TMS320C55x Hardware Extensions for Image/Video Applications Programmer's Reference

Preliminary

SPRU098
February 2002

Preface

Read This First

About This Manual

Welcome to the TMS320C55x Hardware Extensions for Image/Video Applications Programmer's Guide. The hardware extensions on the C5510 and C5509 DSPs strike the perfect balance of fixed-function performance with programmable flexibility, while achieving low-power consumption, and cost that traditionally has been difficult to find in the video-processor market.

How to Use This Manual

The information in this document describes the TMS320C55x hardware extension in several different ways.

- ❑ Chapter 1 provides an overview of the hardware extensions.
- ❑ Chapter 2 discusses the DCT/IDCT hardware extension.
- ❑ Chapter 3 discusses the motion estimation hardware extension.
- ❑ Chapter 4 discusses the pixel interpolation hardware extension.
- ❑ Appendix A provides the source code for the hardware extension macros in alphabetic order.

Notational Conventions

This document uses the following conventions:

- ❑ Program listings, program examples, and interactive displays are shown in a `special typeface`.
- ❑ The TMS320C55x is also referred to in this reference guide as the C55x.

Related Documentation From Texas Instruments

The following books describe the TMS320C55x devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number. Many of these documents can be found on the Internet at http://www.ti.com.

TMS320C55x Image/Video Processing Library Programmer's Reference (literature number SPRU037) describes a collection of high-level optimized image/video processing functions. These functions include many C-callable, assembly-optimized, general-purpose image/video processing routines.

TMS320C55x Technical Overview (SPRU393). This overview is an introduction to the TMS320C55x™ digital signal processor (DSP). The TMS320C55x is the latest generation of fixed-point DSPs in the TMS320C5000™ DSP platform. Like the previous generations, this processor is optimized for high performance and low-power operation. This book describes the CPU architecture, low-power enhancements, and embedded emulation features of the TMS320C55x.

TMS320C55x DSP CPU Reference Guide (literature number SPRU371) describes the architecture, registers, and operation of the CPU for the TMS320C55x™ digital signal processors (DSPs).

TMS320C55x DSP Algebraic Instruction Set Reference Guide (literature number SPRU375) describes the TMS320C55x™ DSP algebraic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the mnemonic instruction set.

TMS320C55x DSP Mnemonic Instruction Set Reference Guide (literature number SPRU374) describes the TMS320C55x™ DSP mnemonic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the algebraic instruction set.

TMS320C55x Programmer's Guide (literature number SPRU376) describes ways to optimize C and assembly code for the TMS320C55x™ DSPs and explains how to write code that uses special features and instructions of the DSP.

TMS320C55x Assembly Language Tools User's Guide (literature number SPRU280) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for TMS320C55x™ devices.

TMS320C55x Optimizing C Compiler User's Guide (literature number SPRU281) describes the TMS320C55x™ C Compiler. This C compiler accepts ANSI standard C source code and produces assembly language source code for TMS320C55x devices.

Trademarks

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments Incorporated. Trademarks of Texas Instruments include: TI, Code Composer Studio, TMS320, TMS320C5000, and TMS320C55x.

Contents

Figures

Tables

Examples

Chapter 1

Introduction

This chapter provides an introduction to the TMS320C55x™ hardware extensions, a brief overview of their features and benefits, and a listing of extension components.

Topic **Page**

1.1 Introduction to the C55x Hardware Extensions

The TMS320C55x™ DSP core was created with an open architecture that allows the addition of application-specific hardware to boost performance on specific algorithms. The hardware extensions on the C5510 and C5509 DSPs strike the perfect balance of fixed function performance with programmable flexibility, while achieving low-power consumption, and cost that traditionally has been difficult to find in the video-processor market. The extensions allow the C5510 and C5509 DSPs to deliver exceptional video codec performance with more than half its bandwidth available for performing additional functions such as color space conversion, user-interface operations, security, TCP/IP, voice recognition and text-to-speech conversion. As a result, a single C5510 or C5509 DSP can power most portable digital video applications with processing headroom to spare.

1.2 Features and Benefits

In this document, a set of macros is provided in Appendix A to cover all C55x hardware extensions. You can use hardware extension macros to implement discrete cosine transform, motion estimation, or pixel interpolation.

For C programming, equivalent C-callable functions are provided as part of the TMS320C55x IMGLIB. Refer to *TMS320C55x Image/Video Processing Library Programmer's Reference* for more information.

The hardware extension features include:

❑ Efficient computation
❑ Low power consumption
❑ Availability in the 5509 and 5510 devices

There are three hardware extensions that are carefully tailored for the C55x DSP generation.

❑ DCT/IDCT hardware extension
❑ Pixel interpolation hardware extension
❑ Motion estimation hardware extension

Chapter 2

DCT/IDCT Hardware Extension

This chapter provides information on the Discrete Cosine Transform (DCT) and Inverse Discrete Cosine Transform (IDCT) hardware extensions.

Topic **Page**

2.1 DCT/IDCT Algorithms

The Discrete Cosine Transform (DCT) is described by the following equation:

$$I(u,v) = \frac{\alpha(u)\alpha(v)}{4} \sum_{x=0}^{7} \sum_{y=0}^{7} i(x,y) \cos\left(\frac{(2x+1)u\pi}{16}\right) \cos\left(\frac{(2y+1)v\pi}{16}\right)$$

where

$$z = 0 \Rightarrow \alpha(z) = \frac{1}{\sqrt{2}}$$
$$z \neq 0 \Rightarrow \alpha(z) = 1$$

The Inverse Discrete Cosine Transform (IDCT) is described by the following equation:

$$i(x,y) = \frac{\alpha(u)\alpha(v)}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} I(u,v) \cos\left(\frac{(2x+1)u\pi}{16}\right) \cos\left(\frac{(2y+1)v\pi}{16}\right)$$

where

$$z = 0 \Rightarrow \alpha(z) = \frac{1}{\sqrt{2}}$$
$$z \neq 0 \Rightarrow \alpha(z) = 1$$

2.2 DCT/IDCT Hardware Extension Description

Fast DCT/IDCT algorithms have been widely studied and several optimized versions exist for specific data sizes. These versions generally minimize the number of chained multiplies in order to avoid the problem of accuracy, while keeping the multiplier size small. The DCT/IDCT hardware extension described here is meant to support two image block sizes, 4x4 pixels and 8x8 pixels. It uses a recursive scheme for 1-D 4/8 points DCT/IDCT that is adapted to support 16-bit signed input data for both DCT and IDCT. Internal datapaths are defined to maintain the accuracy, following H263 algorithm recommendations. The basic steps to complete a 2-D 4×4/8×8 DCT/IDCT are as follows:

1) **Input data**. For the DCT, the macro block (4×4/8×8) is read-in. For the IDCT, DCT coefficient matrix is read-in.

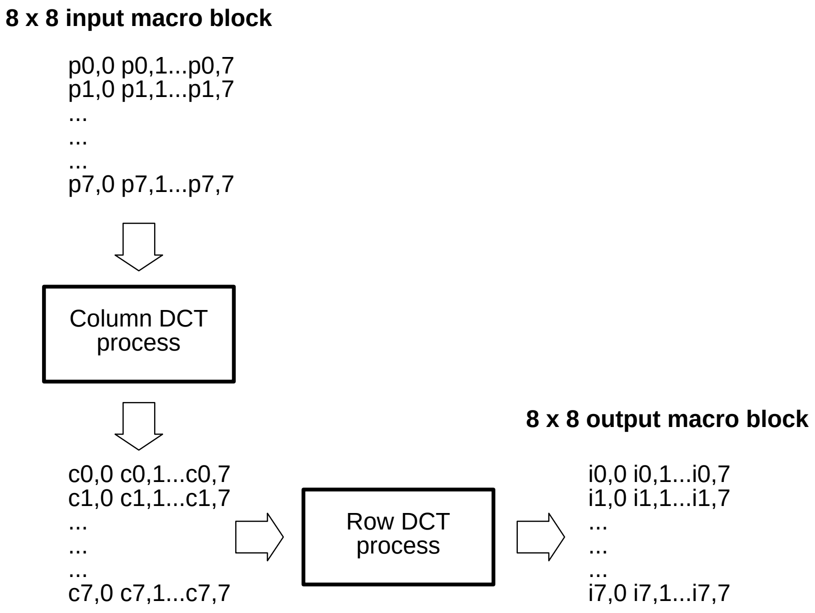
2) **Column DCT/IDCT process**. Process is performed column by column and temporary results are stored in an intermediate memory buffer row by row. In this way, transposition of the intermediate memory buffer is implicit.

3) **Row DCT/IDCT process**. Data read from the intermediate buffer is processed column by column again.

In order to get the maximum performance, the input block and intermediate memory must be located in different DARAM banks.

The advantages of this method are to avoid explicit transposition, and to let the column DCT/IDCT process and the row DCT/IDCT process contain almost the same operations. The DCT process is illustrated in Figure 2–1 (IDCT is almost identical to DCT process):

Figure 2–1. Process of 2-D 8x8 DCT



The sequence of operations to perform a DCT or IDCT is basically a set of calls to the DCT/IDCT hardware extension instructions.

2.3 DCT/IDCT Hardware Extension Instruction Set

All hardware extension instructions are organized into three different functional categories:

- ❑ load + computation + transfer to accumulators

  ACy = copr(k8, ACx, Xmem, Ymem)

- ❑ computation + transfer to accumulators + memory write

  ACy = copr(k8, ACx, ACy), Lmem=ACz

- ❑ Special instructions

  ACy = copr(k8, ACx, ACy)

k8	Instruction code
Xmem, Ymem	Input values
ACx, ACy	Computed and intermediated results

Each column 8x8 (or 4x4) DCT/IDCT transform can be decomposed into 8 cycles numbered 1 to 8. The following gives the k8 values for each cycle of the different DCT/IDCT transforms.

- ❑ The DCT or IDCT column is composed of these cycles and instructions:

Cycle	8x8 DCT	8x8 IDCT	4x4 DCT	4x4 IDCT
1	0x24	0x2d	0x3d	0x3d
2	0x20	0x2f	0x30	0x2f
3	0x21	0x2e	0x21	0x2e
4	0x33	0x3a	0x33	0x3a
5	0x32	0x3b	0x32	0x3b
6	0x26	0x29	0x36	0x39
7	0x27	0x28	0x37	0x38
8	0x25	0x2c	0x35	0x3c

- ❑ The DCT or IDCT row instructions are the same as the column DCT or IDCT instructions for cycles 1 through 7. Cycle 8 differs and contains these instructions:

Cycle	8x8 DCT	8x8 IDCT	4x4 DCT	4x4 IDCT
8	0x22	0x2a	0x31	0x34

- Two special instructions that execute between row and column DCT/IDCT processing are:

  0x23 for 8x8 DCT

  0x2b for 8x8 IDCT

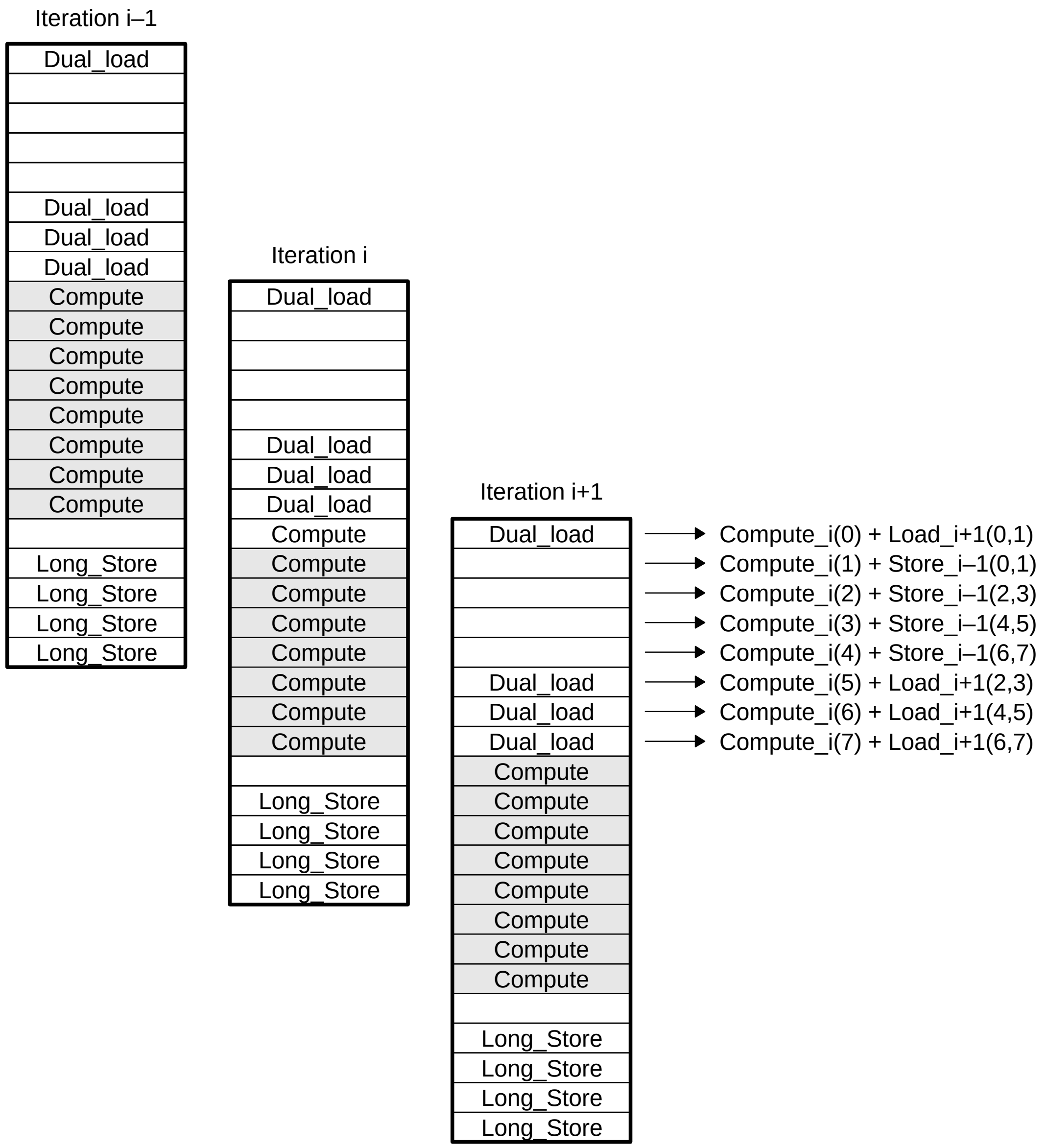
2.4 Source Code

There are two macros to perform 8x8 2-D DCT/IDCT, HWE_DCT_8x8 and HWE_IDCT_8x8. As an illustration of how to implement the hardware extension instruction, HWE_dct_8x8 is fully explained in this section. The source code of HWE_IDCT_8x8 can be found in Appendix A.

