^ |39|
           PACKL4  .L2X    B8,A5,B7        ; ^ |42|
           STW     .D2T2   B7,*B4++        ; ^ |60|
```

Fig. 2-(a). Optimized linear assembly code with packed-data for shading correction without a dependency directive.

Here we can find that there are 7 **nop** cycles inside of the loop, which is due to the dependency problem incurred by the load and store instructions. However, there is no dependency among the memory accesses in this algorithm. So, we add an optimizer directive of **no_mdep** to indicate that there is no memory dependency in this code, which results in the program shown in Fig. 2-(b). This code requires 5 cycles for each loop, which translates 1.25 cycles for each pixel. We also tried manual assembly programming, which results in 4 cycles for each loop. We also developed a program that does not utilize packed-data instructions. The results are summarized in Table 1.

```
loop:   ; PIPED LOOP KERNEL

           AND     .D2     0x2,B9,B0       ; |35|
|| [ A0]  OR      .D1     A4,A6,A3        ; ^ |41|
|| [ B0]  OR      .S2X    B7,A7,B8        ; ^ |38|

   [ B0]  OR      .D2X    B7,A6,B8        ; ^ |39|
||         SHRU2   .S1     A4,0x7,A4       ; @^ |29|

           CMPGT2  .S1     A4,A8,A5        ; @^ |33|
||         SHRU2   .S2X    A5,0x7,B7       ; @^ |28|
||         MPYU4   .M1X    B7,A16,A5:A4    ; @@|25|
||         LDW     .D1T1   *A9++,A16       ; @@@|24|
||         LDW     .D2T2   *B4++,B7        ; @@@|23|

   [ A2]  MPYSU   .M1     2,A2,A2         ;
||         PACKL4  .L2X    B8,A3,B8        ; |42|
||         MV      .D1     A4,A3           ; @|31|
||         AND     .S1     0x1,A5,A0       ; @^ |36|
||         CMPGT2  .S2     B7,B6,B9        ; @^ |32|

   [!A2]  STW     .D2T2   B8,*B5++        ; |60|
|| [ A1]  BDEC    .S1     loop,A1         ; @
||         MV      .S2     B7,B8           ; @|30|
||         AND     .D1     0x2,A5,A0       ; @|37|
||         AND     .L2     0x1,B9,B0       ; @^ |34|
|| [ A0]  OR      .L1     A4,A7,A3        ; @^ |40|
```

Fig. 2-(b). Optimized linear assembly code with packed-data for shading correction with a dependency directive.

Table 1. The number of operations for shading correction

	Linear assembly without packed-data		Linear assembly with packed-data		Manually optimized assembly
	without opt.	with opt.	without opt.	with opt.	-
Cycles /pixel	21	4.25	7.75	1.25	1

3.2. 2D FIR Filter

The 2-dimensional filter has a size of 7*5, and is used for image enhancement and noise suppression. This is the most computation intensive step in the real-time implementation of a digital copier. The input data are dot-producted with the signed 8-bit filter coefficients and the result is shifted for scaling, and clamped to 8-bit unsigned value.

The non packed-data version of code can utilize the symmetry of the filter coefficients for reducing the number of multiplications. In the packed-data version of the code, the non-aligned double word load instruction, **ldndw**, is used for loading 8 pixels of data, and **dotpsu4** instruction is used intensively, which conducts multiply and add operations for 4 taps.

The experimental results show that the assembly-optimized non packed-data linear assembly code for 2D filter requires about 22 cycles, and the assembly-optimized packed-data linear assembly version, whose code is shown in Fig. 3, needs 6 cycles. Note that the manually optimized version also requires 6 cycles.

```
loop:    ; PIPED LOOP KERNEL

     [ B0]  BDEC     .S2     loop,B0              ;
||           ADD      .S1X    B16,A8,A5            ; |70|
||           DOTPSU4  .M2X    B8,A6,B17            ; @@|41|
||           DOTPSU4  .M1     A26,A5,A28           ; @@|46|
||           LDNDW    .D1T1   *A16,A7:A6           ; @@@ ^ |34|

             SHR      .S1     A5,0x3,A5            ; |73|  divide 8
||           ADD      .D2X    B16,A28,B5           ; @|63|
||           ADD      .L1X    B5,A7,A30            ; @|64|
||           DOTPSU4  .M1     A25,A8,A27           ; @@|54|
||           LDNDW    .D1T1   *+A16[A20],A5:A4     ; @@@ ^ |35|

             CMPLT    .L1     A5,A21,A0            ; |75|
||           ADD      .S1     A27,A29,A29          ; @|66|
||           ADD      .D2     B5,B4,B18            ; @|65|
||           DOTPSU4  .M1     A17,A7,A7            ; @@|51|
||           DOTPSU4  .M2X    B6,A6,B5             ; @@|50|
||           LDNDW    .D1T1   *+A16[A3],A9:A8      ; @@@ ^ |37|

