MAPP2200 Smart Vision Sensor. Programmability and Adaptivity

Anders Åström and Robert Forchheimer
IVP Integrated Vision Products and
Department of Electrical Engineering, Linköping University

S-581 83 LINKÖPING, Sweden
email: andersa@isy.liu.se

Abstract

This paper briefly describes the architecture and functions of the smart vision sensor MAPP2200 and shows examples of the assembler syntax which forms the basis of the software development. Two application examples are given and an implementation of adaptive exposure control for MAPP2200 is presented.

1 Architecture

1.1 General

MAPP2200 Vision Sensor is an optical sensor component that includes a digital image processor. Samples of this component have been available since August-91. A block diagram of MAPP is shown in Figure 1. For a more detailed description of MAPP see [5].

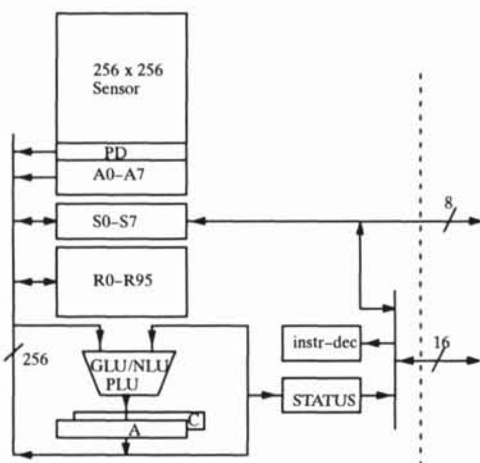

Figure 1, Block diagram of MAPP2200

1.2 Sensor

The sensor area occupies about half of the silicon area and consists of 256 rows with 256 photo-diodes each, see upper part of Figure 1. The image data is read out row-wise in parallel to an analog register (PD). This register content can be converted into digital values in two ways. Either to binary values using a threshold voltage, or to gray scale values using the internal A/D-converter (A0-A7). This A/D-converter can be programmed to convert with different precision, i.e. from 1 bit up to 8 bits, depending on the requirements on speed and precision.

1.3 Processor and instruction set

The lower parts of Figure 1 show the digital section of MAPP. The shift register S0-S7 is an 8 bit bi-directional shift register which can be used for external communication of data or for communication between processors. The processor block consists of 256 parallel processor elements (PE). The internal memory for each PE is 96 bits, R0-R95. The arithmetic logic unit consists of three parts. The Global Logical Unit, GLU, performs global operation such as LFILL which sets all the accumulator bits to 1 which are to the right of the leftmost 1. The Neighborhood Logical Unit, NLU, can be set to apply an arbitrary 3 by 1 binary kernel using the PE and its two neighbors. For instance, (1X0) means that if the left neighbor is 1 and the right is 0 the result is 1, otherwise 0. The X stands for don't care. The Point Logical unit, PLU, operates locally in each PE. Typical operations are AND, OR, XOR, and so on. The following list is a brief summary of all instructions in MAPP.

```
PLU       LD, AND, OR, XOR, LDI, ANDI, ORI,
          XORI, ADD, ADC, ADA, ANDC, ORC, XORC,
          MUXC
NLU       Arbitrary (a,b,c) ; a,b,c ∈ 1,0,X
GLU       MARK, LMARK, RMARK, LRMARK, FILL,
          LFILL, RFILL, LRFILL
Transfer  ST {A,C,Ax,PD},S/R, ST S,S/R, ST R,S
Misc      SETR(ow), SETV(oltage), SETPD, SETAD,
          SETB(oundary), SAVE, LOAD, ROL, ROR,
          INITAD, SRES, INITPD, READPD
```

By reading the status register COUNT we obtain the number of 1's in the accumulator which, we will show, is useful in many applications.