In order to simplify the description of HWE_DCT_8x8, the input macro block is denoted with $X=\{x_{ij}\}_{i,j=0..7}$ and output DCT coefficients with $Y=\{y_{ij}\}_{i,j=0..7}$. The transposition of Column-DCT coefficients is denoted with $T=\{t_{ij}\}_{i,j=0..7}$. The input buffer and intermediate buffer are necessary in the macro. The 8x8 input buffer holds input macro block (MB). The first row (8 words) of the 9x8 intermediate buffer holds temporary results, the remaining 8 rows (8x8=64words) hold column-DCT coefficients.

A typical load-compute-store sequence, for an 8x8 2-D DCT for instance, is illustrated in Figure 2–2.

Figure 2–2. Load-Compute-Store Sequence for 8x8 2-D DCT



The code in Example 2–1 shows the sequence run. The text lines in Helvetica are comments that have been inserted to help explain the instructions. The commentary text must be preceded by a semicolon (;) or removed in order to run the shown code.

Example 2–1. Load-Compute-Store Sequence for 8x8 2-D DCT

```
_HWE_DCT_8x8     .macro
```

Load column 0

```
   AC0 = copr(#0x24,AC0,*(AR2+T0),*(AR1+T0))
```

Load x_{00}, x_{10}, AR1 points to x_{00}, AR2 points to x_{10}

Compute step_1. The computation is not valid because there is not any loaded column in hardware extension.

```
   AC1 = copr(#0x26,AC0,*(AR2+T0),*(AR1+T0))
```

Load x_{20}, x_{30} AR1 points to x_{20}, AR2 points to x_{30}

Compute step_6. The computation is not valid because there is not any loaded column in hardware extension.

```
   AC0 = copr(#0x27,AC0,*(AR2+T0),*(AR1+T0))
```

Load x_{40}, x_{50} AR1 points to x_{40}, AR2 points to x_{50}

Compute step_7. The computation is not valid because there is not any loaded column in hardware extension.

```
   AC1 = copr(#0x25,AC0,*(AR2–T1),*(AR1–T1))
```

Load x_{60}, x_{70} AR1 points to x_{60}, AR2 points to x_{70}

Compute step_8. The computation is not valid because there is not any loaded column in hardware extension.

AR2–T1 and AR1–T1 in the code line above let AR1 point to x01. AR2 points to x11 after this instruction .

```
   localrepeat{
```

See Figure 2–2. Load column i+1, execute column i .

```
      AC0 = copr(#0x24,AC0,*(AR2+T0),*(AR1+T0))
```

Load $x_{0(i+1)}$, $x_{1(i+1)}$. AR1 points to $x_{0(i+1)}$, AR2 points to $x_{1(i+1)}$

Compute step_1 of column i.

```
      AC1 = copr(#0x20,AC0,AC1) , dbl(*AR3+)=AC0  ;
```

Compute step_2 of column i and store $t_{(i–1)0}$ and $t_{(i–1)1}$ to the intermediate buffer. When i=0, the two DCT coefficients stored to the intermediate buffer are invalid because there are not any well-done Column-DCT coefficients in hardware extension. So T_{00} and T_{01} are invalid.

Example 2–1.Load-Compute-Store Sequence for 8x8 2-D DCT (Continued)

```
        AC0 = copr(#0x21,AC1,AC0) , dbl(*AR3+)=AC1
```

Compute step_3 of column i and store $y_{2(i-1)}$ and $y_{3(i-1)}$ to the intermediate buffer. When i=0, the two DCT coefficients stored to the intermediate buffer are invalid because there are not any well-done Column-DCT coefficients in hardware extension. So T_{02} and T_{03} are invalid.

```
        AC1 = copr(#0x33,AC0,AC1) , dbl(*AR3+)=AC0
```

Compute step_4 of column i and store $y_{4(i-1)}$ and $y_{5(i-1)}$ to the intermediate buffer. When i=0, the two DCT coefficients stored to the intermediate buffer are invalid because there are not any well-done Column-DCT coefficients in hardware extension. So T_{04} and T_{05} are invalid.

```
        AC0 = copr(#0x32,AC1,AC0) , dbl(*AR3+)=AC1
```

Compute step_5 of column i and store $y_{6(i-1)}$ and $y_{7(i-1)}$ to the intermediate buffer. When i=0, the two DCT coefficients stored to the intermediate buffer are invalid because there are not well-done Column-DCT coefficients in hardware extension. So T_{06} and T_{07} are invalid.

The first row (8 words) in the intermediate buffer is not invalid, the Column-DCT coefficients of input macro block start at the second row in the intermediate buffer.

```
        AC1 = copr(#0x26,AC0,*(AR2+T0),*(AR1+T0))
```

Load $x_{2(i+1)}$, $x_{3(i+1)}$ AR1 points to $x_{2(i+1)}$, AR2 points to $x_{3(i+1)}$.

Compute step_6 of column i.

```
        AC0 = copr(#0x27,AC0,*(AR2+T0),*(AR1+T0))
```

Load $x_{4(i+1)}$, $x_{5(i+1)}$ AR1 points to $x_{4(i+1)}$, AR2 points to $x_{5(i+1)}$.

Compute step_7 of column i.

```
        AC1 = copr(#0x25,AC0,*(AR2–T1),*(AR1–T1))
```

Load $x_{6(i+1)}$, $x_{7(i+1)}$ AR1 points to $x_{6(i+1)}$, AR2 points to $x_{7(i+1)}$.

Compute step_8 of column i.

```
        }
```

Load row 0, execute column 7, store column 6.

```
            AC0 = copr(#0x24,AC0,*(AR5+T0),*(AR4+T0))
```

Load T_{00}, T_{10} AR4 points to T_{00}, AR5 points to T_{10}

Compute step_1 of column 7.

Example 2–1. Load-Compute-Store Sequence for 8x8 2-D DCT (Continued)

```
AC1 = copr(#0x20,AC0,AC1), dbl(*AR3+)=AC0
```

Compute step_2 of column 7 and store T_{06} and T_{16} to the intermediate buffer.

```
AC0 = copr(#0x21,AC1,AC0), dbl(*AR3+)=AC1
```

Compute step_3 of column 7 and store T_{26} and T_{36} to the intermediate buffer.

```
AC1 = copr(#0x33,AC0,AC1), dbl(*AR3+)=AC0
```

Compute step_4 of column 7 and store T_{46} and T_{56} to the intermediate buffer.

```
AC0 = copr(#0x32,AC1,AC0), dbl(*AR3+)=AC1
```

Compute step_5 of column 7 and store T_{66} and T_{76} to the intermediate buffer.

```
AC1 = copr(#0x26,AC0,*(AR5+T0),*(AR4+T0))
```

Load $T_{20,}$ T_{30} AR4 points to $T_{20,}$ AR5 points to T_{30}.

Compute step_6 of column 6.

```
BRC0 = #7
AC0 = copr(#0x27,AC0,*(AR5+T0),*(AR4+T0))
```

Load $T_{40,}$ T_{50} AR4 points to $T_{40,}$ AR5 points to T_{50}.

Compute step_7 of column 6.

```
AC1 = copr(#0x25,AC0,*(AR5–T1),*(AR4–T1))
```

Load $T_{60,}$ T_{70} AR4 points to $T_{60,}$ AR5 points to T_{70}.

Compute step_8 of column 6.

```
AC1 = copr(#0x23,AC0,AC1)
```

Special DCT instruction between column-DCT and row-DCT of MB.

Load row 1, execute row 0, store column 7.

```
AC0 = copr(#0x24,AC0,*(AR5+T0),*(AR4+T0))
```

Load $T_{01,}$ T_{11} AR4 points to $T_{01,}$ AR5 points to T_{11} .

Compute step_1 of row 0.

```
AC1 = copr(#0x20,AC0,AC1), dbl(*AR3+)=AC0
```

Compute step_2 of row 0 and store y_{07} and y_{17} to the intermediate buffer.

```
AC0 = copr(#0x21,AC1,AC0), dbl(*AR3+)=AC1
```

Compute step_3 of row 0 and store y_{27} and y_{37} to the intermediate buffer.

Example 2–1.Load-Compute-Store Sequence for 8x8 2-D DCT (Continued)

```
          AC1 = copr(#0x33,AC0,AC1), dbl(*AR3+)=AC0
```

Compute step_4 of row 0 and store y_{47} and y_{57} to the intermediate buffer.

```
          AC0 = copr(#0x32,AC1,AC0), dbl(*AR3+)=AC1
```

Compute step_5 of row 0 and store y_{67} and y_{77} to the intermediate buffer.

```
          localrepeat {
```

Load row i+1, execute row i, store row i–1 (i>0) or store column 7 (i=0).

```
          AC1 = copr(#0x26,AC0,*(AR5+T0),*(AR4+T0))
```

Load $T_{2(i+1),}$ $T_{3(i+1)}$ AR4 points to $T_{20,}$ AR5 points to T_{30}.

Compute step_6 of row i.

```
          AC0 = copr(#0x27,AC0,*(AR5+T0),*(AR4+T0))
```

Load $T_{4(i+1)\_,}$ $T_{5(i+1)}$ AR4 points to $T_{4(i+1),}$ AR5 points to $T_{5(i+1)}$.

Compute step_6 of row i.

```
          AC1 = copr(#0x22,AC0,*(AR5–T1),*(AR4–T1))
```

Load $T_{6(i+1)\_,}$ $T_{7(i+1)}$ AR4 points to $T_{6(i+1),}$ AR5 points to $T_{7(i+1)}$.

Compute step_7 of row i.

Load row i+2, execute row i+1, store line i.

```
          AC0 = copr(#0x24,AC0,*(AR5+T0),*(AR4+T0))
```

Load $T_{0(i+2),}$ $T_{1(i+2)}$ AR4 points to $T_{0(i+2),}$ AR5 points to $T_{1(i+2)}$.

Compute step_1 of row i+1.

```
          AC1 = copr(#0x20,AC0,AC1), dbl(*AR6+)=AC0
```

Compute step_2 of row i+1 and store y_{0i} and y_{1i} to the output buffer.

```
          AC0 = copr(#0x21,AC1,AC0), dbl(*AR6+)=AC1
```

Compute step_3 of row i+1 and store y_{2i} and y_{3i} to the output buffer.

```
          AC1 = copr(#0x33,AC0,AC1), dbl(*AR6+)=AC0
```

Compute step_4 of row i+1 and store y_{4i} and y_{5i} to the output buffer.

```
          AC0 = copr(#0x32,AC1,AC0), dbl(*AR6+)=AC1
```

Compute step_5 of row i+1 and store y_{6i} and y_{7i} to the output buffer.

```
          }
.endm
```

The code in Example 2–2 shows how to call the macros in assembly code. Comments (preceded by ;) have been inserted to provide additional explanation.

Example 2–2. Using Assembly to Call DCT and IDCT Macros

```
AR6 = #macro_block / AR6 = dct_coefficient
                        ; Pointer to the input data for 2-D DCT/IDCT.
BRC0 = #6
AR1 = AR6
                        ; Pointer to the even row of input data.
T0 = #0x10
                        ; Jump to the next even row for AR1 or jump to the next odd row for AR2
AR2 = AR1 + #8
                        ; Pointer to the odd row of input data
AR3 = #interm
                        ; Pointer to the intermediated buffer holding column DCT/IDCT coefficients
AR4 = AR3 + #8
                        ; Pointer to the even row of column DCT/IDCT result
AR5 = AR4 + #8
                        ; Pointer to the odd row of column DCT/IDCT result
T1 = #0x2f
                        ; Jump to the start point of next column
_HWA_DCT_8 /_HWA_IDCT_8
```

Implementation Notes

For maximum performance, the input data and output data must be located in different DARAM banks.

