Preliminary Application Note

IMAPCAR Series Processor

SIMD Programming Techniques

Software

IMAPCAR2 Series

IMAPCAR2-200 (uPD720804)

IMAPCAR2-100 (uPD720803)

IMAPCAR2-50 (uPD720802 & uPD720801)

Document No. U20011EE1V0AN00
Date Published: October 2009

Legal Notes

The information in this document is current as of October 2009. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.

- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.
- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".
  The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.
  "Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.
  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).
  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

The quality grade of NEC Electronics products is "Standard" unless otherwise expressly specified in NEC Electronics data sheets or data books, etc. If customers wish to use NEC Electronics products in applications not intended by NEC Electronics, they must contact an NEC Electronics sales representative in advance to determine NEC Electronics' willingness to support a given application.

(Note)

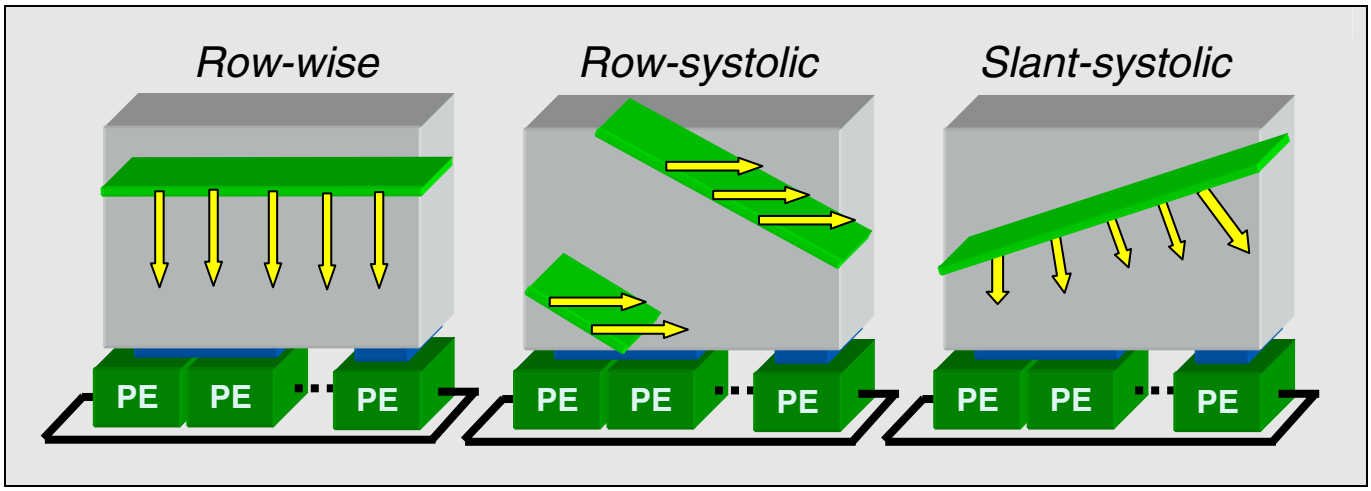
(1) "NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.

(2) "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

Table of Contents

1 Introduction

1.1 Implementation concept



Standard SIMD architectures can process row-wise only mainly due to the memory architecture of the processors (global shared memory). Row-wise approach is used to solve any operation without any dependency.

Due to the very flexible memory model of the processing elements, each processor can locally define its address, such independent processing can occur. Row systolic is used to solved combined top-down and left-right dependency

Slant systolic approach is a derivate from Row systolic but can solve an additional recursive dependency or border effect.

```
/* Implementation concept
 * ----------------------
 * [ ] Row wise
 * [ ] Row-Systolic
 * [ ] Slant-systolic
 * [ ] Rotation by 90°
 */
```