2 Programming environment and syntax

2.1 Syntax

The MAPP2200 is programmed by using a MAPP2200 Assembler for the MAPP instructions and a C program for the controlling process. A simple MAPP program may look like this:

```
.FILE test.h
.PROC M_init
   SETAD STEP1
   SETR 0
.ENDP
.PROC M_run
   INITPD,+      ; Reset this line
   READPD        ; Read next line
   INITAD        ; A/D-conversion
   .ITER(8 i)    ; From A/D-reg to S-reg
     LD A(0 i)
     ST A,S(0 i)
   .ENDI
.ENDP
.ENDF
```

When this source code is run through the assembler the program is converted into arrays of short integers, i.e. 16 bits codes. Each procedure generates its own array. For example, the output file in the above example would contain two arrays of short integers called M_init and M_run. This file is then included in the final C program:

```
#include "test.h"
#include "mapp.def"
main()
{
  int i;
  while (TRUE) {
    writecode(M_init);
    for (i=0;i<256;i++) {
      writecode(M_run);
      Store(0,i); /* Output device */
    }
  }
}
```

The include file mapp.def contains the definitions of writecode, Store, and other MAPP driver routines.

Macros are used to simplify and minimize MAPP programming. The macros are expanded using the C preprocessor. The macro must be defined within the program, or in a file included in the MAPP program. A standard macro library has been developed which contains arithmetics and filter functions.

2.2 Bit-serial programming

As MAPP does not have a multi-bit ALU we have to perform all arithmetics in a bit-serial fashion. A MAPP2200 routine which adds two b-bit operands looks like:

```
LD R(X)
ADD R(Y)
.ITER (b -1 addi)
  ADC R(X 1 addi)
  ADA R(Y 1 addi)
  ST A,R(Z addi)
.ENDI
ST C,R(Z b -1)
```

In order to aid the MAPP2200 programmer a number of bit serial routines have been developed. Examples of routines in this library are: add, subtract, add 2-comp, sub 2-comp, mult, mult 2-comp, multiplex, etc.

2.3 2D filters

As MAPP is a linear array of processors we only have access to one line of pixels at a time. To implement 2D filters we have to use the registers as temporary storage for previous rows. This means that we need to store a number of previous row/results to obtain the following rows. This type of processing is called Row-parallel Pipe-lining, see [1]. A typical C program may look like the one in section 2.1. The include file test.h again contains two MAPP code sequences, M_init and M_run. The M_init resets the registers used and sets up the ADC etc. The M_run contains the code for the filter. A typical macro call from the standard filter library in MAPP looks like this:

```
LOWPASS(IN,PREV,PPREV,OUT,TMP,b)
```

IN is the location of the input register group to the routine. PREV and PPREV are two previous results which must be stored between each run. OUT is the location where the result is put. TMP is the location of the temporary memory needed. b is the number of bits used in the routine. The routine uses a 3x3 lowpass filter which can be separated into 4 additions, see [4]. It is very easy to combine a number of macros. This enables us both to perform different operations and to use larger kernels. For instance:

```
LOWPASS(IN,PREV1,PPREV1,TMP1,TMP2)
LAPLACE2(TMP1,PREV2,PPREV2,OUT,TMP2)
```

In this case the output from LOWPASS is passed directly to LAPLACE2. Note that we can use the same temporary memory in both routines. The effect we get is that we will have more smoothing in the image as we now apply the following filter:

```
                          1  4   6  4  1
1 2 1     1   2  1        4  0  -8  0  4
2 4 2  *  2 -12  2   =    6 -8 -28 -8  6
1 2 1     1   2  1        4  0  -8  0  4
                          1  4   6  4  1
```