Benchmarks

As shown in Table 2–1, macro performance differs in relation to which memory type is used for the input data and intermediated buffer.

Table 2–1. DCT/IDCT Hardware Extension Macros Performance by Memory Type

Memory	**DARAM1**	**DARAM1**	**DARAM1**	**SARAM1**
Input data/intermediate buffer	**DARAM2**	**DARAM1**	**SARAM1**	**SARAM2**
DCT	151 cycles	207 cycles	206 cycles	226 cycles
IDCT	149 cycles	205 cycles	204 cycles	223 cycles

Chapter 3

Motion Estimation Hardware Extension

This chapter describes the motion estimation hardware extension, algorithm, and instruction set, and provides sample source code.

Topic **Page**

3.1 Motion Estimation Algorithm

Motion estimation is the most time-consuming part in video compression algorithms such as MPEG4 and H263. Basically, motion estimation is the technique to provide the minimum value of absolute difference (MAD) and the corresponding location (motion vector) between a 16x16 reference block and some blocks in a searching window.

Suppose $X = \{x_{ij}\}_{03i,j<16}$ is the 16x16 reference block and $Y = \{Y_{ij}\}_{0<=i,j<16}$ is the 16x16 macro block in the searching window. The macro block is sometimes called the search block. Sum of the absolute difference (SAD), or absolute difference (AD) for short, is defined as $AD = \Sigma_{0 \le i,j < 16}|x_{ij} - y_{ij}|$

Suppose that the center of the searching window (48x48 or 32x32) in the image is (m_1, n_1) and the center of the best-match searching block in the image is (m_2, n_2). Then, the motion vector is defined as follows:

$$(V_1, V_2) = (m_2 - m_1, n_2 - n_1).$$

There are several ways to categorize different motion estimation techniques.

First, the motion estimation techniques can be organized into two categories based on the searching strategy:

- ❑ Motion estimation with full searching
- ❑ Motion estimation with fast searching

Second, the motion estimation techniques can be organized in two categories based on the searching pixels:

- ❑ Pixel-based motion estimation
- ❑ Half-pixel-based motion estimation

Third, the motion estimation techniques can be organized in two categories based on the number of returned motion vectors:

- ❑ One motion vector (1 MV)
- ❑ Four motion vectors (4 MV)

The motion estimation with fast searching includes the following strategies:

- ❑ 3-step search (distances of 4,2,1)
- ❑ 4-step search (distances of 8, 4, 2, 1)
- ❑ 4-step search plus half-pixel refinement (distances of 8, 4, 2, 1 and ½).

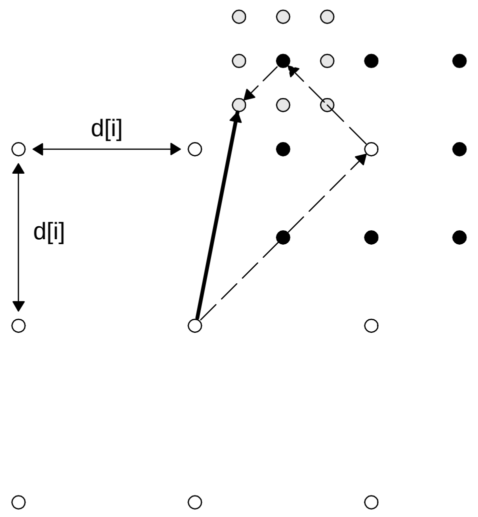
Motion estimation with full searching is straight forward. First, all absolute differences between the 16x16 reference block and all macro blocks in the searching window are calculated. Second, MAD and corresponding motion vector are computed.

Example 3–1 illustrates the motion estimation by the 4-step search method.

Example 3–1. Motion Estimation by 4-Step Search

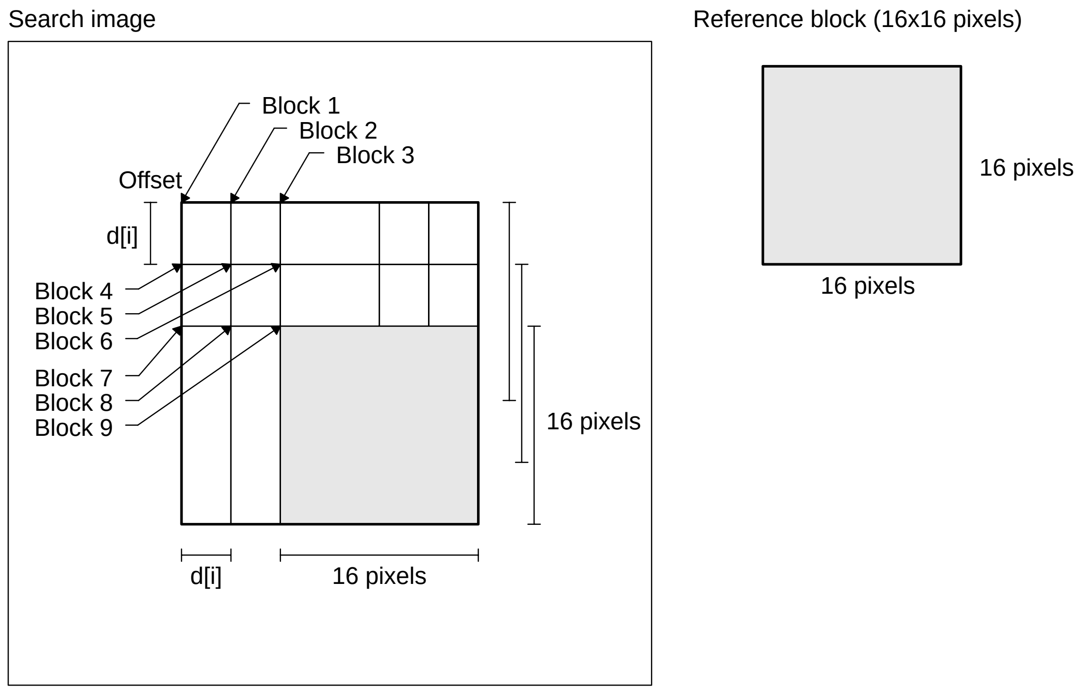
```
(a) Initialization
    d={8,4,2,1}
(b) Process:
    for( i=0; i<4; i++)
    {
        Compute three upper absolute differences for d[i].
        Compute three central absolute differences for d[i].
        Compute three lower absolute differences for d[i].
        Compute the minimum value of the 9-AD table (see Figure 3–1 )
        Start above process around the minimum location for the new dis-
        tance d[i+1].
    }
```

Figure 3–1. Motion Estimation With 4-Step Searching



- - ▸ - - Partial search
━━▶ Resulting vector
○ Center of search block when using d = 8
● Center of search block when using d = 4
◌ Center of search block when using d = 2

Figure 3–2. Computing Nine Absolute Differences for d[i]



3.2 Motion Estimation Hardware Extension Description

In order to compute nine absolute differences for a given distance, three identical operators are called using a pipelined mode. This means that a full scan is performed in three passes: the first pass computes the three upper points, then the second pass computes the three central points, and at last the three lower points are computed.

3.3 Motion Estimation Hardware Extension Instruction Set

The motion estimation hardware extension requires three 16-bit data for pixel carriage and three 16-bit absolute differences (ADs) that are computed and accumulated. Thus, the most useful motion estimation hardware extension operational mode is:

[ACx, ACy] = copr (k8, ACx, ACy, Xmem, Ymem, Coeff)

k8	The five LSB-bits of k8 are instruction code.	
	Bit4	0 in non-reset mode
	Bit3	1 enables AD3 (lower AD3
		0 disables AD3
	Bit2	1 enables AD2 (middle AD)
		0 disables AD2
	Bit1	1 enables AD1 (upper AD)
		0 disables AD1
	Bit0	Process data coming from an odd (1) or even (0) line of the search window
ACx, ACy	Accumulated ADs	
Xmem, Ymem	Pointer to a set of two adjacent pixels from the following two rows of the searching window. Xmem carries pixels from the odd line and Ymem carries pixels from the even line of the searching window.	
Coeff	Pointer to two adjacent pixels from the reference window	

The specification for the instruction code k8 is described in Table 3–1 and Table 3–2.

In the initialization mode, all important parameters of motion estimation hardware extension are set up including the searching distance and miscellaneous absolute difference configuration. After the initialization mode, the instructions belonging to the process mode are in charge of completing the motion estimation process.

Table 3–1. Specification of Instruction Code k8 in Process Mode

Hex	Reset Bit Bit4	AD3enable Bit3	AD2 enable Bit2	AD1 enable Bit1	Odd or Even Bit0	Description
00	0	0	0	0	0	All AD off; even line
01	0	0	0	0	1	All AD off; odd line
02	0	0	0	1	0	AD1 on; even line
03	0	0	0	1	1	AD1 on; odd line
06	0	0	1	1	0	AD1 and AD2 on; even line
07	0	0	1	1	1	AD1 and AD2 on; odd line
0E	0	1	1	1	0	All AD on; even line
0F	0	1	1	1	1	All AD off; odd line
0C	0	1	1	0	0	AD2 and AD3 on; even line
0D	0	1	1	0	1	AD2 and AD3 on; odd line
08	0	1	0	0	0	AD3 on; even line
09	0	1	0	0	1	AD3 on; odd line

Table 3–2. Specification of Instruction Code k8 in Initialization Mode

Hex	Reset Bit	Dist(2)	Dist(1)	Dist(0)	Not Used	Description
12	1	0	0	1	0	Set D to 1/2
14	1	0	1	0	0	All AD off; odd line
18	1	1	0	0	0	AD1 on; even line
1A	1	1	0	1	0	AD1 on; odd line
1C	1	1	1	0	0	AD1 and AD2 on; even line

3.4 Motion Estimation Macros

The different motion estimation techniques described in section 3.1 can be implemented with several macros that use the motion estimation hardware extension instruction set.

The set of macros shown here can be classified in the following three different functional categories:

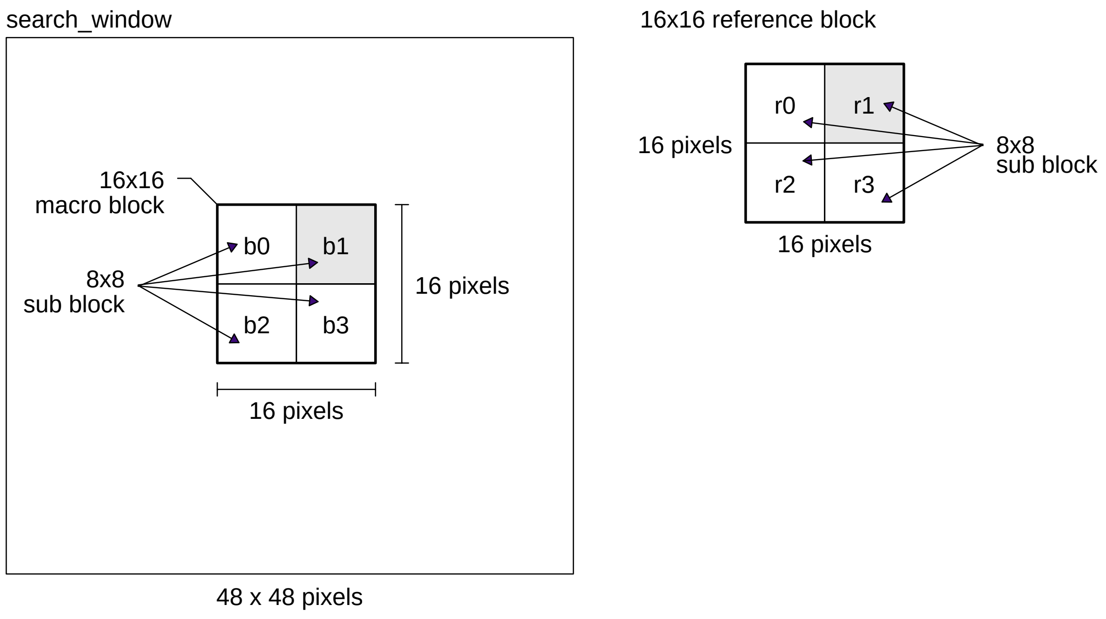
- ❑ Four Motion Vector and pixel-based macros
- ❑ One Motion Vector and pixel-based macros
- ❑ One Motion Vector and half-pixel-based macros

As an illustration of how to implement the motion estimation hardware extension instructions, the macro types are explained in this section, all other motion estimation macros can be found in Appendix A.

3.4.1 Four Motion Vector (MV) and Pixel-Based Macros

In the macros, the 16x16 reference block is divided into four 8x8 sub-blocks. Then, the motion vector and the minimum absolute difference (MAD) for the best-match 8x8 sub-block in the searching window are calculated (see Figure 3–3). To calculate the four motion vectors, you need to call the macros four times.