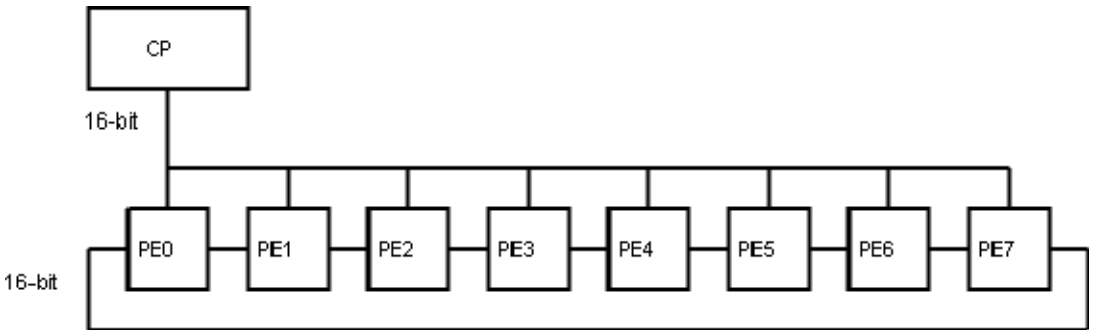
1.2 Code improvements

```
/* Code improvement
 * ---------------
 * [ ] Limit the number of scalar operations. e.g. ":["
and ":]"
 * [ ] Use of 16bit operands instead of 8bit
 * [ ] Factorize the operations
 * [ ] Loop unrolling : partial / complete
 * [ ] Use trinary operators to replace mif statements
 * [ ] Avoid usage of extern variables (No-Cache access
variables)
 */
```

2 Addition of data in a vector processor

A sep variable type is a declaration of a variable for every processing element.

The function addsep makes the addition of a sep variable among the processing elements and stores the result into a scalar variable located in the control processor local memory.



2.1 Scalar way using the control processor (C like)

Standard C programmers would write the addsep code by adding each element one by one using a loop.

Example 1

```
uint addsep_1(sep uchar sep_data){
  uint i,sum_data=0;

  for(i=0;i<PENO;i++) sum_data = sep_data:[i:];

  return sum_data;
}
```

- Execution time: 90 steps @ 64PEs XCDEV version 1.50

2.2 Systolic approach

Using the shift operation on the PE array directly, the efficiency can be improved.

Example 2

```
uint addsep_2(sep uchar sep_data){
  uint i,sum_data=0;
  sep uint sum_sep_data=sep_data;

  for(i=1;i<=(PENO/2);i*=2)
    sum_sep_data = sum_sep_data:<i;

  return sum_sep_data:[0:];
}
```

- Execution time: 81 steps@ 64PEs XCDEV version 1.50

2.3 Unrolling the loop keeping the vector addition

Unrolling the loop while using the shift operation on PE array helps "VLIWization" of the instructions and results in a better performance.

Example 3

```
uint addsep_2(sep uchar sep_data){
  sep uint sum_data;

  sum_data=sep_data;

  sum_data+=sum_data:<1;
  sum_data+=sum_data:<2;
  sum_data+=sum_data:<4;
  sum_data+=sum_data:<8;
  sum_data+=sum_data:<16;
  if(PENO==32) return sum_data:[0:];
  sum_data+=sum_data:<32;
  if(PENO==64) return sum_data:[0:];
  sum_data+=sum_data:<64;
  if(PENO==128) return sum_data:[0:];

  return 0;
}
```

- Execution time: 43 steps @ 64PEs XCDEV version 1.50

2.4 Using a mix of vector operation and scalar operation

The overall efficiency of each code of line is reduced by a factor two at every stage.

Using direct addressing for the final stage would reduce the execution time by a few cycles.

Example 4

```
uint addsep_3(sep uchar sep_data){
  uint i;
  sep uint sum_data;

  sum_data=sep_data;

  sum_data+=sum_data:<1;
  sum_data+=sum_data:<2;
  sum_data+=sum_data:<4;
  sum_data+=sum_data:<8;
  if(PENO==32) return sum_data:[0:] + sum_data:[16:];
  sum_data+=sum_data:<16;
  if(PENO==64) return sum_data:[0:] + sum_data:[32:];
  sum_data+=sum_data:<32;
  if(PENO==128) return sum_data:[0:] + sum_data:[64:];

  return 0;
}
```

- Execution time: 39 steps @ 64PEs XCDEV version 1.50

2.5 Multiple addition in parallel (basic)

Making multiple additions in parallel is often used in the applications. One approach would be to use four addesp one after each other.