3 Examples

3.1 Moment calculation

This routine is based on the ideas in [3] which describes a moment calculation performed on an SIMD array. The computation is separated into a vertical and a horizontal computation. The input of the image, as a binary image through the comparators, is done during the vertical computation where the moments m_{x0}, m_{x1}, and m_{x2} are computed. None of these computations are performed as multiplications as they can be separated into additions. Each PE computes the moments m_{x0}, m_{x1}, and m_{x2} in its own column. When the vertical scan is completed we have to perform the horizontal computation in order to obtain

the moments m_{00}, m_{01}, m_{02}, m_{10}, m_{11}, and m_{20}. The first three of these moments can be obtained as a horizontal summation of m_{x0}, m_{x1}, and m_{x2}, which is done using the powerful count feature. For example, if m_{x0} is placed in register 0 to N-1 we compute m_{00} as:

```
m00 = 0;
for (i=0;i<N;i++) {
  writecmd(LD | i);
  m00 += (COUNT)<<i;
}
```

To compute m_{10}, m_{11}, and m_{20} we have to perform a multiplication followed by the global addition described above. The value to multiply by will be a constant vector ranging from 0 to 255, corresponding to PE 1 to 256. By using certain tricks the constant vector can be generated in approximately 256 cycles. To obtain m_{10}, m_{x0} is multiplied by the constant vector and then globally added together. If we multiply m_{x0} once again by the constant vector we will get m_{20}. Finally, m_{11} is obtained by multiplying m_{x1} by the constant vector. This can be summarized to:

64500	MAPP instructions
112	μCtrl instructions "load and add loops"

This yields a total of approximately 70000 MAPP instructions, which in a 4MHz system corresponds to 60 frames/s.

Given the 6 moments the total area will be m_{00}. We can further compute the center of gravity, $\bar{x}$ and $\bar{y}$, and the slope of the principle axis, Θ.

3.2 Segmentation and moment

The previous moment calculation computes the moments of the whole image as if it contained only one object. However, if we want to compute the moments for several objects we have to perform a segmentation. The idea is to scan the image vertically and to make use of the global logic unit, GLU, to enable a run-length, R-L, coding of the image. The reason to perform a R-L coding of the image is to check the connectivity between objects. furthermore, the moments can easily be obtained from the R-L code. Figure 2 shows that by keeping the previous row we can see if two objects becomes one or if an object splits into two branches. We now keep a list containing all objects and their moment data. If two objects turn up to be one object the moment data are added together and all references to the second object are changed to the first object.

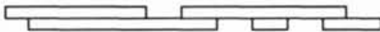

Figure 2

Each object has 6 moment variables m_{00}, m_{01}, m_{02}, m_{10}, m_{11}, and m_{20}. They are incremented as follows, given that we are at row R, with an object that starts at position P, and with a length of L.

```
m00 += L
m01 += L*R
m02 += L*R*R
m10 += L*(2*P+L-1)/2
m20 += L*L*L/3+L*L*(6*P-3)/6+L*(6*P*P-6*P+1)/6
m11 += R*L*(2*P+L-1)/2
```

After having scanned the whole image we end up with a list of objects and their moments which can be used to compute the center of gravity and the slope of the principal axis.

4 Adaptivity

4.1 Introduction

Adaptivity to different light conditions is achieved by utilizing the ability to COUNT the number of pixels which have passed a certain threshold in one cycle. If the image is over-exposed the Inthist diagram (the integrated histogram) might look as shown in Figure 3(a). If on the other hand the image is under-exposed Inthist would look like Figure 3(b). The idea behind an adaptivity routine for MAPP2200 is to change the exposure time so that X % of the pixels have the pixel value Y or higher, see Figure 3(c).

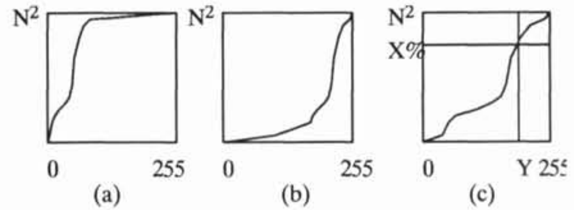

Figure 3

The number of pixels which have the pixel value Y or higher is acquired by comparing the analogue value of the pixel with Y followed by reading the "COUNT" status. This is done row wise, before the normal A/D-conversion is started.