Figure 3–3. Motion Estimation With Four Motion Vectors



Two macros are needed to calculate the absolute difference between an 8x8 sub-block in the reference block and a 8x8 sub-block in the searching window. For example, in Figure 3–3 r_1 is a sub-block in 16x16 reference block and b_1 is a sub-block in search_window.

- ❑ HWE_ME_4MV_even macro calculates the absolute difference if the first pixel of the sub-block in the reference block is the first pixel of a 16-bit word. The result is held by AC0.
- ❑ HWE_ME_4MV_even macro calculates the absolute difference if the first pixel of the sub-block in the reference block is the second pixel of a 16-bit word. The result is held by AC0.

Note:

The reference block and searching window must be aligned on a 32-bit boundary. The reference block and searching window must be located in different DARAM banks for optimal cycle performance.

As an illustration of how to implement the Four Motion Vector and Pixel-Based Macros, the macro HWE_ME_4MV_even is illustrated in Example 3–2.

In order to simplify the description of HWE_DCT_8x8, input sub-block in reference block is denoted with $R=\{r_{ij}\}_{i,j=0..7}$ and sub-block in the searching window is denoted with $S=\{s_{ij}\}_{i,j=0..7}$.

The text in Helvetica is comments that have been inserted to help explain the instructions, which are shown in Courier. The commentary text must be preceded by a semicolon (;) or removed in order to run the shown code.

Example 3–2. _HWE_ME_4MV_even Macro

```
BRC0 = #6    ;repeat 7 times
AC0,AC1 = copr(#0x5c,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
```

Set up the hardware extension, load s_{00} ,s_{01}, r_{00} and r_{01} in the hardware extension, initialize SAD. AR0 points to s_{00} and s_{01}. CDP points to r_{00} and r_{01}.

```
AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
```

SAD1 is on, load s_{02} ,s_{03}, r_{02} and r_{03} in the hardware extension, accumulate SAD, AR0 points to $s_{02}$and s_{03}. CDP points to r_{02} and r_{03}.

```
AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
```

SAD1 is on, load s_{04} ,s_{05}, r_{04} and r_{05} in the hardware extension, accumulate SAD, AR0 points to $s_{04}$and s_{05}. CDP points to r_{04} and r_{05}.

```
AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*(CDP+T0)))
```

SAD1 is on, load s_{06} ,s_{07}, r_{06} and r_{07} in the hardware extension, accumulate SAD, AR0 points to s_{06} and $s_0$7. CDP points to r_{06} and r_{07}. After this instruction, AR0 points to the next row of s_{01}. CDP points to the next row of r_1.

Example 3–2._HWE_ME_4MV_even Macro (Continued)

```
localrepeat{
AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
```

SAD1 is on, load s_{i0} ,s_{i1}, r_{i0} and r_{i1} in the hardware extension, accumulate SAD, AR0 points to $s_{io}$and s_{i1}. CDP points to r_{i0} and r_{i1}. After this instruction, AR0 points to the next row of s_1. CDP points to the next row of r_1.

```
AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
```

SAD1 is on, load s_{i2} ,s_{i3}, r_{i2} and r_{i3} in the hardware extension, accumulate SAD, AR0 points to $s_{i2}$and s_{i3}. CDP points to r_{i2} and r_{i3}. After this instruction, AR0 points to the next row of s_1. CDP points to the next row of r_1.

```
AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
```

SAD1 is on, load s_{i4} ,s_{i5}, r_{i4} and r_{i5} in the hardware extension, accumulate SAD, AR0 points to s_{i4} and s_{i5}. CDP points to ri0 and ri1. After this instruction, AR0 points to the next row of s_1. CDP points to the next row of r_1.

```
AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*(CDP+T0)))
```

SAD1 is on, load $s_{i6}s_{i7}$ r_{i6} and r_{i7} in the hardware extension, accumulate SAD, AR0 points to $s_{i6}$and s_{i7}. CDP points to ri6 and ri7. After this instruction, AR0 points to the next row of s_1. CDP points to the next row of r_1.

```
}
AC0,AC1 = copr(#0x43,AC0,AC1,*AR0,*AR1,coef(*CDP))
```

SAD1 is on. Accumulate SAD1 based on the pixels loaded in previous instruction.

```
AC0,AC1 = copr(#0x40,AC0,AC1,*AR0,*AR1,coef(*CDP))
```

Reset

```
            .endm
```

The code in Example 3–3 shows how to call the macros in assembly code. The comments provide additional explanation.

Example 3–3. Calling Motion Estimation Macros in Assembly Code

```
AR0=search_window + offset + (8/2) ; Pointer on the 8x8 sub-block b1 of the 16x16
                                   ; macro block in the searching window.
AR1=AR0+48                         ; Pointer on the next even rows of sub-block in
                                   ; searching window
CDP=ref_block +4                   ; Pointer on the 8x8 sub-block r1 of the 16x16
                                   ; reference block
T0=5                              ; Pitch on the reference 16x16 block
T1=44                             ; Pitch on the search_window
HWE_ME_4MV_odd /
HWE_ME_4MV_even
```

Benchmark:

Assuming the reference block and searching window are located in different DARAM banks:

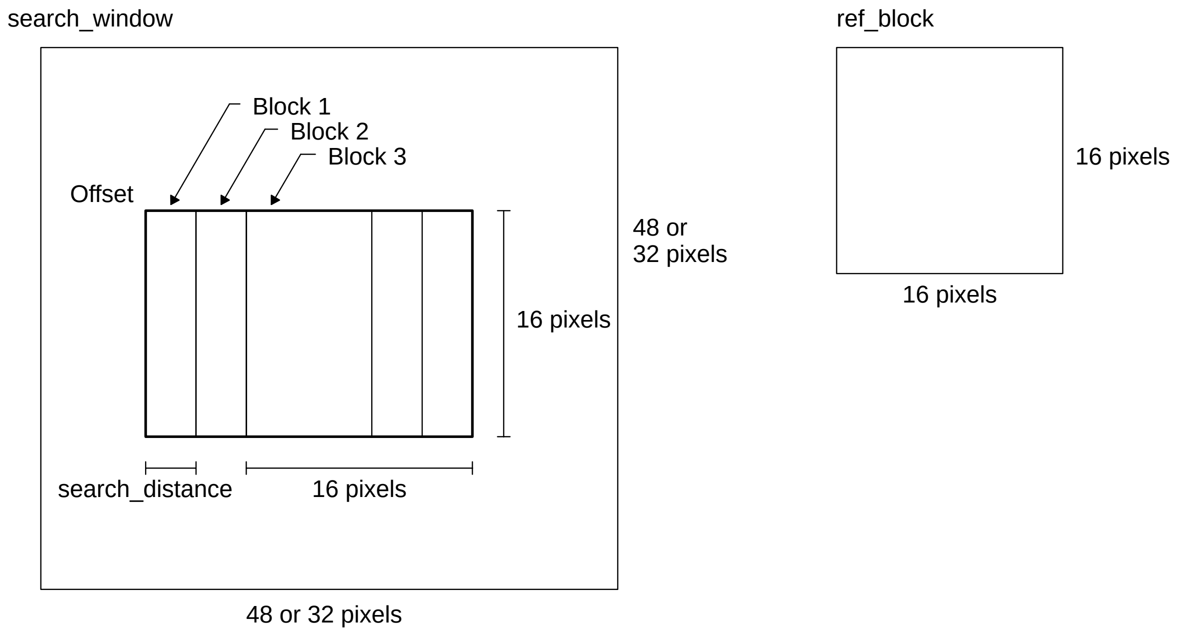
HWE_ME_4MV_odd	42 cycles
HWE_ME_4MV_even	37 cycles

3.4.2 One Motion Vector and Pixel-Based Macros

Given a distance, these macros return three absolute differences between a 16x16 reference block and the three macro blocks in a searching window (typically of size 48x48 or 32x32). To compute the nine absolute differences for a given distance, it is necessary to call the corresponding macro three times as shown in the *TMS320C55x Image/Video Processing Library Programmer's Reference*.

Figure 3–4 shows how to calculate three absolute differences between three blocks (block1, block2, and block3) and ref_block using the HWE_ME_8, HWE_ME_4, HWE_ME_2, and HWE_ME_1 macros. The C55x IMGLIB provides the function IMG_mad_16x16_4step that calls these macros.

Figure 3–4. Three Absolute Differences Between Three Blocks



Here is the description of the macros.

- The **HWE_ME_8** macro calculates the three absolute differences between the reference block and the three macro blocks in one row in the searching window. The distance between the top-left pixel of adjacent macro blocks is 8. Low part of AC0 holds AD1; High part of AC0 holds AD2; Low part of AC1 holds AD3.
- The **HWE_ME_4** macro calculates the three absolute differences between the reference block and the three macro blocks in one row in the searching window. The distance between the top-left pixel of adjacent macro blocks is 4. Low part of AC0 holds AD1; High part of AC0 holds AD2; Low part of AC1 holds AD3.
- The **HWE_ME_2** macro calculates the three absolute differences between the reference block and the three macro blocks in one row in search window. The distance between the top-left pixel of adjacent macro blocks is 2. Low part of AC0 holds AD1; High part of AC0 holds AD2; Low part of AC1 holds AD3.
- The **HWE_ME_1** macro calculates the three absolute differences between the reference block and the macro blocks block in one row in search window. The distance between the top-left pixel of adjacent macro blocks is 1. Low part of AC0 holds AD1; High part of AC0 holds AD2; Low part of AC1 holds AD3.

HWE_ME_8, _HWE_ME_4, and _HWE_ME_2 can only compute absolute differences for the blocks that are the first pixel at the beginning of a word.

HWE_ME_1 can only compute absolute differences for the blocks that are the second pixel at the beginning of a word.

The reference block and the search_window must be mapped at an even address if the HWE_PI is directly on the search_window.

The source code can be found in Appendix A. Example 3–4 illustrates how to call the above macros in assembly code.

Example 3–4. Calling One Motion Vector and Pixel-Based Macros in Assembly Code

```
BRC0 = #6                      ; Set loop for the macro (all the time BRCx = 6)
CDP=#ref_block                 ; Pointer on the reference block
AR0=#search_window + offset    ; Pointer on the first line of the block you have
                               ; chosen in the search_window
AR1 = AR0 + #((3*16)/2)        ; Pointer to the next line
T0 = #33                       ; (search_window width unpacked)– 15
HWE_ME_8
HWE_ME_4
HWE_ME_2
HWE_ME_1
```

Benchmark:

Assuming the reference block and searching window are located in different DARAM banks:

HWE_ME_8	159 cycles
HWE_ME_4	154 cycles
HWE_ME_2	150 cycles
HWE_ME_1	152 cycles

3.4.3 One Motion Vector and Half-Pixel-Based

Half-pixel motion estimation is needed in some video compression application. Before half-pixel motion estimation, pixel interpolation is needed. (see pixel interpolation section).

Here are the descriptions of corresponding macros:

❑ The **HWE_ME_half_1** macro computes two absolute differences between three 16x16 blocks. One is the reference block packed (128 words) and the two others are adjacent interpolated 6x16 blocks packed but organized by the pixel interpolation hardware extension output.

❑ The **HWE_ME_half_2** macro computes the absolute differences between two 16x16 blocks. One is the reference block packed (128 words) and the other are interpolated 16x16 blocks packed but organized by the pixel interpolation hardware extension output.

❑ The **HWE_ME_half_3** macro computes two absolute differences between three 8x8 blocks. One is the reference block packed (128 words) and the two others are interpolated 8x8 blocks packed but organized by the pixel interpolation hardware extension output

❑ The **HWE_ME_half_4** macro computes the absolute differences between three 8x8 blocks. One is the reference block packed (128 words) and the two adjacent others are interpolated 8x8 blocks packed but organized by the pixel interpolation hardware extension output.

Example 3–5 and Example 3–6 illustrate how to call the macros in the assembly code. The text in Helvetica is comments that have been inserted to help explain the instructions, which are shown in Courier. The commentary text must be preceded by a semicolon (;) or removed in order to run the shown code.

Example 3–5. Calling HWE_ME_half_1 and HWE_ME_half_2 in Assembly Code

Offset from the interpolated_pixels array base address to start on a particular interpolated macro block.

```
CDP = #ref_block
```

Pointer on the reference block.

```
AR0 = AR0 + #interp_pixels
```

Pointer on the first line of the block you have chosen among the interpolated blocks.

```
AR1 = AR0 + #36
```

36 is a constant independent of your implementation; it is fixed when using the pixel interpolation hardware extension.

```
T0 = #42
```

42 is a constant independent of your implementation; it is fixed when using the pixel interpolation hardware extension.

```
_HWE_ME_half_1 /  HWE_ME_half2
```

Example 3–6. Calling HWE_ME_half_3 and HWE_ME_half_4 in Assembly Code

```
BRC0 = #2
AR2 = AR2 + #ref_block
```

AR2 is a parameter of the macro it must be equal to CDP .

```
CDP = AR2
AR0 = AR0 + #interp_pixels
```

Pointer in the first pixel of a particular block.

```
AR1 = AR0 + #36
```

Pointer to the next line. 36 is a constant independent of your implementation; it is fixed by the pixel interpolation hardware extension.

```
 T1 = #2
```

2 is a constant independent of your implementation; it is fixed by the pixel interpolation hardware extension.

```
 T0 = #64
```

64 is a constant independent of your implementation; it is fixed by the pixel interpolation hardware extension.

```
 DR2 = #4
```

4 is a constant independent of your implementation; it is fixed by the pixel interpolation hardware extension.

```
HWE_ME_half_3
HWE_ME_half_4
```

Benchmark:

HWE_ME_half_1	153 cycles
HWE_ME_half_2	152 cycles
HWE_ME_half_3	87 cycles
HWE_ME_half_4	84 cycles

Chapter 4

Pixel Interpolation Hardware Extension

This chapter describes the pixel interpolation hardware extension, including the algorithm, description, instruction set and sample source code.

