Chapter 2

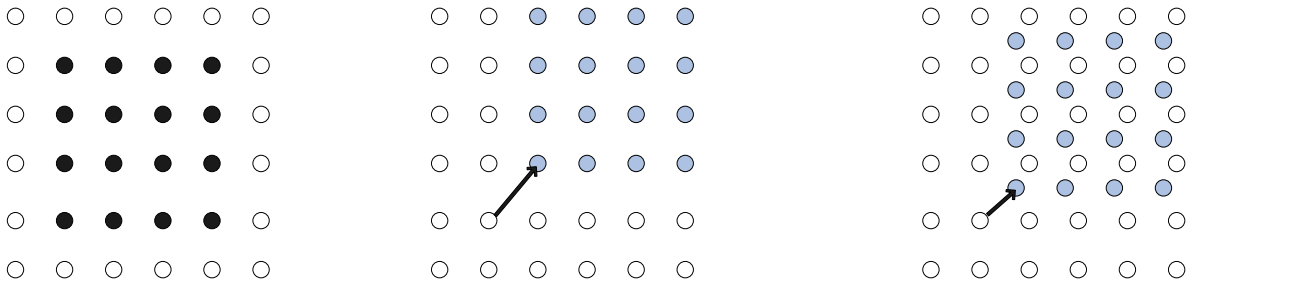
Interpolation Algorithm and New Instructions

2.1 The Interpolation Procedure

The interpolation procedure is used to predict each block from a reference picture. Since the luma and chroma samples at sub-sample positions do not exist in the reference picture, interpolation from nearby coded samples is used to compute that.

2.1.1 Motion Vectors

Interpolation operation is performed based on a given motion vector. Before we review the interpolation operation, we give the definition of motion vector. Movement of a reference block is described by a motion vector. The motion vector of a partition in the current frame is predicted from an area of the same size in the reference frame. Fig. 2.1 shows an example of motion vector. Fig. 2.1(a) is a 4×4 block in current frame, and Fig. 2.1(b), Fig. 2.1(c) give two different motion vectors based on two different reference frames. If the horizontal and vertical components of the motion vector are integers (Fig. 2.1(b)), the referenced samples in the reference block actually exists (grey dots). Thus, prediction samples are readily computed. However, if one or both vector components are fractional values (Fig. 2.1(c)), the prediction samples are generated by interpolation between adjacent samples in the reference frame (white dots).



(a) 4x4 block in current frame (b) 4x4 reference block: vector (1, -1) (c) 4x4 reference block: vector (0.75, -0.5)

Figure 2.1: Movement of a block depending on different motion vectors.

2.1.2 Generation of Interpolated Samples

The half-way samples between integer-position samples are generated first. Each half-pixel sample that is adjacent to two integer samples is interpolated with weights $(1/32, -5/32, 5/8, 5/8, -5/32, 1/32)$. For example, in Fig. 2.2, gray squares with upper letter denote integer sample position while the white ones with lower letter denote fractional sample position. Half-pixel sample d is calculated from the six horizontal integer samples $\{E, F, G, H, I, J\}$ with the following formula:

$$d = round((E - 5F + 20G + 20H - 5I + J)/32) \tag{2.1}$$

Similarly, m is interpolated by sampling $\{A, C, G, M, R, T\}$.

Once all of the samples horizontally and vertically adjacent to integer samples have been calculated, the remaining half-pixel positions are calculated by interpolating six horizontal or vertical half-pixel samples. For example, o is generated by filtering $\{j, k, m, q, r, s\}$ (note that the result is the same whether o is interpolated horizontally or vertically; note also that un-rounded versions of m and q are used to generate o). The six-tap interpolation filter is relatively complex but produces an accurate fit to the integer-sample data and hence good motion compensation performance.