Example 5

```
void addsep_5(sep uchar sep_data[4],uint data_sum_array[4]){
  sep_data[0]=addsep_3(data_sum_array[0]);
  sep_data[1]=addsep_3(data_sum_array[1]);
  sep_data[2]=addsep_3(data_sum_array[2]);
  sep_data[3]=addsep_3(data_sum_array[3]);
}
```

- Execution time: 171 steps @ 64PEs XCDEV version 1.50

2.6 Multiple additions with data parallelization

Another possibility is to parallelize additions using only one PE array. Suppose we initially have:

The last step will be to extract every summed data ‘sum_a, sum_b, sum_c & sum_d’ from the respective PE 0…3.

Example 6

```
void addsep_7(sep uchar sep_data[4],uint data_sum_array[4]){
  sep uint sep_sum;
  sep uint sum_temp[4];

  /* Prepare the data and sum with 4PEs into */
   mif(!(PENUM&0x01)){
     sum_temp[0]+=:<sum_temp[0];
     sum_temp[2]+=:<sum_temp[2];
   } melse {
     sum_temp[1]+=:>sum_temp[1];
     sum_temp[3]+=:>sum_temp[3];
   }

   mif(!(PENUM&0x02)){
     sum_temp[0]+=sum_temp[0]:<2;
     sum_temp[1]+=sum_temp[1]:<2;
   } melse{
     sum_temp[2]+=sum_temp[2] :>2;
     sum_temp[3]+=sum_temp[3] :>2;
   }
   sep_sum=sum_temp[PENUM&0x03];

   /* Sum with PEs in sep the 4 values in parallele */
   sep_sum+=sep_sum:<4;
   sep_sum+=sep_sum:<8;
   sep_sum+=sep_sum:<16;
   sep_sum+=sep_sum:<32;

   /* Sum & store with control processor the remaining subsum */
   data_sum_array[0] = sep_sum:[  0:]+sep_sum:[ 64:];
   data_sum_array[1] = sep_sum:[  1:]+sep_sum:[ 65:];
   data_sum_array[2] = sep_sum:[  2:]+sep_sum:[ 66:];
   data_sum_array[3] = sep_sum:[  3:]+sep_sum:[ 67:];
}
```

- Execution time: 138 steps @ 64PEs XCDEV version 1.50

- More details

Four vectors have to be added into four scalar values as shown in the table bellow:

PENUM	0	1	2	3	…
sep uchar a	a0	a1	a2	A3	…
sep uchar b	b0	b1	b2	b3	…
sep uchar c	c0	c1	c2	c3	…
sep uchar d	d0	d1	d2	d3	…

The goal is to make and gather the results in a different PE for each value, PE multiple of 4 take care of a values, PE multiple of 4(+1) take care of b and so on.

The step one would look like as follows

PENUM	0	1	2	3	…
sep uint l_a	a0+a1	x	a2+a3	x	…
sep uint l_b	x	b0+b1	x	b2+b3	…
sep uint l_c	c0+c1	x	c2+c3	x	…
sep uint l_d	x	d0+d1	x	d2+d3	…

The second step would be as follows:

PENUM	0	1	2	3	…
sep uint l_a	a0+a1+ a2+a3	x	x	x	…
sep uint l_b	x	b0+b1+ b2+b3	x	x	…
sep uint l_c	x	x	c0+c1+ c2+c3	x	…
sep uint l_d	x	x	x	d0+d1+ d2+d3	…

The final step is to gather those data into the same variable to enable the parallelization of the operation.