Topic **Page**

4.1 Pixel Interpolation Hardware Extension Algorithm

Pixel interpolation is an important part in video compression algorithms such as MPEG4 and H263. Pixel interpolation can be used in video encoding and video decoding.

4.1.1 Pixel Interpolation for Video Encoding

The hardware extension implements a half-pixel interpolation algorithm. Three sub-pixels (U, M, R) belonging to a square of pixels (A, B, C, D) are computed using the following equations:

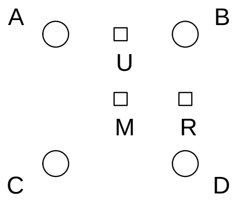
$$U = \frac{A + B + Rnd}{2}$$

$$M = \frac{A + B + C + D + 2 * Rnd}{4}$$

$$R = \frac{B + D + Rnd}{2}$$

Figure 4–1 illustrates the relationship of the pixels and sub-pixels in pixel interpolation for video encoding.

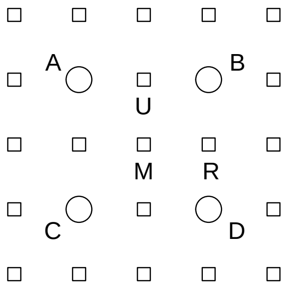
Figure 4–1. Pixel Interpolation for Video Encoder



Depending on the controls given to the hardware extension during the init phase, results can optionally be rounded by addition of ½ LSB (by setting *Rnd* to 1), so that pixel resolution is kept.

To obtain a full pixel interpolation on a XxX pixels block, the pixel interpolation equations have to be applied on the (X+2)x(X+2) corresponding block. Figure 4–2 illustrates an example for X=2, from an original block of 2x2 pixels:

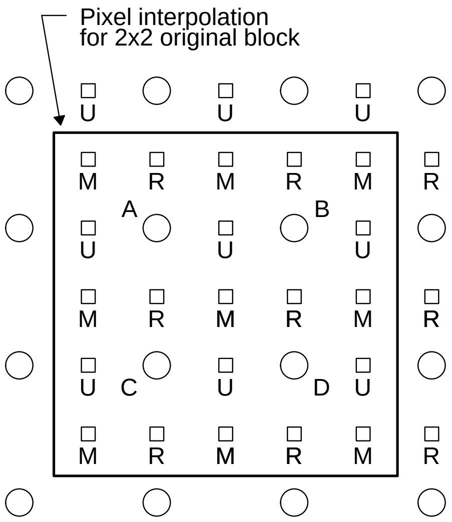
Figure 4–2. Pixel Interpolated Result for 2x2 Original Pixels



In Figure 4–1 through Figure 4–6, the squares are interpolated pixels and the circles are original pixels.

In order to get the interpolated result of a 2x2 original block (A, B, C, and D) as shown in Figure 4–2, a 4x4 extended block is needed for the computation. Figure 4–3 illustrates this 4x4 extended block, which consists of all open circles.

Figure 4–3. Pixel Interpolation Process for the 4x4 Extended Block



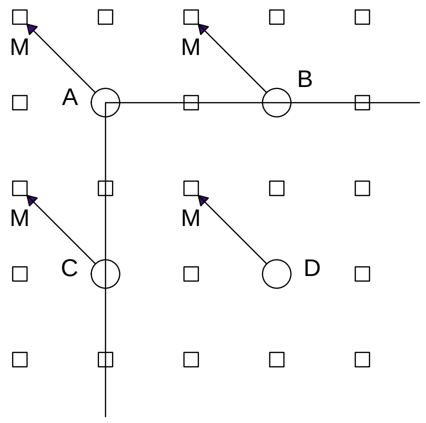
During pixel interpolation of an original 16x16 pixel block, keep these items in mind:

- ❑ The macros do not operate in a 16x16 original macro block, but in a 18x18 extended block called Macro Block Extension (MBE).
- ❑ The pixel-interpolated block size is 33x33.

4.1.2 Pixel Interpolation for Video Decoding

In the decoder part of the video algorithm, a half pixel best matching macro block must be computed. It is built with the best matching macro block at d=1 and a half pixel motion vector, result of the motion estimation at d=1/2. The hardware extension also has a decoding functionality to reconstruct it. Only one of the three computations (U, M, or R) is needed. The hardware has enough resources to process two identical computations in parallel each cycle. In Figure 4–4, only the M pixels are computed by the hardware extension. The selection of the M pixels is determined by the motion vector.

Figure 4–4. Pixel Interpolation for Video Decoder



4.2 Pixel Interpolation Hardware Extension Description

The block does not have to be stored locally. It is directly fetched from the full image zone. The macro block to interpolate is 16x16 pixels.

The MBE(macro block + pixels extension) that must be fetched in order to interpolate is 18x18 pixels.

Considering that read access in the memory is 32 bits, an 18x20 pixel block (where 20 is the multiple of 4 that is the nearest to 18) must be fetched. These conditions lead to four different alignment configurations for the MBE.

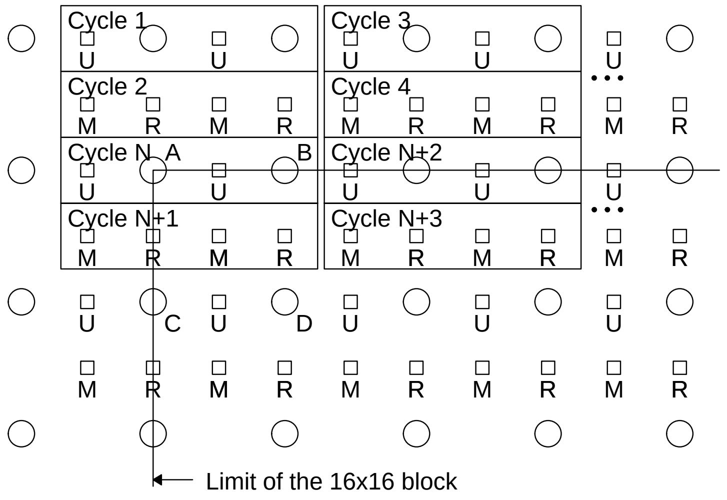
The four upper-left pixels of the MBE can be:

- ❑ Doubleword aligned: `| o o o o |`
- ❑ One byte right from Doubleword aligned: `|   o o o | o`
- ❑ Two bytes right from Doubleword aligned: `|     o o | o o`
- ❑ Three bytes right from Doubleword aligned: `|       o | o o o`

To handle these four disalignment cases, a parameter passed to the accelerator during the init phase defines which case is the current one. The corresponding subprogram is executed (so four variants of the interpolation encoding routine will exist). It is assumed that a full image is organized in memory so the first pixel of each line has the same alignment. The consequence is that the first pixel of each line of the MBE also has the same alignment.

In the coding mode of the hardware extension, the interpolated zone is provided by the hardware extension as illustrated in Figure 4–5.

Figure 4–5. Number of Cycles for Interpolation Operation



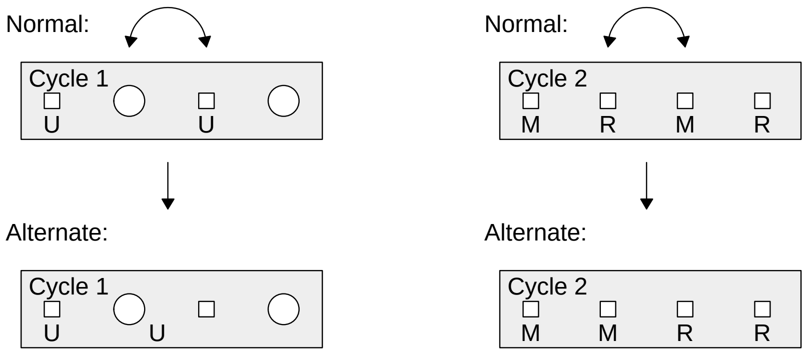
Assuming that in Figure 4–5 N is equal to 19, a row of the interpolated block is 36 (9×4) pixels long. Only the first 33 pixels are useful.

As you can see on the picture, the first line of the interpolated zone is useless. So from a 36x34 window, only the lower 33x33 part corresponds to the interpolated block you are interested in. The Cycle 1, Cycle 2... values do not take into account the number of cycles needed to launch internal pipes of the hardware extension.

In Figure 4–5 the interpolated pixels are mixed with original pixels. This is problematic because in a video algorithm, a three absolute difference motion estimation is based on the interpolated block. The first and third absolute differences are the comparison between the reference and 16x16 macro blocks made of only M pixels. The second absolute difference is a comparison between the reference and a 16x16 macro block made of only R pixels. Because the hardware extension fetches search pixels by words, having interpolated and original pixels mixed makes the motion estimation impossible. That is why the pixel interpolation hardware extension has an alternative output pixel organization where each type of pixels is contiguous: M pixels, R pixels, original pixels, and U pixels.

This alternate method consists in swapping pixel 2 and pixel 3 of the output of the hardware extension as shown in Figure 4–6.

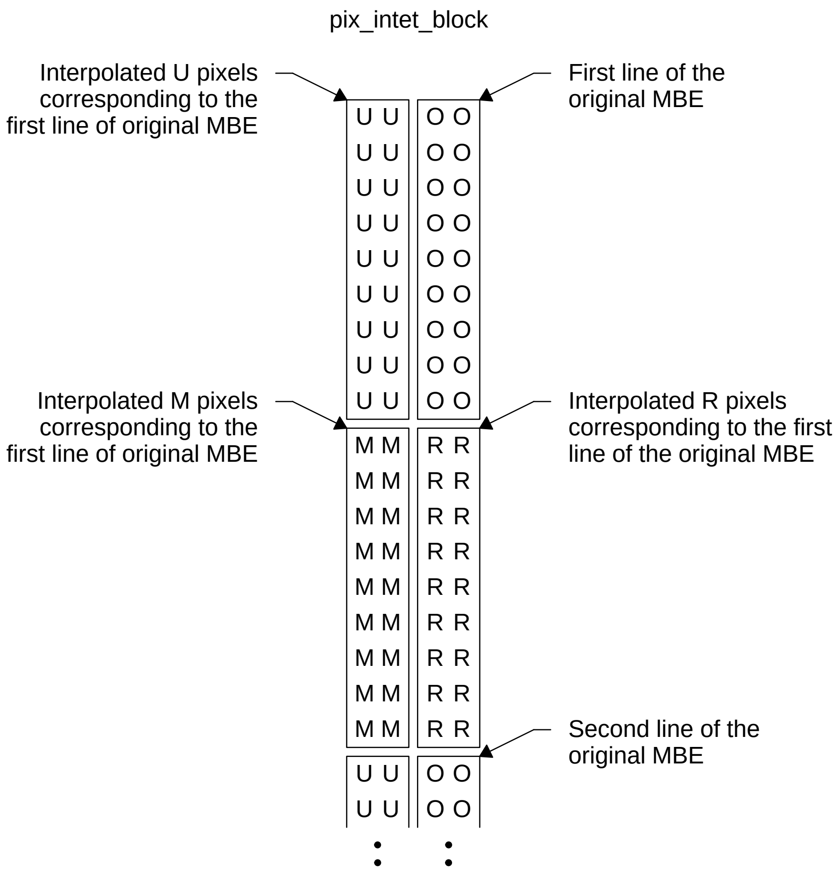
Figure 4–6. Swapping Pixel 2 and Pixel 3



With this method the ME can fetch, by 16-bit (two pixels) packets, either only U pixels, or M pixels, or R pixels, or original pixels. The ME routine for d=1/2 supposes that the search zone is organized this way.

Finally, the alternate output of pixel interpolation for encoder is organized as shown in Figure 4–7.

Figure 4–7. Pixel Interpolation Result of Separated Original Pixels and Interpolated Pixels


pix_intet_block
Interpolated U pixels corresponding to the first line of original MBE
First line of the original MBE
Interpolated M pixels corresponding to the first line of original MBE
Interpolated R pixels corresponding to the first line of the original MBE
Second line of the original MBE
U U
O O
M M
R R

4.3 Pixel Interpolation Hardware Extension Instruction Set

The pixel interpolation extension requires two 16-bit data for pixels carriage and two 32-bit pixel carriage channels that write back results to internal accumulators. Thus, the data flows used by the accelerator are of the kind:

❑ Loading pixels and compute

ACy=copr(k8, ACx,Lmem)

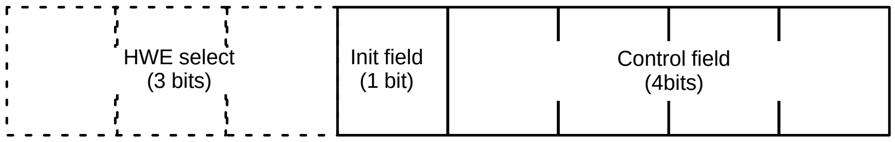
❑ Loading pixels, compute, and store

ACy=copr(k8,ACx,Lmem) || Lmem=ACz

k8	Instruction code
Lmem	Carry 4 pixels along a line of the image block
ACx	Carry the address of any internal register in read or write emulation mode
ACy	Interpolated pixels (output)

The instruction code K8 is organized as shown in Figure 4–8:

Figure 4–8. Organization of the Pixel Interpolation Instruction Code (K8)



The upper three bits are used to select the hardware extension that is activated for the current instruction. In this case, these three bits must be set to 000 so the pixel interpolation hardware extension is on, while others are off. The five lower bits allow sending controls to the hardware extension for the current cycle.