Once all the half-pixel samples are available, the samples at quarter-step (quarter-pixel) positions are produced by linear interpolation (Fig. 2.3). Quarter-pixel positions with two horizontally or vertically adjacent half- or integer-position samples (e.g. $\{a, c, i, k\}$ and

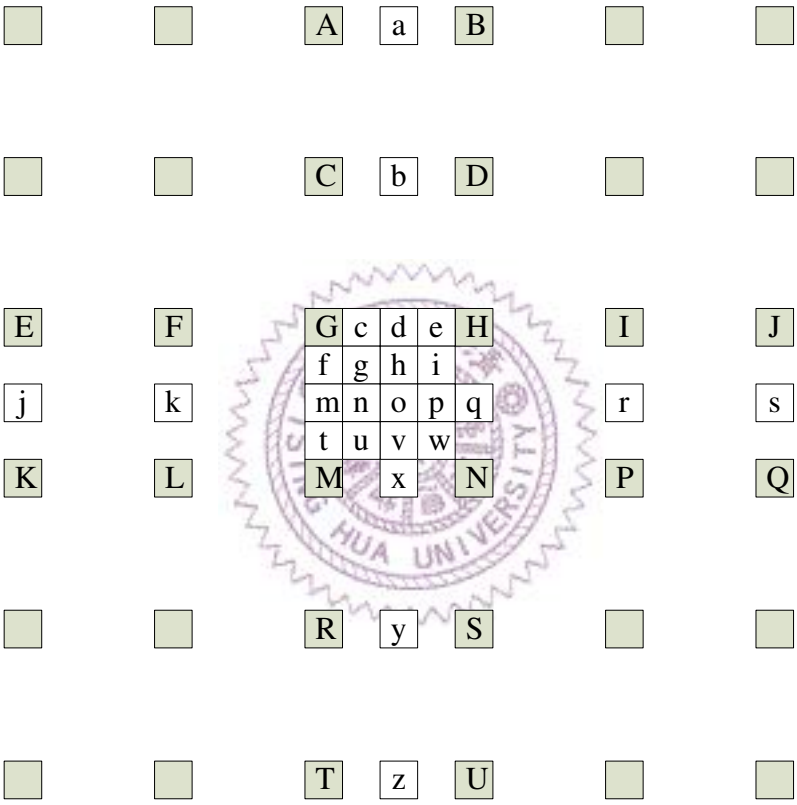


Figure 2.2: Pixel locations in the sub-pixel interpolation scheme

$\{d, f, n, q\}$ in lower part of Fig. 2.3) are linearly interpolated using these adjacent samples, for example:

$$a = round((G + b)/2) \tag{2.2}$$

The remaining quarter-pixel positions ($\{e, g, p, r\}$ in lower part of Fig. 2.3) are linearly interpolated using a pair of diagonally opposite half-pixel samples. For example, g is interpolated between b and m.