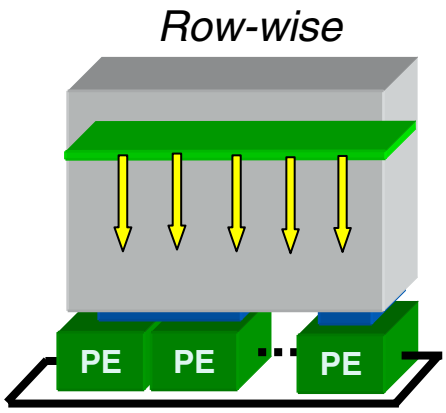
PENUM	0	1	2	3	…
sep uint l_sum	a0+a1+ a2+a3	b0+b1+ b2+b3	c0+c1+ c2+c3	d0+d1+ d2+d3	…

The rest of the algorithm is very common to the original addsep procedure.

3 Filter code writing

An array of sep is declared in this example to form a 2D memory space for storing an image.

The image filters usually have no special dependency to previous result and are implemented in a row wise approach.



The filter studied in this example is a 3x1 high pass filter on an image of 128 pixels long by 128 pixels wide.

Matrix: [-0.5 0 0.5]

3.1 Row-wise approach

The standard implementation looks as in the example bellow.

Example 1

```
void filter1(sep uchar *src, uint nbline){

  uint i;
  for(i=0;i<nbline;i++){
    src[i]=128+((src[i]:<2)-(src[i]))/2;
  }
}
```

- Execution time: 9919 clock cycles @ 64PEs XCDEV version 1.50

3.2 Using 16-bit computing

The XC core is a 16-bit architecture and therefore, making a cast to 16-bit can help the compiler to provide more efficient code.

Example 2

```
void filter2(sep uchar *src, uint nbline){

  uint i;
  for(i=0;i<nbline;i++){
    src[i]=128+((((sep uint)src[i]:<2)/2)-((sep uint)src[i]/2));
  }
}
```

- Execution time: 3036 clock cycles @ 64PEs XCDEV version 1.50

3.3 Operations factorization

Division is a very costly operation in the XC core. Factorizing the division improves the computation time quite well.

Example 3

```
void filter3(sep uchar *src, uint nbline){

  uint i;
  for(i=0;i<nbline;){
    src[i++]=128+(((sep uint)src[i]:<2)-((sep uint)src[i]))/2;
  }
}
```

- Execution time: 2738 clock cycles @ 64PEs XCDEV version 1.50

3.4 Partial unrolling of loops

While the XC core is a VLIW architecture, unrolling the loops helps on maximizing the occupation rate of the VLIW.

Example 4

```
void filter4(sep uchar *src, uint nbline){

  uint i;
  for(i=0;i<nbline;i++){
    src[i++]=128+(((sep uint)src[i]:<2)-((sep uint)src[i]))/2;
    src[i]=128+(((sep uint)src[i]:<2)-((sep uint)src[i]))/2;
  }
}
```

- Execution time: 2226 clock cycles @ 64PEs XCDEV version 1.50

4 Histogram computation

Tests are carried out in 128 greyscale levels for simplicity.

4.1 Column histogram

The first step of the histogram computation is to calculate one histogram for each column of the image as displayed in the example bellow.

Example 1

```
...
for(i=0;i<LINES;i++){
     wrk[src[i]>>1]++;
}
...
```

The second step is to globalize the histograms by adding the histogram among the PEs.

4.2 Parallelization of the operation

The histogram addition can be added between the PEs using the previously developed addsep.

Example 1

```
void histogram_1(sep uchar wrk[],sep uint res[],int lines) {
  uint i,k;
  sep uchar systolic_index=PENUM;
  sep uint systolic_buffer,temp_res;

  for(k=0;k<PENO;k+=4){
     addsep_4(&wrk[k],&res[0],k);
     addsep_4(&wrk[k+PENO],&res[1],k);
  }
}
```

- Execution time: 10646 clock cycles @ 64PEs XCDEV version 1.50

4.3 Rotating the data

The XC core architecture is very powerful in top-down computation and at the same time very efficient in 90 degree rotation. Using rotation of data is often use the speed-up the process.