             CMPGT    .L1     A5,A23,A0            ; |76|
|| [ A0]    MV       .S1     A21,A5               ; |80|
||           ADD      .D2     B16,B17,B16          ; @|68|
||           DOTPSU4  .M1     A26,A9,A29           ; @@|55|
||           DOTPSU4  .M2X    B9,A4,B16            ; @@|45|
||           LDNDW    .D1T1   *+A16[A24],A7:A6     ; @@@ ^ |36|

     [ A1]   MPYSU    .M1     2,A1,A1              ;
|| [ A0]    MV       .S1     A23,A5               ; |81|
||           ADD      .L1X    A30,B18,A4           ; @|67|
||           DOTPSU4  .M2     B8,B4,B16            ; @@|58|
||           LDNDW    .D1T2   *+A16[A18],B5:B4     ; @@@ ^ |38|

    [!A1]   STB      .D1T1   A5,*A22++            ; |85|
||           ADD      .S1     A29,A4,A8            ; @|69|
||           DOTPSU4  .M2     B7,B5,B17            ; @@|59|
||           ADD      .D2X    B17,A8,B4            ; @@|62|
||           ADD      .L1     0x1,A16,A16          ; @@@ ^ |78|
||           DOTPSU4  .M1     A19,A7,A8            ; @@@|42|
```

Fig. 3. Optimized packed-data linear assembly code for 2D FIR filter.

3.3. X-zoom

A digital copier independently zooms original image in X-Y directions. The zooming ratio of 25% to 400% in 1% step is usually needed. The X-zoom is the scaling of the original image along a scanned line and is performed by digital processing, while the Y-zoom is conducted by changing the scanning speed. We employed the interpolation based method for X-zoom, and an 8-tap FIR filter is employed. Since a wide range of zooming ratio is needed, it is necessary to find out the value of the hypothetical pixel point. Note that this location is pre-computed according to the zooming ratio to reduce the overhead of real-time processing [2]. The packed-data version code utilized **ldndw** instruction and **dotpsu4** instruction intensively. The results are summarized in Table 2.

3.4. Halftoning

The halftoning is a very important step in a digital copier because most printing units only support bi-level or a few levels for each pixel while the scanned image is represented in 8-bit data. We use the error diffusion method with the 8*3 kernel shown in Fig 4. Although the original Floyd and Steinberg kernel employs a very small order of the error diffusion filter, a higher order filter is employed mainly because of the increase in the resolution of image (600 dpi) [4]. When the filter order is lower than the order of parallelism supported by the architecture, some multiple output parallel computation methods can yield better results [5]. However, in this study, we just try to compute the filter kernel in parallel because the order of the filter is quite large. Note that it is very complex to compute multiple pixels at a time since the quantization error of the current pixel is used as the input of the next pixel in the error diffusion method.
The packed-data version code utilized **ldndw** and **dotpsu4** instructions repeatedly. The error values of the previous pixels are stored in the registers, thus they need not be loaded inside of the loop. The assembly-optimized packed-data linear assembly version code is shown in Fig. 5. The non-packed-data optimized linear assembly code for halftoning requires about 15 cycles, and the optimized packed-data linear assembly code needs about 7 cycles, and the manually optimized version needs 7 cycles.

e00	e01	e02	e03	e04	e05	e06	e07
e10	e11	e12	e13	e14	e15	e16	e17
e20	e21	e22	*				

Fig. 4. Error diffusion filter kernel.

4. IMPLEMENTATION RESULTS AND THE EFFECTS OF CACHE MISSES

Table 2 summarizes the implementation results of the functional steps shown in Fig. 1. As shown in this table, the quality, in terms of the number of cycles, of the optimized linear assembly version is almost similar to that of the manually optimized assembly codes. Note that the version with the packed-data instructions is about 100% to 267 % more efficient than the non packed-data version.
We also need to consider the cache miss effects. TMS320C6414 contains two separate L1 caches of 16 KB, one for program and the other for data. There is also 1 MB of L2 cache, which is configured as a RAM block. Since the size of codes shown above are all very small, there is no need to worry about the program cache misses. This is a very typical data intensive application. Since the 2D filtering and error diffusion halftoning require multiple input lines, a small block of input data is stored in the L1 cache memory, instead of holding a line data. The cache evaluation results are shown in Table 3. Note that when the input data is stored as an array of 32*256, where 256 is the number of lines, the cache misses are substantial. However, for the other cases, the miss effects are not noticeable due to the large size of L1 cache and the block processing of image data.