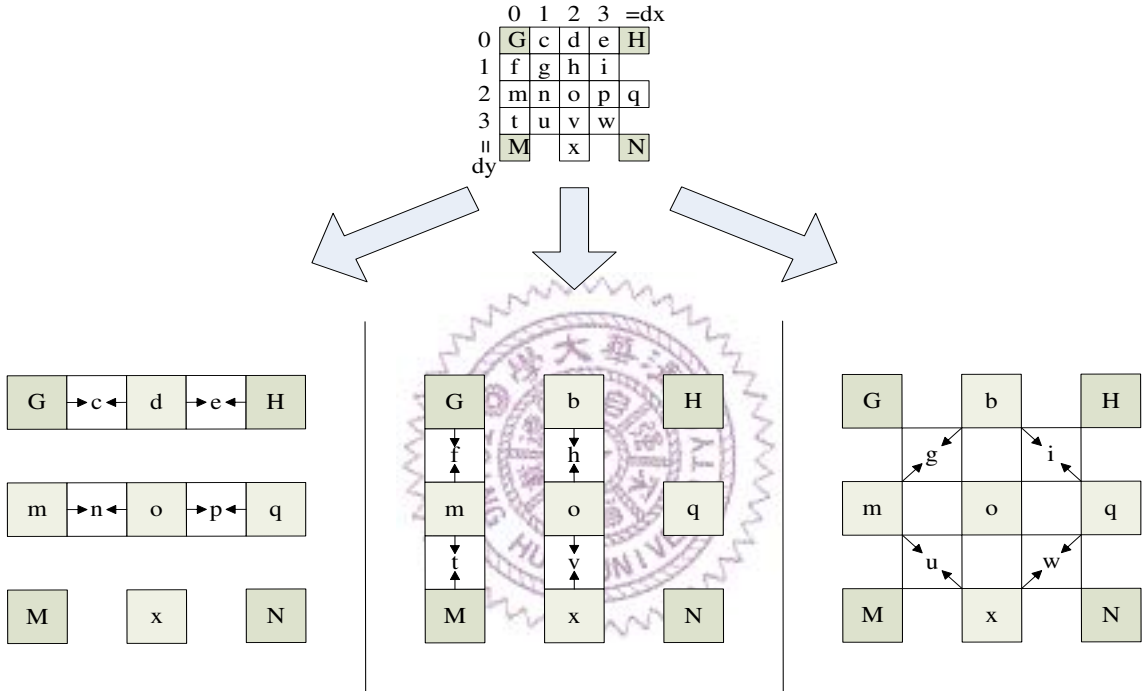


Figure 2.3: Interpolation of quarter-pixel positions

2.1.3 Algorithm Flow

The interpolation procedure consists of six cases: Full-pixel position, Vertical interpolation (no-horizontal interpolation, **NH**), Horizontal interpolation (no-vertical interpolation, **NV**), Vertical after Horizontal interpolation (**VH**), Horizontal after Vertical interpolation (**HV**), and Diagonal interpolation(**D**). Since each case performs similar operations, we could improve the computation time using dedicated instructions with little overhead. Upper part of Fig. 2.3 shows the positions of pixels which are used to predict sample pix-

els according to the motion vector. $\{G,\ H,\ M,\ N\}$ are current pixels while others are positions we want to predict. According to Fig. 2.2, we show all the formula to calculate $\{c, d, e, f, g, h, i, m, n, o, p, q, t, u, v, w, x\}$ as follows.

$$d = (E - 5F + 20G + 20H - 5I + J + 16)/32 \tag{2.3}$$

$$x = (K - 5L + 20M + 20N - 5P + Q + 16)/32 \tag{2.4}$$

$$m = (A - 5C + 20G + 20M - 5R + T + 16)/32 \tag{2.5}$$

$$q = (B - 5D + 20H + 20N - 5S + U + 16)/32 \tag{2.6}$$

$$\begin{cases} o = (a - 5b + 20d + 20x - 5y + z + 512)/1024 \\ \text{or } o = (j - 5k + 20m + 20q - 5r + s + 512)/1024 \end{cases} \tag{2.7}$$

$$c = (G + d + 1) >> 1 \tag{2.8}$$

$$e = (d + H + 1) >> 1 \tag{2.9}$$

$$f = (G + m + 1) >> 1 \tag{2.10}$$

$$g = (d + m + 1) >> 1 \tag{2.11}$$

$$i = (d + q + 1) >> 1 \tag{2.12}$$

$$n = (m + o + 1) >> 1 \tag{2.13}$$

$$p = (o + q + 1) >> 1 \tag{2.14}$$

$$t = (m + M + 1) >> 1 \tag{2.15}$$

$$u = (m + x + 1) >> 1 \tag{2.16}$$

$$v = (o + x + 1) >> 1 \tag{2.17}$$

$$w = (x + q + 1) >> 1 \tag{2.18}$$

Upper part of Fig. 2.3 also shows the value of dx and dy that represent the motion vector. By the equations above and Fig. 2.4, we can easily figure out that:$\{c, d, e, x\}$ are interpolated by NV procedure, $\{f, m, t, q\}$ are interpolated by NH procedure, $\{h, o, v\}$ are interpolated by VH procedure, $\{n, p\}$ are interpolated by HV procedure, and $\{g, i, u, w\}$ are interpolated by D procedure.