Example 2

```
void histogram_2(sep uchar wrk[],sep uint res[],int lines) {
  uint i,k;
  sep uchar wrk2[256],systolic_index=PENUM;
  sep uint systolic_buffer,temp_res;

  lxRot90(wrk,1,0,128);
  lxRot90(&wrk[PENO],1,0,128);

  for(k=0,res[0]=0,res[1]=0;k<PENO;k++){
     res[0]+=wrk[k];
     res[2]+=wrk[k+64];
  }
}
```

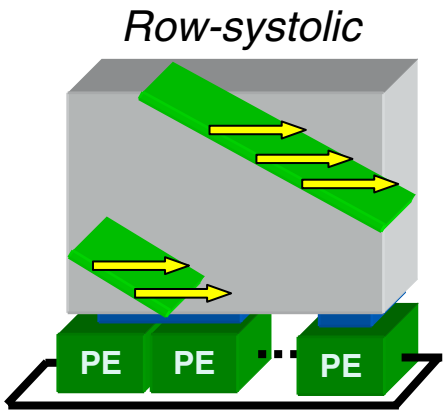
- Execution time: 5051 clock cycles @ 64PEs XCDEV version 1.50

4.4 Row-Systolic approach

The issue in the rotation is that it requires one more step which is pure time cost. The idea for improvement would be to take advantage of the flexibility of the architecture while making pseudo rotation and making the histogram collection in parallel.

Row systolic approach demonstrated here combines local PE addressing and left right communication.

A circular buffer is used as the accumulator and is shifted at every cycle. The column addressed by the addition depends on the processing element number and the iteration number.



Example 3

```
void histogram_3(sep uchar wrk[],sep uint res[],int lines) {
  uint i,k;
  sep uchar systolic_index=PENUM;
  sep uint systolic_buffer_1,systolic_buffer_2,temp_res;

  systolic_buffer_1=0;
  systolic_buffer_2=0;

  for(i=0;i<PENO;i++){    // [0 … 127]
    systolic_buffer_1+=wrk[systolic_index];
    systolic_buffer_1=:<systolic_buffer_1;
      systolic_index=:<systolic_index;
  }
  for(i=0;i<PENO;i++){    // [128 … 255]
      systolic_buffer_2+=wrk[systolic_index+128];
    systolic_buffer_2=:<systolic_buffer_2;

     systolic_index=:<systolic_index;
   }
  res[0]=systolic_buffer_1;
  res[1]=systolic_buffer_2;
}
```

- Execution time: 1909 clock cycles @ 64PEs XCDEV version 1.50

4.5 Factorization

Factorization intends at reducing the control cost against the computation cost by having a single loop making both operation in parallel. This is possible because both stages are fully independent each other.

Example 4

```
void histogram_4(sep uchar wrk[],sep uint res[],int lines) {
  uint i,k;
  sep uchar systolic_index=PENUM;
  sep uint systolic_buffer_1,systolic_buffer_2,temp_res;

  // Systolic addition of the Histogram
    systolic_buffer_1=0;
    systolic_buffer_2=0;

    for(i=0;i<PENO;i++){
      systolic_buffer_1+=wrk[systolic_index];
      systolic_buffer_1=:<systolic_buffer_1;

      systolic_buffer_2+=wrk[systolic_index+128];
      systolic_buffer_2=:<systolic_buffer_2;

      systolic_index=:<systolic_index;
    }
    res[0]=systolic_buffer_1;
    res[1]=systolic_buffer_2;
}
```

- Execution time: 1538 clock cycles @ 64PEs XCDEV version 1.50

5 Integral of an image

The basic equation for image integral looks as follows:

For each parameter [i ; j] from [0,0] to [N,N]:

integral_out(j, i) = image_in(j, i) + integral_out (j-1, i)