When the Init field is set to 0, the Control field is useless, and the hardware extension is in Init mode. Only the settings in ACx are useful in Init mode. Every routine begins with an instruction with the 8 fields set to 0x00 (hardware extension pixel interpolation activated + Init mode). So before calling such a routine, you must set ACx to the correct value.

When the Init field is set to 1, the hardware extension is in Running mode and the K8 Control field sets the internal controls for the current cycle.

4.3.1 Initialization Mode

The initialization of the pixel interpolation hardware extension sets several controls inside the device. The main information that should to be set during this stage is:

- **Mode bit**. The coder or decoder mode that the device works on.

  When this bit is set to 1, the CODER mode is enabled.

  When this bit is set to 0, the DECODER mode is enabled.

- **Rounding bit**. Which rounding is enabled for the computations.

  When this bit is set to 1, the rounding method is the addition of a ½ LSB.

  When this bit is set to 0, there is no rounding, just truncation of the results.

- **Two vector/output bits**. The half-pixel vector that is interesting (when decoder mode activated).

  When these two bits are set to 01, the vector is the number 1. Its coordinates are [–1/2;0]. It corresponds to the calculation of a U pixel.

  When these two bits are set to 10, the vector is the number 2. Its coordinates are [–1/2;–1/2]. It corresponds to the calculation of a M pixel.

  When these two bits are set to 11, the vector is the number 3. Its coordinates are [0;1/2]. It corresponds to the calculation of a R pixel.

  These values are useful only in decoder mode. This field becomes the Output Mode if Mode bit is set to 1. In this mode:

  When these two bits are set to 00, the output management is the normal one, where interpolated and original pixels are mixed.

  When these two bits are set to any other value, the output management is the alternate one, where both interpolated and original pixels are contiguous.

- **Two disalignment bits**. The type of disalignment for the pixel interpolation hardware extension.

  These two bits describe four different disalignment cases. (See section 4.2 on page 4-5.)

  At this time, the controls are updated with the values of the lower 5 bits of the ACx of the current instruction, as follows:

Mode bit	Bit 5 of ACx
Rounding bit	Bit 4 of ACx
Vector bits	Bits 3–2 of ACx
Disalignment bits	Bits 1–0 of ACx

4.3.2 Running Mode

Running mode is set when the fifth bit of the controls field is set to 1. The four lower bits are then exported to the hardware extension. These devices are:

- Upper input shift registers
- Lower input shift registers
- Partial results registers
- Complete results registers

The four field bits affect the registers as shown in Table 4–1. These controls combinations are needed in order to implement coder and decoder pixel interpolation.

Table 4–1. Five Lower Control Bits Impact on Registers of the Hardware Extension

Five Lower Control Bits of K8	Upper Input Shift Registers	Lower Input Shift Registers	Partial Results Registers	Complete Results Registers
0x10	Load value, style 1	No Load	No Load	Load
0x11	No Load	Load value, style 1	Load	No Load
0x12	No Load	No Load	No Load	Load
0x13	No Load	No Load	Load	No Load
0x14	Load value, style 1	No Load	Load	No Load
0x15	No Load	Load value, style 1	No Load	Load
0x16	Load value, style 2	No Load	No Load	Load
0x17	No Load	Load value, style 2	Load	No Load
0x18	Load value, style 3	No Load	No Load	Load
0x19	No Load	Load value, style 3	Load	No Load
0x1A	Load value, style 4	No Load	Load	No Load
0x1B	No Load	Load value, style 4	No Load	Load
0x1C	Load value, style 5	No Load	Load	No Load
0x1D	No Load	Load value, style 5	No Load	Load

4.4 Source Code

Only pixel interpolation for the encoder is discussed in this chapter. There are four variants of the encoding routine. One variant for each possible case of disagreement of the original MBE:

- ❏ HWE_PI_16x16_0
- ❏ HWE_PI_16x16_1
- ❏ HWE_PI_16x16_2
- ❏ HWE_PI_16x16_3

These four macros give the you enough options to implement pixel interpolation in video encoder. It is not necessary to develop new pixel interpolation hardware extension routines for video encoder. So, it is easy for you to skip the details of these routines. You should focus how to use these routines correctly and efficiently. Many good examples can be found in *TMS320C55x Image/Video Processing Library Programmer’s Reference*. The source code of other routines can be found in Appendix A.

Example 4–1 shows how to call the listed macros in assembly code.

Example 4–1. Calling Pixel Interpolation in Video Encoder

```
AR2 = #offset               ; Offset from src buffer base addr
; Source
AR2 = AR2 + #src1           ; First line src address
T0 = #16                   ; Jump of 2 WORDS to the next line
AR3 = AR2 + #(48/2)         ; Second line src address
; Destination
AR0 = #dst                   ; First line stored value array addr
AR1 = AR0 + #18              ; Second line stored value addr
T1 = #0x14                  ; Jump between two line of stored values
AC0 = #0x35                  ; Upper Rounding, Disalignment 1
HWE_PI_16x16_1
HWE_PI_16x16_2
HWE_PI_16x16_3
HWE_PI_16x16_4
```

Appendix A

Source Code for Hardware Extensions

This appendix provides the source code for the hardware extension macros in alphabetic order.

Topic **Page**

A.1 HWE_DCT_8x8

```
_HWE_DCT_8x8   .macro
; load column 0
            AC0 = copr(#0x24,AC0,*(AR2+T0),*(AR1+T0))
            AC1 = copr(#0x26,AC0,*(AR2+T0),*(AR1+T0))
            AC0 = copr(#0x27,AC0,*(AR2+T0),*(AR1+T0))
            AC1 = copr(#0x25,AC0,*(AR2–T1),*(AR1–T1))
            localrepeat{
; load column i+1, execute column i
            AC0 = copr(#0x24,AC0,*(AR2+T0),*(AR1+T0))
            AC1 = copr(#0x20,AC0,AC1) , dbl(*AR3+)=AC0
            AC0 = copr(#0x21,AC1,AC0) , dbl(*AR3+)=AC1
            AC1 = copr(#0x33,AC0,AC1) , dbl(*AR3+)=AC0
            AC0 = copr(#0x32,AC1,AC0) , dbl(*AR3+)=AC1
            AC1 = copr(#0x26,AC0,*(AR2+T0),*(AR1+T0))
            AC0 = copr(#0x27,AC0,*(AR2+T0),*(AR1+T0))
            AC1 = copr(#0x25,AC0,*(AR2–T1),*(AR1–T1))
            }
; load row 0, execute column 7, store column 6
            AC0 = copr(#0x24,AC0,*(AR5+T0),*(AR4+T0))
            AC1 = copr(#0x20,AC0,AC1), dbl(*AR3+)=AC0
            AC0 = copr(#0x21,AC1,AC0), dbl(*AR3+)=AC1
            AC1 = copr(#0x33,AC0,AC1), dbl(*AR3+)=AC0
            AC0 = copr(#0x32,AC1,AC0), dbl(*AR3+)=AC1
            AC1 = copr(#0x26,AC0,*(AR5+T0),*(AR4+T0))
            BRC0 = #7
            AC0 = copr(#0x27,AC0,*(AR5+T0),*(AR4+T0))
            AC1 = copr(#0x25,AC0,*(AR5–T1),*(AR4–T1))
;special DCT mode.
            AC1 = copr(#0x23,AC0,AC1)
;load row 1, execute row 0, store column 7
            AC0 = copr(#0x24,AC0,*(AR5+T0),*(AR4+T0))
            AC1 = copr(#0x20,AC0,AC1), dbl(*AR3+)=AC0
            AC0 = copr(#0x21,AC1,AC0), dbl(*AR3+)=AC1
            AC1 = copr(#0x33,AC0,AC1), dbl(*AR3+)=AC0
            AC0 = copr(#0x32,AC1,AC0), dbl(*AR3+)=AC1
            localrepeat {
; load row i+1, execute row i, store row i–1 (i>0) or store column 7 (i=0).
            AC1 = copr(#0x26,AC0,*(AR5+T0),*(AR4+T0))
            AC0 = copr(#0x27,AC0,*(AR5+T0),*(AR4+T0))
            AC1 = copr(#0x22,AC0,*(AR5–T1),*(AR4–T1))
; load row i+2, execute row i+1, store line i.
            AC0 = copr(#0x24,AC0,*(AR5+T0),*(AR4+T0))
            AC1 = copr(#0x20,AC0,AC1), dbl(*AR6+)=AC0
            AC0 = copr(#0x21,AC1,AC0), dbl(*AR6+)=AC1
            AC1 = copr(#0x33,AC0,AC1), dbl(*AR6+)=AC0
            AC0 = copr(#0x32,AC1,AC0), dbl(*AR6+)=AC1
            }
.endm
```

A.2 HWE_IDCT_8x8

```
_HWE_iDCT_8   .macro
; IDCT N=8
; load column 0
            AC0 = copr(#0x2d,AC0,*(AR2+T0),*(AR1+T0))
            AC1 = copr(#0x29,AC0,*(AR2+T0),*(AR1+T0))
            AC0 = copr(#0x28,AC0,*(AR2+T0),*(AR1+T0))
            AC1 = copr(#0x2c,AC0,*(AR2–T1),*(AR1–T1))
            localrepeat {
; load column i+1, execute column i
            AC0 = copr(#0x2d,AC0,*(AR2+T0),*(AR1+T0))
            AC1 = copr(#0x2f,AC0,AC1) , dbl(*AR3+)=AC0
            AC0 = copr(#0x2e,AC1,AC0) , dbl(*AR3+)=AC1
            AC1 = copr(#0x3a,AC0,AC1) , dbl(*AR3+)=AC0
            AC0 = copr(#0x3b,AC1,AC0) , dbl(*AR3+)=AC1
            AC1 = copr(#0x29,AC0,*(AR2+T0),*(AR1+T0))
            AC0 = copr(#0x28,AC0,*(AR2+T0),*(AR1+T0))
            AC1 = copr(#0x2c,AC0,*(AR2–T1),*(AR1–T1))
            }
; load line 0, execute column 7, store column 6
            AC0 = copr(#0x2d,AC0,*(AR5+T0),*(AR4+T0))
            AC1 = copr(#0x2f,AC0,AC1), dbl(*AR3+)=AC0
            AC0 = copr(#0x2e,AC1,AC0), dbl(*AR3+)=AC1
            AC1 = copr(#0x3a,AC0,AC1), dbl(*AR3+)=AC0
            AC0 = copr(#0x3b,AC1,AC0), dbl(*AR3+)=AC1
            AC1 = copr(#0x29,AC0,*(AR5+T0),*(AR4+T0))
            BRC0 = #7
            AC0 = copr(#0x28,AC0,*(AR5+T0),*(AR4+T0))
            AC1 = copr(#0x2c,AC0,*(AR5–T1),*(AR4–T1))
; special IDCT mode
            AC1 = copr(#0x2b,AC0,AC1)
; load line 1, execute line 0, store column 7
            AC0 = copr(#0x2d,AC0,*(AR5+T0),*(AR4+T0))
            AC1 = copr(#0x2f,AC0,AC1), dbl(*AR3+)=AC0
            AC0 = copr(#0x2e,AC1,AC0), dbl(*AR3+)=AC1
            AC1 = copr(#0x3a,AC0,AC1), dbl(*AR3+)=AC0
            AC0 = copr(#0x3b,AC1,AC0), dbl(*AR3+)=AC1
            localrepeat {
; load line i+1, execute line i, store line i–1 (i>0) or store column 7 (i=0)
            AC1 = copr(#0x29,AC0,*(AR5+T0),*(AR4+T0))
            AC0 = copr(#0x28,AC0,*(AR5+T0),*(AR4+T0))
            AC1 = copr(#0x2a,AC0,*(AR5–T1),*(AR4–T1))
; load line i+2, execute line i+1, store line i
            AC0 = copr(#0x2d,AC0,*(AR5+T0),*(AR4+T0))
            AC1 = copr(#0x2f,AC0,AC1), dbl(*AR6+)=AC0
            AC0 = copr(#0x2e,AC1,AC0), dbl(*AR6+)=AC1
            AC1 = copr(#0x3a,AC0,AC1), dbl(*AR6+)=AC0
            AC0 = copr(#0x3b,AC1,AC0), dbl(*AR6+)=AC1
            }
      .endm
```