In our experiment, we analyze JM code written in C in detail. We found that the interpolation procedure contains a loop of three levels in each case. Fig. 2.5 shows the structure of the loop. Each time the interpolation procedure runs one case which is selected according to the motion vector. The first *for-loop* chooses a raw/column of the block, and the second decides which element in the block will be interpolated. By equation (2.1), we know that the inner *for-loop* requires six multiplication to produce the result of current pixel. Each computation consist of three steps: *Checking Boundaries*, *Load Pixel*, and *Multiply_Addition* as shown in Fig 2.5.

Load Pixel and the *Multiply_Addition* steps perform loading the value and calculating operations, respectively. *Checking Boundaries* is to check if the referenced pixel is outside the frame. Without this step, the procedure may load the wrong value and results in unpredictable fault.

2.2 New Instructions

From the analysis, we discovere that *Checking Boundaries* is redundant during most of the executions. The new instructions we propose are to reduce these checking actions.

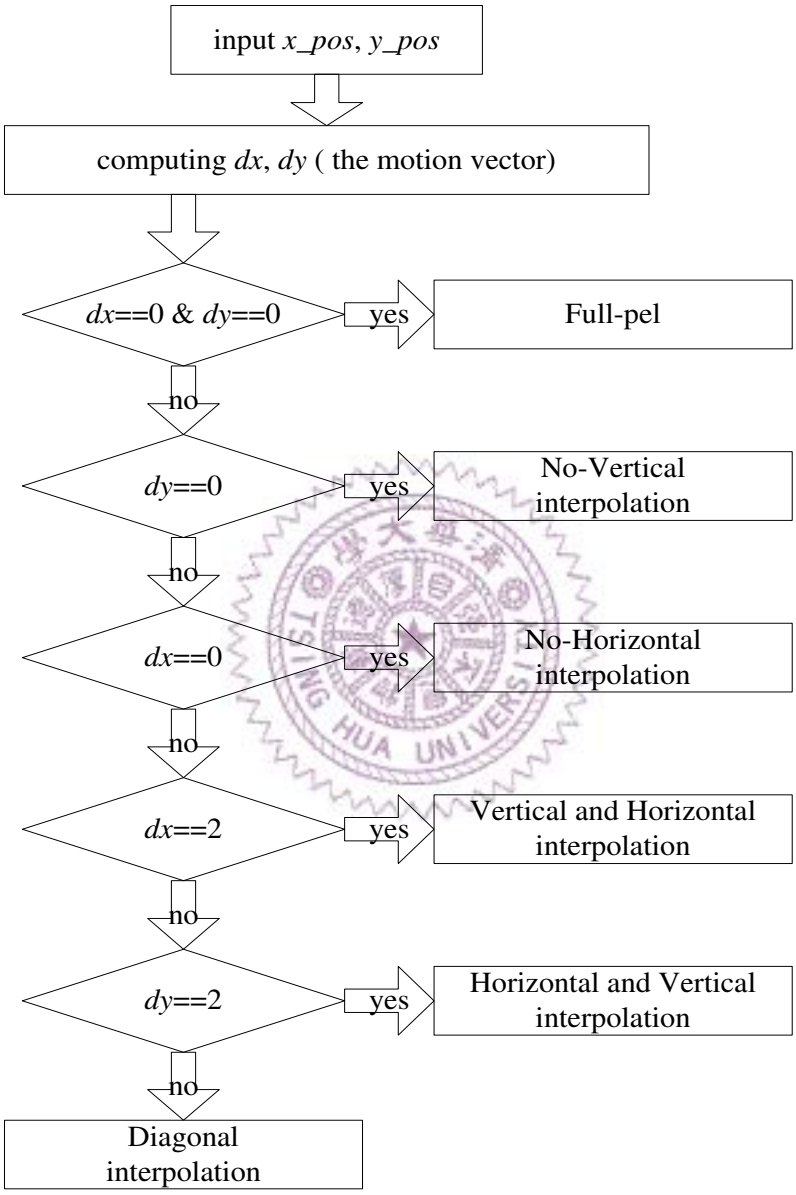


Figure 2.4: The flow chart of the algorithm of the interpolation procedure

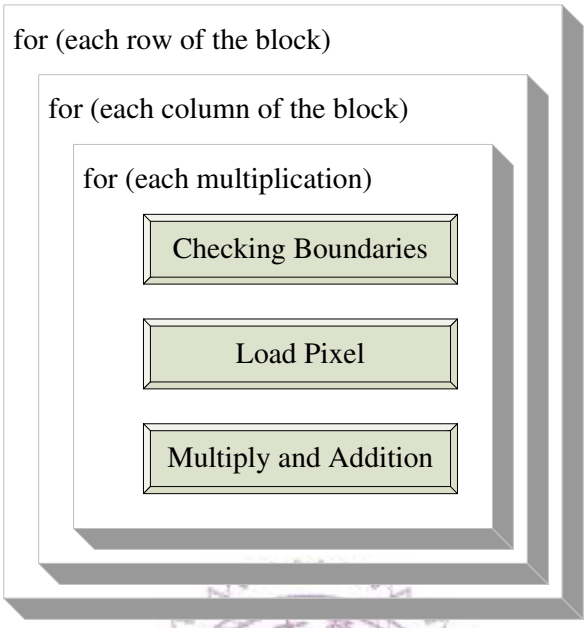


Figure 2.5: Computation structure of the three level loop

LOOP(six times) :

```
R_to_load = MIN ( R_UpperBound, R_ref_pel )  || NOP  || R_coef = [ I2 ++ ]
R_to_load = MAX ( R_Lowerbound , R_to_load )
P_to_load = R_to_load                                   Checking Boundaries

P_to_load = P_base + P_to_load