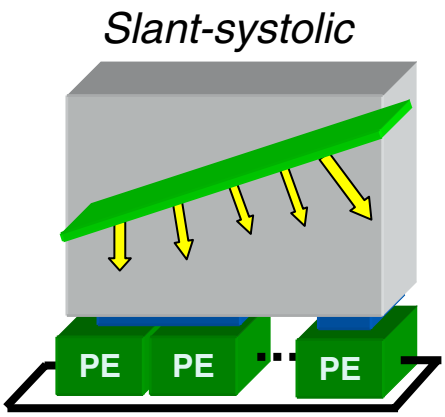
+ integral_out (j, i-1) – integral_out (j-1, i-1);

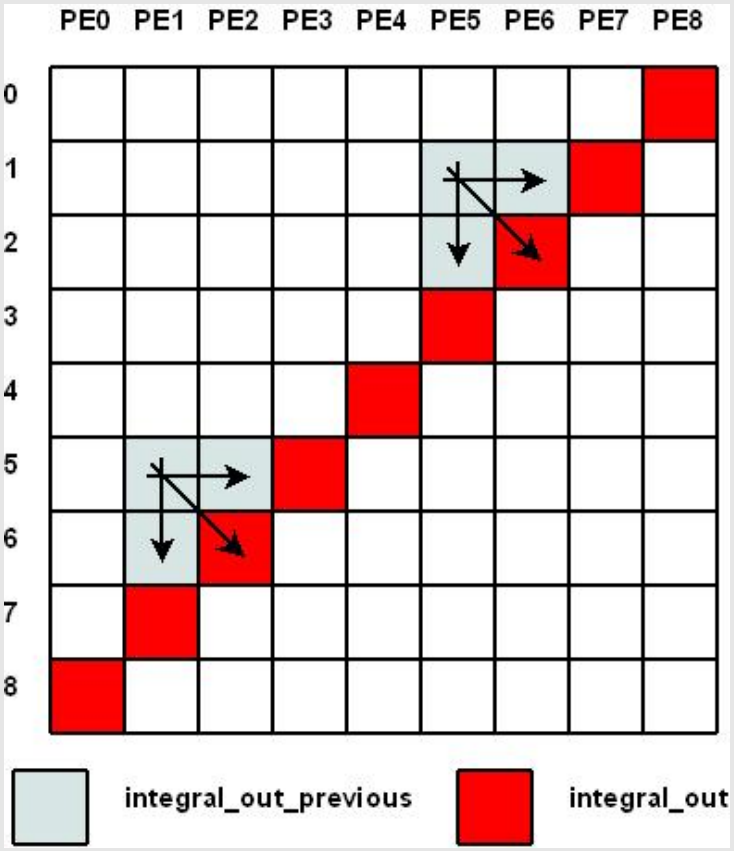
The equations make clear that there is a top-down and left-right dependency with the result of the previous computations.

The computation of a pixel at position [i,j] depends completely on the result of the top, left and top left previous results.

Moreover, the borders have to be initialized and therefore there is a very big need to take into account the border effects.

The method to implement such algorithms is the slant systolic approach as described bellow.





Slant systolic approach applied to image integral

5.1 mif based implementation

To take care of the border effect, the mif statement has to be used to distinguish between all different cases.