A.3 HWE_ME_1

```
_HWE_ME_1   .macro
            .noremark 5579
            BRC1 = #6
            AC0,AC1 = copr(#0x54,AC0,AC1,*AR0+,*AR1,coef(*CDP))
            AC0,AC1 = copr(#0x40,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
localrepeat {
            repeat(#0x4)                                    ; repeat 5 times
            AC0,AC1 = copr(#0x4e,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4e,AC0,AC1,*(AR0+T0),*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4e,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4e,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            repeat(#0x4)                                    ; repeat 5 times
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*(AR1+T0),coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            }
            repeat(#0x7)                                    ; repeat 8 times
            AC0,AC1 = copr(#0x4e,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x49,AC0,AC1,*AR0,*AR1+,coef(*CDP))
            AC0,AC1 = copr(#0x40,AC0,AC1,*AR0,*AR1,coef(*CDP))
      .remark 5579
      .endm
```

A.4 HWE_ME_2

```
_HWE_ME_2   .macro
            .noremark 5579
            BRC1 = #6
            AC0,AC1 = copr(#0x58,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
localrepeat {
            repeat(#0x5)                                ; repeat 6 times
            AC0,AC1 = copr(#0x4e,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4e,AC0,AC1,*(AR0+T0),*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4e,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            repeat(#0x5)                                ; repeat 6 times
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*(AR1+T0),coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            }
            repeat(#0x7)                                ; repeat 8 times
            AC0,AC1 = copr(#0x4e,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4d,AC0,AC1,*AR0,*AR1+,coef(*CDP))
            AC0,AC1 = copr(#0x49,AC0,AC1,*AR0,*AR1+,coef(*CDP))
            AC0,AC1 = copr(#0x40,AC0,AC1,*AR0,*AR1,coef(*CDP))
      .remark 5579
      .endm
```

A.5 HWE_ME_4

```
_HWE_ME_4   .macro
            .noremark 5579
;BRC0 = #6
            AC0,AC1 = copr(#0x5a,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
localrepeat {
            repeat(#0x5)                                    ; repeat 6 times
            AC0,AC1 = copr(#0x4e,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4e,AC0,AC1,*(AR0+T0),*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4e,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            repeat(#0x5)                                    ; repeat 6 times
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*(AR1+T0),coef(*CDP+))
            AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            }repeat(#0x7)                                   ; repeat 8 times
            AC0,AC1 = copr(#0x4e,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x4d,AC0,AC1,*AR0,*AR1+,coef(*CDP))
            AC0,AC1 = copr(#0x4d,AC0,AC1,*AR0,*AR1+,coef(*CDP))
            AC0,AC1 = copr(#0x49,AC0,AC1,*AR0,*AR1+,coef(*CDP))
            AC0,AC1 = copr(#0x49,AC0,AC1,*AR0,*AR1+,coef(*CDP))
            AC0,AC1 = copr(#0x40,AC0,AC1,*AR0,*AR1,coef(*CDP))
      .remark 5579
      .endm
```

A.6 HWE_ME_4MV_even

```
            BRC0 = #6    ;repeat 7 times
            AC0,AC1 = copr(#0x5c,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*(CDP+T0)))
localrepeat{
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*(CDP+T0)))
            }
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0,*AR1,coef(*CDP))
            AC0,AC1 = copr(#0x40,AC0,AC1,*AR0,*AR1,coef(*CDP))
            ; reset
      .endm
```

A.7 HWE_ME_4MV_odd

```
_HWE_ME_4MV_odd .macro
            BRC0 = #2    ;repeat 3 times
            AC0,AC1 = copr(#0x54,AC0,AC1,*AR0+,*AR1,coef(*CDP)) ;init
            AC0,AC1 = copr(#0x40,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1+,coef(*(CDP+T0)))
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0,*AR1+,coef(*CDP+))
            localrepeat{
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0+,*(AR1+T1),coef(*(CDP+T0)))
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1+,coef(*(CDP+T0)))
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0,*AR1+,coef(*CDP+))
            }
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0,*AR1+,coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0+,*(AR1+T1),coef(*CDP))
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x40,AC0,AC1,*AR0,*AR1,coef(*CDP))
      .endm
```

A.8 HWE_ME_8

```
_HWE_ME_8       .macro
                .noremark 5579
;BRC0 = #6                                             ; repeat 7 times
                AC0,AC1 = copr(#0x5c,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
                AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
                AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
                AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
                AC0,AC1 = copr(#0x43,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
                AC0,AC1 = copr(#0x47,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
                AC0,AC1 = copr(#0x47,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
                AC0,AC1 = copr(#0x47,AC0,AC1,*AR0+,*AR1,coef(*CDP+))
                AC0,AC1 = copr(#0x47,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
 localrepeat {
                repeat(#0x5)                           ; repeat 6 times
                AC0,AC1 = copr(#0x4e,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
                AC0,AC1 = copr(#0x4e,AC0,AC1,*(AR0+T0),*AR1+,coef(*CDP+))
                AC0,AC1 = copr(#0x4e,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
                repeat(#0x5)                           ; repeat 6 times
                AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
                AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*(AR1+T0),coef(*CDP+))
                AC0,AC1 = copr(#0x4f,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
                repeat(#0x7)                           ; repeat 8 times
                AC0,AC1 = copr(#0x4e,AC0,AC1,*AR0+,*AR1+,coef(*CDP+))
                repeat(#0x3)                           ; repeat 4 times
                AC0,AC1 = copr(#0x4d,AC0,AC1,*AR0,*AR1+,coef(*CDP))
                AC0,AC1 = copr(#0x49,AC0,AC1,*AR0,*AR1+,coef(*CDP))
                AC0,AC1 = copr(#0x49,AC0,AC1,*AR0,*AR1+,coef(*CDP))
                AC0,AC1 = copr(#0x49,AC0,AC1,*AR0,*AR1+,coef(*CDP))
                AC0,AC1 = copr(#0x49,AC0,AC1,*AR0,*AR1+,coef(*CDP))
                AC0,AC1 = copr(#0x40,AC0,AC1,*AR0,*AR1,coef(*CDP))
      .remark 5579
      .endm
```

A.9 HWE_ME_half_1

```
_HWE_ME_half_1   .macro
            T1 = #2
            BRC0 = #6
            AC0,AC1 = copr(#0x52,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            ;repeat(#0x6)
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
localrepeat {
            repeat(#0x4)                                    ; repeat 5 times
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T0),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            repeat(#0x4)                                    ; repeat 5 times
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*(AR1+T0),coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            }repeat(#0x7)                                    ; repeat 8 times
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x45,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP))
            AC0,AC1 = copr(#0x40,AC0,AC1,*AR0,*AR1,coef(*CDP))
      .endm
```

A.10 HWE_ME_half_2

```
_HWE_ME_half_2   .macro
            T1 = #2
            BRC0 = #6
            AC0,AC1 = copr(#0x52,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            repeat(#0x6)
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
localrepeat {
            repeat(#0x4)                                    ; repeat 5 times
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T0),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            repeat(#0x4)                                    ; repeat 5 times
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*(AR1+T0),coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
}
            repeat(#0x7)                                    ; repeat 8 times
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T1),*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x40,AC0,AC1,*AR0,*AR1,coef(*CDP))
      .endm
```

A.11 HWE_ME_half_3

```
HWE_ME_half_3   .macro
            AC0,AC1 = copr(#0x52,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AR2 = CDP
            AR2 = AR2 + DR2
            CDP = AR2
            AC0,AC1 = copr(#0x47,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
localrepeat {
            AC0,AC1 = copr(#0x46,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T0),*(AR1+T1),coef(*CDP+))
            AR2 = CDP
            AR2 = AR2 + DR2
            CDP = AR2
            AC0,AC1 = copr(#0x46,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x47,AC0,AC1,*(AR0+T1),*(AR1+T0),coef(*CDP+))
            AR2 = CDP
            AR2 = AR2 + DR2
            CDP = AR2
            AC0,AC1 = copr(#0x47,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
            }
            AC0,AC1 = copr(#0x46,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x46,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x45,AC0,AC1,*AR0,*AR1,coef(*CDP))
            AC0,AC1 = copr(#0x40,AC0,AC1,*AR0,*AR1,coef(*CDP))
      .endm
```

A.12 HWE_ME_half_4

```
_HWE_ME_half_4   .macro
            AC0,AC1 = copr(#0x52,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AR2 = CDP
            AR2 = AR2 + DR2
            CDP = AR2
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
            localrepeat {
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T0),*(AR1+T1),coef(*CDP+))
            AR2 = CDP
            AR2 = AR2 + DR2
            CDP = AR2
            AC0,AC1 = copr(#0x42,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*AR1,coef(*CDP+))
            AC0,AC1 = copr(#0x43,AC0,AC1,*(AR0+T1),*(AR1+T0),coef(*CDP+))
            AR2 = CDP
            AR2 = AR2 + DR2
            CDP = AR2
            AC0,AC1 = copr(#0x43,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
            }
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP+))
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0,*(AR1+T1),coef(*CDP))
            AC0,AC1 = copr(#0x42,AC0,AC1,*AR0,*AR1,coef(*CDP))
            AC0,AC1 = copr(#0x40,AC0,AC1,*AR0,*AR1,coef(*CDP))
      .endm
```

A.13 HWE_PI_16x16_0

```
_HWE_PI_16x16_0   .macro
            BRC0 = #15                                      ; repeat 16 times
            AC1 = copr (#0x0 , AC0, dbl(*AR2+));
            AC1 = copr (#0x10, AC0, dbl(*AR3+));
            AC1 = copr (#0x11, AC0, AC1);
            AC1 = copr (#0x12, AC0, AC1);
            AC1 = copr (#0x13, AC0, dbl(*AR2+));
            AC1 = copr (#0x10, AC0, dbl(*AR3+));
            AC1 = copr (#0x11, AC0, AC1);
            AC1 = copr (#0x12, AC0, AC1);
            AC1 = copr (#0x13, AC0, dbl(*AR2+));
            AC1 = copr (#0x10, AC0, dbl(*AR3+));
            AC1 = copr (#0x11, AC0, AC1);
            AC1 = copr (#0x12, AC0, AC1),dbl(*AR0+)=AC1;
            AC1 = copr (#0x13, AC0, dbl(*AR2+)),dbl(*AR1+)=AC1;
            AC1 = copr (#0x10, AC0, dbl(*AR3+)),dbl(*AR0+)=AC1;
            AC1 = copr (#0x11, AC0, AC1),dbl(*AR1+)=AC1;
            AC1 = copr (#0x12, AC0, AC1),dbl(*AR0+)=AC1;
            AC1 = copr (#0x13, AC0, dbl(*(AR2+T0))),dbl(*AR1+)=AC1;

blockrepeat {
            AC1 = copr (#0x10, AC0, dbl(*(AR3+T0))),dbl(*AR0+)=AC1;
            AC1 = copr (#0x11, AC0, dbl(*AR2+)),dbl(*AR1+)=AC1;
            AC1 = copr (#0x10, AC0, dbl(*AR3+)),dbl(*AR0+)=AC1;
            AC1 = copr (#0x11, AC0, AC1),dbl(*AR1+)=AC1;
            AC1 = copr (#0x12, AC0, AC1),dbl(*AR0+)=AC1;
            AC1 = copr (#0x13, AC0, dbl(*AR2+)),dbl(*AR1+)=AC1;
            AC1 = copr (#0x10, AC0, dbl(*AR3+)),dbl(*AR0+)=AC1;
            AC1 = copr (#0x11, AC0, AC1),dbl(*AR1+)=AC1;
            AC1 = copr (#0x12, AC0, AC1),dbl(*AR0+)=AC1;
            AC1 = copr (#0x13, AC0, dbl(*AR2+)),dbl(*AR1+)=AC1;
            AC1 = copr (#0x10, AC0, dbl(*AR3+)),dbl(*(AR0+T1))=AC1;
            AC1 = copr (#0x11, AC0, AC1),dbl(*(AR1+T1))=AC1;
            AC1 = copr (#0x12, AC0, AC1),dbl(*AR0+)=AC1;
```

```
        AC1 = copr (#0x13, AC0, dbl(*AR2+)),dbl(*AR1+)=AC1;
        AC1 = copr (#0x10, AC0, dbl(*AR3+)),dbl(*AR0+)=AC1;
        AC1 = copr (#0x11, AC0, AC1),dbl(*AR1+)=AC1;
        AC1 = copr (#0x12, AC0, AC1),dbl(*AR0+)=AC1;
        AC1 = copr (#0x13, AC0, dbl(*(AR2+T0))),dbl(*AR1+)=AC1;
        }
        AC1 = copr (#0x10, AC0, dbl(*(AR3+T0))),dbl(*AR0+)=AC1;
        AC1 = copr (#0x11, AC0, AC1),dbl(*AR1+)=AC1
        AC1 = copr (#0x10, AC0, AC1),dbl(*AR0+)=AC1;
        AC1 = copr (#0x11, AC0, AC1),dbl(*AR1+)=AC1
        AC1 = copr (#0x12, AC0, AC1),dbl(*AR0+)=AC1;
        AC1 = copr (#0x13, AC0, AC1),dbl(*AR1+)=AC1
        AC1 = copr (#0x10, AC0, AC1),dbl(*AR0+)=AC1;
        AC1 = copr (#0x11, AC0, AC1),dbl(*AR1+)=AC1
        AC1 = copr (#0x12, AC0, AC1),dbl(*AR0+)=AC1;
        AC1 = copr (#0x13, AC0, AC1),dbl(*AR1+)=AC1
        AC1 = copr (#0x10, AC0, AC1),dbl(*AR0+)=AC1;
        AC1 = copr (#0x11, AC0, AC1),dbl(*AR1+)=AC1
.endm
```