R_tmpresult = B [ P_to_load ] ( Z )                     Load Pixel

R_tmpresult *= R_coef
R_result = R_result + R_tmpresult
R_ref_pel = R_ref_pel + R3 // R3 = 1                    Multiply_Addition
```

Figure 2.6: Code sequence generated by VisualDSP4.0

2.2.1 CHKB instruction

Current DSP instructions include MIN/MAX instruction. But it still requires many instructions to accomplish the checking step. In interpolation procedure, to perform checking , the procedure extracts the minimum between the value loaded and the upper bound of the referenced block. Then, it extracts the maximum between the result and the lower bound (always zero). Current ISA can use the MIN/MAX instructions to achieve this goal. After that, we move the result into pointer-register for the use of load. It takes at least three instructions to accomplish the checking step and takes about 35% of instructions in the inner *for-loop*.

Since our goal is to decrease the number of instructions in the inner *for-loop*, we first focus on reducing these instructions. The regulation of *Checking-Boundaries* give us clue to design $CHKB$ instruction. Here is the syntax of $CHKB$ instruction:

$$Preg0 = CHKB(Preg1, Preg2) \tag{2.19}$$

Main functionality of $CHKB$ instruction is simple. The semantic of $CHKB$ instruction is to first take the minimum, *minp*, of $Preg1$ and $Preg2$, then take the maximum of *minp* and zero, and finally store the result to $Preg0$. The original code sequences shown in Fig 2.6 is changed to codes with the $CHKB$ as shown in Fig. 2.7. Furthermore, this instruction is designed to perform on DAG (Data Address Generator) so that it will not need redundant move instructions.