Example 1

```
void integral_1(sep uchar * src,sep ulong * src_out, uint nbline, uint stripe){

  int j;

  sep int nb_PENUM=PENUM;
  sep int LineNumber;
  sep uint jLineNumber;
  sep uint jLineNumber_tmp;

   sep ulong tmp, tmp_a, tmp_b,tmp_c,tmp_d;


   for(j=0;j<nbline+128;j++){

      LineNumber=j-nb_PENUM;

     mif((LineNumber>=0)&&(LineNumber<=(nbline))){

         jLineNumber = LineNumber;
         jLineNumber_tmp=jLineNumber-1;
         mif((PENUM==0)&&(jLineNumber==0))
           src_out[jLineNumber]=src[jLineNumber];

         mif ((PENUM == 0)&&(jLineNumber!=0))
           src_out[jLineNumber]=src[jLineNumber]+src_out[jLineNumber_tmp];

         mif ((PENUM != 0)&&(jLineNumber==0))
              src_out[jLineNumber]=src[jLineNumber]+(:>src_out[jLineNumber_tmp]);

         mif((PENUM != 0)&&(jLineNumber!=0))

src_out[jLineNumber]=(src[jLineNumber]+src_out[jLineNumber_tmp]+(:>src_out[jLineNumber_tmp]))-
(:>src_out[jLineNumber_tmp-1]);
      }
   }
}
```

- Execution time: 37880 clock cycles @ 64PEs XCDEV version 1.50

5.2 Operations factorization

The operation factorization consists in a preparation stage to collect the different operands.

Once collected, the global computation which is the most time consuming part is then just executed once.

Example 2

```
void integral_2(sep uchar * src,sep ulong * src_out, uint nbline, uint stripe){

  int j;

  sep int nb_PENUM=PENUM;
  sep int LineNumber;
  sep uint jLineNumber;
  sep uint jLineNumber_tmp;

   sep ulong tmp, tmp_a, tmp_b,tmp_c,tmp_d;


   for(j=0;j<nbline+128;j++){

      LineNumber=j-nb_PENUM;

     mif((LineNumber>=0)&&(LineNumber<=(nbline))){

         jLineNumber = LineNumber;
         jLineNumber_tmp=jLineNumber-1;

         tmp_a=src[jLineNumber];
         mif(jLineNumber!=0) tmp_b=src_out[jLineNumber_tmp];
         melse tmp_b=0;
         mif(PENUM!=0)tmp_c=:>src_out[jLineNumber_tmp];
         melse tmp_c=0;
         mif((PENUM!=0)&&(jLineNumber!=0))tmp_d=(:>src_out[jLineNumber_tmp-1]);
         melse tmp_d=0;

         src_out[jLineNumber]=(tmp_a+tmp_b+tmp_c)-tmp_d;

     }
   }
}
```

- Execution time: 26907 clock cycles @ 64PEs XCDEV version 1.50

5.3 Ternary operator

In the XC core, the ternary operation helps reducing the access to the external memory.

In a mif melse statement, the memory is accesses twice, one for the true and one for the false statement.

In a ternary operation, the condition is prepared in the register and the memory is accessed just once.

Example 3

```
void integral_3(sep uchar * src,sep ulong * src_out, uint nbline, uint stripe){

  int j;

  sep int nb_PENUM=PENUM;
  sep int LineNumber;
  sep uint jLineNumber;
  sep uint jLineNumber_tmp;

   sep ulong tmp, tmp_a, tmp_b,tmp_c,tmp_d;

   for(j=0;j<nbline+128;j++){

      LineNumber=j-nb_PENUM;

     mif((LineNumber>=0)&&(LineNumber<=(nbline))){

         jLineNumber = LineNumber;
         jLineNumber_tmp=jLineNumber-1;

         tmp_a=src[jLineNumber];
         tmp_b=(jLineNumber!=0)? src_out[jLineNumber_tmp]:0;
         tmp_c=(PENUM!=0)? (:>src_out[jLineNumber_tmp]):0;
         tmp_d=((PENUM!=0)&&(jLineNumber!=0))? (:>src_out[jLineNumber_tmp-1]):0;

         src_out[jLineNumber]=(tmp_a+tmp_b+tmp_c)-tmp_d;

     }
   }
}
```

- Execution time: 18374 clock cycles @ 64PEs XCDEV version 1.50

6 Revision history

Version	Date	Document Number	Description
1.0	Oct. 2009	U20011EE1V0AN00	First version. XCDEV v1.50

The following revision list shows all functional changes compared to the previous version.

Chapter	Page	Description