A.14 HWE_PI_16x16_1

```
_HWE_PI_16x16_1 .macro
             AC1 = copr (#0x00 , AC0, dbl(*AR2+));
             AC1= copr (#0x16, AC0, dbl(*AR3+));
             AC1 = copr (#0x17, AC0, AC1);
             AC1 = copr (#0x12, AC0, dbl(*AR2+));
             AC1 = copr (#0x14, AC0, dbl(*AR3+));
             AC1 = copr (#0x15, AC0, AC1);
             AC1 = copr (#0x13, AC0, AC1);
             AC1 = copr (#0x12, AC0, dbl(*AR2+));
             AC1 = copr (#0x14, AC0, dbl(*AR3+));
             AC1 = copr (#0x15, AC0, AC1);
             AC1 = copr (#0x13, AC0, AC1)
           BRC0 = #15     ; repeat 16 times
blockrepeat {
             AC1 = copr (#0x12, AC0, dbl(*AR2+)), dbl(*AR0+)=AC1;
             AC1 = copr (#0x14, AC0, dbl(*AR3+)), dbl(*AR1+)=AC1;
             AC1 = copr (#0x15, AC0, AC1), dbl(*AR0+)=AC1;
             AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
             AC1 = copr (#0x12, AC0, dbl(*(AR2+T0))), dbl(*AR0+)=AC1;
             AC1 = copr (#0x14, AC0, dbl(*(AR3+T0))), dbl(*AR1+)=AC1;
             AC1 = copr (#0x15, AC0, AC1), dbl(*AR0+)=AC1;
             AC1 = copr (#0x13, AC0, dbl(*AR2+)), dbl(*AR1+)=AC1;
             AC1 = copr (#0x16, AC0, dbl(*AR3+)), dbl(*AR0+)=AC1;
             AC1 = copr (#0x17, AC0, AC1), dbl(*AR1+)=AC1;
             AC1 = copr (#0x12, AC0, dbl(*AR2+)), dbl(*AR0+)=AC1;
             AC1 = copr (#0x14, AC0, dbl(*AR3+)), dbl(*AR1+)=AC1;
             AC1 = copr (#0x15, AC0, AC1), dbl(*AR0+)=AC1;
             AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
             AC1 = copr (#0x12, AC0, dbl(*AR2+)), dbl(*AR0+)=AC1;
             AC1 = copr (#0x14, AC0, dbl(*AR3+)), dbl(*AR1+)=AC1;
             AC1 = copr (#0x15, AC0, AC1), dbl(*(AR0+T1))=AC1;
             AC1 = copr (#0x13, AC0, AC1), dbl(*(AR1+T1))=AC1;
             }
             AC1 = copr (#0x12, AC0, dbl(*AR2+)), dbl(*AR0+)=AC1;
```

```
        AC1 = copr (#0x14, AC0, dbl(*AR3+)), dbl(*AR1+)=AC1;
        AC1 = copr (#0x15, AC0, AC1), dbl(*AR0+)=AC1;
        AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
        AC1 = copr (#0x12, AC0, dbl(*AR2+)), dbl(*AR0+)=AC1;
        AC1 = copr (#0x14, AC0, dbl(*AR3+)), dbl(*AR1+)=AC1;
        AC1 = copr (#0x15, AC0, AC1), dbl(*AR0+)=AC1;
        AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
        AC1 = copr (#0x16, AC0, AC1), dbl(*AR0+)=AC1;
        AC1 = copr (#0x17, AC0, AC1), dbl(*AR1+)=AC1;
        AC1 = copr (#0x12, AC0, AC1), dbl(*AR0+)=AC1;
        AC1 = copr (#0x14, AC0, AC1), dbl(*AR1+)=AC1;
        AC1 = copr (#0x15, AC0, AC1), dbl(*AR0+)=AC1;
        AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
        AC1 = copr (#0x12, AC0, AC1), dbl(*AR0+)=AC1;
        AC1 = copr (#0x14, AC0, AC1), dbl(*AR1+)=AC1;
        AC1 = copr (#0x15, AC0, AC1), dbl(*AR0+)=AC1;
        AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
.endm
```

A.15 HWE_PI_16x16_2

```
_HWE_PI_16x16_2 .macro
               AC1 = copr (#0x0 , AC0, dbl(*AR2+));
               AC1 = copr (#0x18, AC0, dbl(*AR3+));
               AC1 = copr (#0x19, AC0, dbl(*AR2+));
               AC1 = copr (#0x10, AC0, dbl(*AR3+));
               AC1 = copr (#0x11, AC0, AC1);
               AC1 = copr (#0x12, AC0, AC1);
               AC1 = copr (#0x13, AC0, dbl(*AR2+));
               AC1 = copr (#0x10, AC0, dbl(*AR3+));
               AC1 = copr (#0x11, AC0, AC1);
               AC1 = copr (#0x12, AC0, AC1);
               AC1 = copr (#0x13, AC0, dbl(*AR2+))
               BRC0 = #15                                     ; repeat 16 times
            blockrepeat {
               AC1 = copr (#0x10, AC0, dbl(*AR3+)), dbl(*AR0+)=AC1;
               AC1 = copr (#0x11, AC0, AC1), dbl(*AR1+)=AC1;
               AC1 = copr (#0x12, AC0, AC1), dbl(*AR0+)=AC1;
               AC1 = copr (#0x13, AC0, dbl(*(AR2+T0))), dbl(*AR1+)=AC1;
               AC1 = copr (#0x10, AC0, dbl(*(AR3+T0))), dbl(*AR0+)=AC1;
               AC1 = copr (#0x11, AC0, AC1), dbl(*AR1+)=AC1;
               AC1 = copr (#0x12, AC0, AC1), dbl(*AR0+)=AC1;
               AC1 = copr (#0x13, AC0, dbl(*AR2+)), dbl(*AR1+)=AC1;
               AC1 = copr (#0x18, AC0, dbl(*AR3+)), dbl(*AR0+)=AC1;
               AC1 = copr (#0x19, AC0, dbl(*AR2+)), dbl(*AR1+)=AC1;
               AC1 = copr (#0x10, AC0, dbl(*AR3+)), dbl(*AR0+)=AC1;
               AC1 = copr (#0x11, AC0, AC1), dbl(*AR1+)=AC1;
               AC1 = copr (#0x12, AC0, AC1), dbl(*AR0+)=AC1;
               AC1 = copr (#0x13, AC0, dbl(*AR2+)), dbl(*AR1+)=AC1;
               AC1 = copr (#0x10, AC0, dbl(*AR3+)), dbl(*AR0+)=AC1;
               AC1 = copr (#0x11, AC0, AC1), dbl(*AR1+)=AC1;
               AC1 = copr (#0x12, AC0, AC1), dbl(*(AR0+T1))=AC1;
               AC1 = copr (#0x13, AC0, dbl(*AR2+)), dbl(*(AR1+T1))=AC1;
                   }
               AC1 = copr (#0x10, AC0, dbl(*AR3+)), dbl(*AR0+)=AC1;
```

```
          AC1 = copr (#0x11, AC0, AC1), dbl(*AR1+)=AC1;
          AC1 = copr (#0x12, AC0, AC1), dbl(*AR0+)=AC1;
          AC1 = copr (#0x13, AC0, dbl(*AR2+)), dbl(*AR1+)=AC1;
          AC1 = copr (#0x10, AC0, dbl(*AR3+)), dbl(*AR0+)=AC1;
          AC1 = copr (#0x11, AC0, AC1), dbl(*AR1+)=AC1;
          AC1 = copr (#0x12, AC0, AC1), dbl(*AR0+)=AC1;
          AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
          AC1 = copr (#0x18, AC0, AC1), dbl(*AR0+)=AC1;
          AC1 = copr (#0x19, AC0, AC1), dbl(*AR1+)=AC1;
          AC1 = copr (#0x10, AC0, AC1), dbl(*AR0+)=AC1;
          AC1 = copr (#0x11, AC0, AC1), dbl(*AR1+)=AC1;
          AC1 = copr (#0x12, AC0, AC1), dbl(*AR0+)=AC1;
          AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
          AC1 = copr (#0x10, AC0, AC1), dbl(*AR0+)=AC1;
          AC1 = copr (#0x11, AC0, AC1), dbl(*AR1+)=AC1;
          AC1 = copr (#0x12, AC0, AC1), dbl(*AR0+)=AC1;
          AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
.endm
```

A.16 HWE_PI_16x16_3

```
_HWE_PI_16x16_3 .macro
            AC1 = copr (#0x0 , AC0, dbl(*AR2+))
            AC1 = copr (#0x1a, AC0, dbl(*AR3+))
            AC1 = copr (#0x1b, AC0, dbl(*AR2+))
            AC1 = copr (#0x14, AC0, dbl(*AR3+))
            AC1 = copr (#0x15, AC0, AC1)
            AC1 = copr (#0x13, AC0, AC1)
            AC1 = copr (#0x12, AC0, dbl(*AR2+))
            AC1 = copr (#0x14, AC0, dbl(*AR3+))
            AC1 = copr (#0x15, AC0, AC1)
            AC1 = copr (#0x13, AC0, AC1)
            AC1 = copr (#0x12, AC0, dbl(*AR2+))
            AC1 = copr (#0x14, AC0, dbl(*AR3+))
            BRC0 = #15                                      ; repeat 16 times
            blockrepeat {
            AC1 = copr (#0x15, AC0, AC1), dbl(*AR0+)=AC1;
            AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
            AC1 = copr (#0x12, AC0, dbl(*AR2+)), dbl(*AR0+)=AC1;
            AC1 = copr (#0x14, AC0, dbl(*AR3+)), dbl(*AR1+)=AC1;
            AC1 = copr (#0x15, AC0, dbl(*(AR2+T0))), dbl(*AR0+)=AC1;
            AC1 = copr (#0x1c, AC0, dbl(*(AR3+T0))), dbl(*AR1+)=AC1;
            AC1 = copr (#0x1d, AC0, dbl(*AR2+)), dbl(*AR0+)=AC1;
            AC1 = copr (#0x1a, AC0, dbl(*AR3+)), dbl(*AR1+)=AC1;
            AC1 = copr (#0x1b, AC0, dbl(*AR2+)), dbl(*AR0+)=AC1;
            AC1 = copr (#0x14, AC0, dbl(*AR3+)), dbl(*AR1+)=AC1;
            AC1 = copr (#0x15, AC0, AC1), dbl(*AR0+)=AC1;
            AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
            AC1 = copr (#0x12, AC0, dbl(*AR2+)), dbl(*AR0+)=AC1;
            AC1 = copr (#0x14, AC0, dbl(*AR3+)), dbl(*AR1+)=AC1;
            AC1 = copr (#0x15, AC0, AC1), dbl(*AR0+)=AC1;
            AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
            AC1 = copr (#0x12, AC0, dbl(*AR2+)), dbl(*(AR0+T1))=AC1;
            AC1 = copr (#0x14, AC0, dbl(*AR3+)), dbl(*(AR1+T1))=AC1;
            }
```

```
      AC1 = copr (#0x15, AC0, AC1), dbl(*AR0+)=AC1;
      AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
      AC1 = copr (#0x12, AC0, dbl(*AR2+)), dbl(*AR0+)=AC1;
      AC1 = copr (#0x14, AC0, dbl(*AR3+)), dbl(*AR1+)=AC1;
      AC1 = copr (#0x15, AC0, dbl(*AR2+)), dbl(*AR0+)=AC1;
      AC1 = copr (#0x1c, AC0, dbl(*AR3+)), dbl(*AR1+)=AC1;
      AC1 = copr (#0x1d, AC0, dbl(*AR2+)), dbl(*AR0+)=AC1;
      AC1 = copr (#0x1a, AC0, dbl(*AR3+)), dbl(*AR1+)=AC1;
      AC1 = copr (#0x1b, AC0, AC1), dbl(*AR0+)=AC1;
      AC1 = copr (#0x14, AC0, AC1), dbl(*AR1+)=AC1;
      AC1 = copr (#0x15, AC0, AC1), dbl(*AR0+)=AC1;
      AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
      AC1 = copr (#0x12, AC0, AC1), dbl(*AR0+)=AC1;
      AC1 = copr (#0x14, AC0, AC1), dbl(*AR1+)=AC1;
      AC1 = copr (#0x15, AC0, AC1), dbl(*AR0+)=AC1;
      AC1 = copr (#0x13, AC0, AC1), dbl(*AR1+)=AC1;
      AC1 = copr (#0x12, AC0, AC1), dbl(*AR0+)=AC1;
      AC1 = copr (#0x14, AC0, AC1), dbl(*AR1+)=AC1;
.endm
```

Index