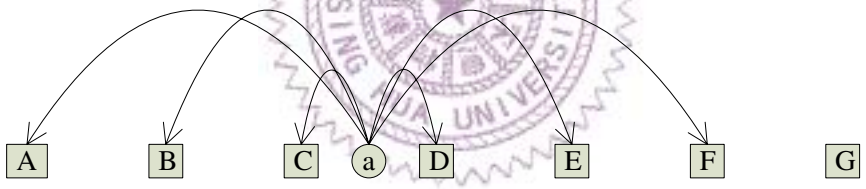
```
P_to_load = CHKB ( P_UpperBound, P_ref_pel )
ACC = 0
P_to_load = P_base + P_to_load
MNOP || R_coef = [ I2 ++ ] || R_tmpresult = B [ P_to_load ] (Z)
LOOP: ( five times )
      P_ref_pel = P_ref_pel + P3       // next pixel to reference ( P3 = 1 )
      P_to_load = CHKB ( P_UpperBound, P_ref_pel )
      ACC += ( R_coef * R_tmpresult ) (IS) || R_coef = [ I2 ++ ] || R_tmpresult = B [ P_to_load ] (Z)
ACC += R_coef.L * R_tmpresult.L (IS)
R_result = ACC
```

Figure 2.7: The code sequence applying the CHKB instruction

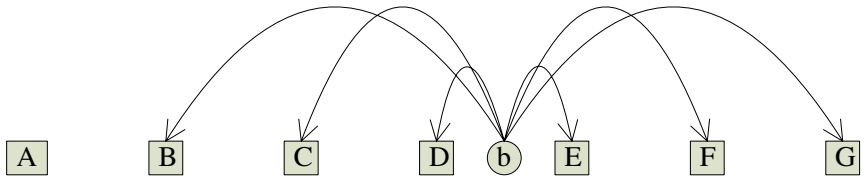
The original code sequences will take 46 instructions. With $CHKB$ instruction, it only take 22 instructions to complete each calculation.

2.2.2 BFLOAD instruction and LDMVBF instruction

Now we focus on the *Load* action. In a more detailed analysis, we checked the pixels required in each calculation. As shown in Fig. 2.8, the grey squares represent the pixel referenced and the white ones represent the predicted position of horizontal interpolation. Each horizontal interpolation will require six loads to get the reference pixels. In Fig. 2.8(a), performing horizontal interpolation on a needs to load A, B, C, D, E, and F one at a time. Then, in the next iteration, interpolation on b requires loading B, C, D, E, F, and G as shown in Fig. 2.8(b). Apparently, where five loads are redundant which are B, C, D, E, and F. Fig. 2.9 shows a similar case where vertical interpolation is performed. Unfortunately, the number of registers is not enough to store those values temporally in CPU. Therefore, new instructions and a new buffer are developed to achieve this goal.



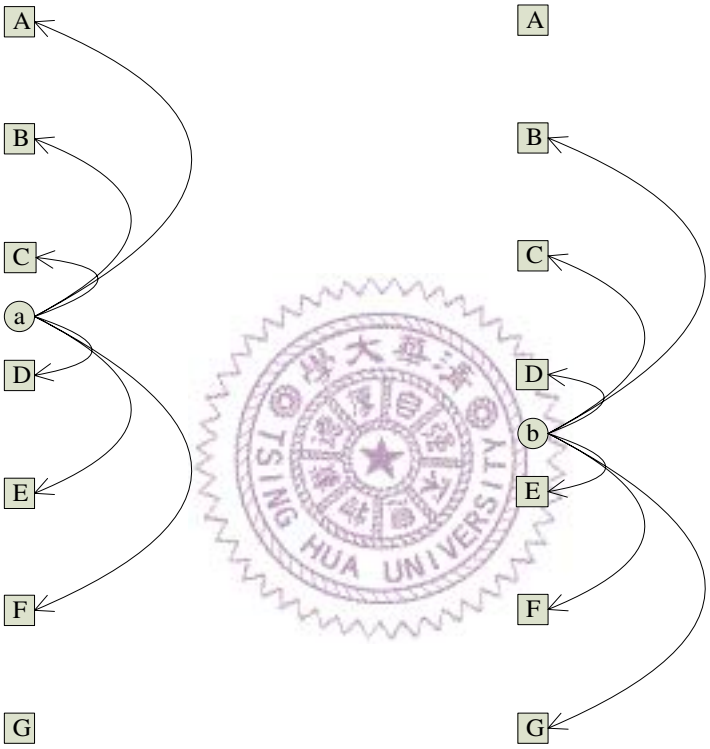
(a) Step1: No-Vertical interpolating on position *a*