```
loop:   ; PIPED LOOP KERNEL

            ADD         .S2X        B17,A16,B4          ; |59|
||          DOTPSU4     .M2         B4,B22,B5           ; @|48|
||          DOTPSU4     .M1         A4,A7,A9            ; @|52|
||          LDNDW       .D2T1       *+B9[B20],A5:A4     ; @@|43|

   [ A0]    BDEC        .S1         loop,A0             ;
||          ADD         .S2         B17,B4,B5           ; ^ |60|
||          DOTPSU4     .M2         B5,B8,B4            ; @|49|
||          LDNW        .D2T1       *+B9[B16],A4        ; @@|44|

            SHR         .S2         B5,0x3,B5           ; ^ |61|
||          LDNDW       .D2T2       *B9,B5:B4           ; @@|39|

            MV          .D2         B23,B24         ;
||          ADD         .S2         B4,B5,B17       ; ^ |63|
||          ADD         .D1         A4,A5,A16       ; @|57|

            ADD         .D1         1,A8,A8         ; |65|
||          CMPGT       .L2         B17,B21,B0      ; ^ |65|
||          MV          .S2         B9,B23          ;
||          LDBU        .D2T2       *B18++,B4       ; @|45|

    [ B0]   STB         .D1T1       A18,*-A8(1)     ; |66|
||  [ B0]   SUB         .S2         B17,B7,B17      ; ^ |67|
||          ADD         .L2         B5,B4,B4        ; @|56|
||          ADD         .D2         0x1,B9,B9       ; @@|72|
||          DOTPSU4     .M1         A4,A6,A4        ; @@|50|

    [!B0]   STB         .D1T1       A17,*-A8(1)     ; |68|
||          MV          .S2         B24,B6          ;
||          STB         .D2T2       B17,*+B6[B19]   ; |70|
||          ADD         .L2X        A9,B4,B17       ; @|58|
||          MPY         .M2         0x2,B17,B17     ; @^ |53|
||          DOTPSU4     .M1         A5,A3,A5        ; @@|51|
```

Fig. 5. Optimized packed-data linear assembly code for halftoning.

Table 2. Implementation results (cycles/pixel).

	Optimized linear assembly without packed-data	Optimized linear assembly with packed-data	Manually optimized assembly	Min. cycles
Shading Correction	4.25	1.25	1	0.375
7*5 2D Filter	22	6	6	6
X-zoom	10	5	5	4
Halftoning	15	7	7	5

5. CONCLUDING REMARKS

The linear assembly programming followed by the assembly optimization in software results in a quite good quality of code, which is comparable to that of the manually optimized version, when the dependencies among memory accesses are properly suggested in terms of the directives. The packed-data processing instructions also bring about 100% to 267% of speedup. The developed code requires about 19 cycles for each pixel, which translates that a 600MHz C6414 CPU can perform all the real-time processing needed for a 30 ppm, 600 dpi, A4 size copier. The programmable CPU based architecture not only can support real-time image processing but is much better for implementing complex off-line functions, such as image compression. Thus, the DSP-based hardware and programs seem quite attractive for the implementation of next generation multi-function digital copiers.

Table 3. Data access statistics (stall/read hit/read miss).

	128*64	64*128	32*256
Shading Correction	115/ 4722/20	103/ 5863/18	103/ 6643/18
7*5 2D Filter	263/ 79578/45	290/ 82388/50	985/ 87904/166
X-zoom	241/ 41379/35	199/ 42857/29	937/ 45695/135
Halftoning	889/ 55026/150	848/ 57082/141	1628/ 60919/272

6. ACKNOWLEDGMENTS

This study was supported by the Brain Korea 21 Project (0019-19990027) and the National Research Laboratory program (2000-X-7155) supported by the Ministry of Science and Technology in KOREA.

7. REFERENCES

[1] *TMS320C6414 Data Sheet*, Aug. 2002, Texas Instruments.

[2] J. W. Ahn and W. Sung, " Pentium-MMX based implementation of a digital copier," in *Proc. 1998 IEEE Workshop on Signal Processing Systems(SiPS98)*, Oct. 1998, pp. 142-151.

[3] *TMS320C6000 Optimizing C Compiler User's Guide*, Literature number SPRU187I, Apr. 2001, Texas Instruments.

[4] R. W. Floyd and L. Steinberg, " An adaptive algorithm for spatial grayscale," in *Proceedings for SID*, vol. 17, no. 2, 1976, pp. 75-77.

[5] Jae-Woo Ahn and Wonyong Sung, " Multimedia processor-based implementation of an error-diffusion halftoning algorithm exploiting subword parallelism," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 16, no. 2, pp. 129-138, Feb. 2001.

[6] *TMS320C6000 Programmer's Guide*, Literature number SPRU198F, Feb. 2001, Texas Instruments.