(b) Step2: No-Vertical interpolating on position *b*

Figure 2.8: Pixels referenced to perform horizontal interpolation on position a and b

The idea is to design a buffer just for interpolation. Tracing the assembly code generated by VisualDSP4.0 of JM, we discover that each load of pixel only loads a byte then zero-extension is performed. This means that each entry of buffer have one byte in size. There



(a) Step1:
No-Horizontal interpolating on position a

(b) Step2:
No-Horizontal interpolating on position b

Figure 2.9: Pixels referenced to perform vertical interpolation on position a and b

are five loads that overlap. Therefore, the number of entries of buffer is five. By the above observation, we could design new instructions whose syntax are:

$$Dreg = LDMVBF[Preg] \tag{2.20}$$

$$Dreg = BFLOAD \tag{2.21}$$

The $LDMVBF$ instruction performs operations similar to $Dreg = B[Preg](Z)$, which loads byte of the data pointed by $Preg$, zero-extends the value, and finally stores it into destination Dreg. The only different part between $LDMVBF$ and Load-Byte-Zero-Extension is that the lower order byte of the loaded result is copied into the bottom of the buffer by $LDMVBF$. The buffer acts like a circular array. While a new data is to stored, the oldest data will be overwritten. That is, assuming we have a buffer named BF and indexed 0-4. $LDMVBF$ will cause the effect of moving the data of {BF[1], BF[2], BF[3], BF[4]} into {BF[0], BF[1], BF[2], BF[3]}, respectively. Meanwhile lower order byte of the data loaded by $LDMVBF$ will be stored into {BF[4]}.

The $BFLOAD$ instruction performs a simpler operation. It loads the top value of the buffer into the destination Dreg and zero-extends. When the load is done, this instruction rotates the buffer by eight bits. Let {BF[0], BF[1], BF[2], BF[3], BF[4]} store the data of $\{a, b, c, d, e\}$ respectively, the new buffer will become {b, c, d, e, a} after $BFLOAD$ is performed. The reason to design the new instruction is because in interpolation procedure, a data loaded from memory, it can remain in buffer for five operations. To make the hardware implementation more efficient, the new buffer is not required to be byte-accessible.

These special instructions are used dedicately. Therefore, a macro is written to utilize these instructions. While writing these parts of code, we have to break the inner loop to make these instructions work efficiently. Fig 2.10 shows the original assembly code, and Fig. 2.11 shows the code with $BFLOAD$ and $LDMVBF$ instructions. The original code will take about 1168 instructions, while the new code takes only about 564 instructions. In our design, these instructions can be eecuted in parallel with other instruction to obtain more efficiency.

The reason not using another loop for the second iteration is the pipeline issue. There's enough instructions independent from those parallel ones for programmers to fill into the

```
LOOP: ( four times ) {
  …
  /* calculating the base of block*/
  /* push registers */
  …
    LOOP: ( four times ) {
        …
        /* calculating the base of pixel to load*/
        …
        ACC = 0
        R_to_load = MIN ( R_UpperBound , R_ref_pel ) || NOP || R_coef = [ I2 ++ ]
        LOOP: ( six times )
              R_to_load = MAX ( R_LowerBound, R_toload )
              P_to_load = R_to_load                    //check boundaries done
              R_ref_pel = R_ref_pel + R3               //next pixel to reference ( R3 = 1 )
              P_to_load = P_base + P_to_load           //offset
              R_tmpresult = B [ P_to_load ] ( Z )      //load pixel
              R_to_load = MIN ( R_UpperBound , R_ref_pel )|| NOP || R_coef = [ I2 ++ ]
              ACC += R_tmpresult.L * R_coef.L (IS)  //calculate result
        R_result = ACC.W
        ...
        /* calculating the address of block[i][j] and stores the result into it */
        ...
    }
  /* pop back values in register */
}
```

Figure 2.10: Original code sequence

```
LOOP: ( four time ) {
   …
  /* calculating the base of block and the pixel to load, then push registers */
   …
   R_to_load = MIN ( R_UpperBound , R_ref_pel ) || NOP || R_coef = [ I2 ++ ]
   LOOP: ( six times )
        R_to_load = MAX ( R_LowerBound, R_toload )
        P_to_load = R_to_load                          //check boundaries done
        R_ref_pel = R_ref_pel + R3                     //next pixel to reference ( R3 = 1 )
        P_to_load = P_base + P_to_load                 //offset
        R_tmpresult = LDMVBF [ P_to_load ]             //load pixel
        R_to_load = MIN ( R_UpperBound , R_ref_pel ) || NOP || R_coef = [ I2 ++ ]
        ACC += R_tmpresult.L * R_coef.L (IS)           //calculate result
   R_result = ACC.W
   ...
   /* calculating the address of block[i][j] and stores the result into it*/
   ...
   LOOP: ( three times ) {
      ACC += R_tmpresult.L * R_coef.L (IS) || R_coef = [ I2 ++ ]|| R_tmpresult = BFLOAD
      ACC += R_tmpresult.L * R_coef.L (IS) || R_coef = [ I2 ++ ]|| R_tmpresult = BFLOAD
      ACC += R_tmpresult.L * R_coef.L (IS) || R_coef = [ I2 ++ ]|| R_tmpresult = BFLOAD
      ACC += R_tmpresult.L * R_coef.L (IS) || R_coef = [ I2 ++ ]|| R_tmpresult = BFLOAD
      ACC += R_tmpresult.L * R_coef.L (IS) || R_coef = [ I2 ++ ]|| R_tmpresult = BFLOAD
      ACC += R_tmpresult.L * R_coef.L (IS) || R_coef = [ I2 ++ ]|| R_tmpresult = LDMVBF [ P_to_load ]
      …
      /*stores the result into block[i][j]*/
      …
   }
}
```

Figure 2.11: The code sequence applying BFLOAD and LDMVBF instructions

slot.

Finally, we list all the new instructions proposed in Table. 2.1.

Table 2.1: Descriptions of new instructions

Instruction Syntax	Descriptions
$dest.Preg = CHKB(src1.Preg, src0.Preg)$	$dest = max(0, min(src1, src2))$ used to check boundaries
$dest.Dreg = BFLOAD$	$dest = buffer[0](Z)$ $buffer[0] <= buffer[1]$ $buffer[1] <= buffer[2]$ $buffer[2] <= buffer[3]$ $buffer[3] <= buffer[4]$ $buffer[4] <= buffer[0]$ used to get the value stored previously in buffer and rotate the buffer by 8-bits
$dest.Dreg = LDMVBF[src.Preg]$	$dest = B[src](Z)$ $buffer[0] <= buffer[1]$ $buffer[1] <= buffer[2]$ $buffer[2] <= buffer[3]$ $buffer[3] <= buffer[4]$ $buffer[4] <= B[src]$ used to store the loaded value and shift the buffer by 8-bits