4.2 Mathematical description

An image distribution can be modelled by the Rayleigh distribution:

$$p(x) = \frac{x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}} \quad (1)$$

$$P(x) = 1 - e^{-\frac{x^2}{2\sigma^2}} \quad (2)$$

In the MAPP system the pixel value is measured as the voltage over the photo diode which is pre-charged at a certain time. The relation between voltage and exposure time is given by (3). The voltage over the photo-diode can not be negative which means that all pixel values corresponding to a negative voltage are mapped to V=0.

$$V(t) = V_0 - kIt \quad (3)$$

To find the optimal exposure time we compute the squared difference between P(x) and a linear Inthist = (x/a x≤a & 1

x > a), (4), and its derivative (5), which is equal to zero when (6). We can now estimate a value of P(x) in terms of a. For instance, P(a/2) = 0,49, which means that 49% of all pixel should have a pixel value below 128. This is more fully discussed in [2].

$$S = \int_0^a \left(P(x) - \frac{x}{a}\right)^2 dx + \int_a^\infty (P(x) - 1)^2 dx \qquad (4)$$

$$\frac{dS}{da} = \frac{1}{3} - 2\frac{\sigma^2}{a^2} + 2\frac{\sigma^2}{a^2}e^{-\frac{a^2}{2\sigma^2}} \qquad (5)$$

$$\frac{a^2}{\sigma^2} = 5,56 \qquad (6)$$

To control the exposure time we can adjust the offset and a wait loop within the main loop. The exposure time is then the time from the precharge until the photo diode is read out as shown in the following program:

```
while (TRUE) {
        for (i = 0 ; i < 256 ; i + + ) {
                reset_row(i + m);
                read_row(i);
                compute();
                wait(n);
        }
        Update(m.n);
}
```

The time to perform one inner loop when n = 0 is t_a and each wait loop takes t_b which in the general case, for a given offset = m and n means an exposure time as,

$$t_{tot} = t_a m + t_b mn \qquad (7)$$

For a given x the number of pixels which have passed the threshold is shown in (8) leading to (9).

$$N = Me^{-\frac{x^2}{2\sigma^2}} \qquad (8)$$

$$\ln\frac{N}{M} = -\frac{1}{2}\frac{C^2}{t_{tot}^2} \qquad (9)$$

For a given t_{tot} and its corresponding $N = N_1$ the optimal value of N is acquired by adjusting t_{tot} as in (10).

$$\hat{t_{tot}} = t_{tot}\alpha\sqrt{\frac{\ln\frac{N_1}{M}}{\ln\frac{\hat{N}}{M}}} \qquad (10)$$

The α value is a correction factor which compensates for differences between the model and the actual image. New value of m and n are obtained first from (10) and then from (11) and (12). This is described in [2] .

$$\hat{m} = \left\lceil \frac{\hat{t_{tot}}}{t_a} \right\rceil \qquad (11)$$

$$\hat{n} = \frac{\hat{t_{tot}} - t_a\hat{m}}{t_b\hat{m}} \qquad (12)$$

This is done to ensure that we do not use a longer wait loop then necessary (13). The reason to do so is that the offset value do not alter the frame rate whilst the wait loop slows it down.

$$t_a m + t_b mn < t_a(m + 1) \qquad (13)$$

5 Conclusions

We have briefly described the architecture and functions of the smart vision sensor MAPP2200. We then showed examples of the assembler syntax which forms the basis of the software development. Two application examples were given which demonstrated the generality of the processor. Finally, we showed that adaptive exposure control can be implemented very easily in MAPP2200.

Acknowledgement

The authors would like to thank Jan-Erik Strömberg for valuable discussions regarding automatic control of the exposure adaptivity.

References

[1] Åström A., *A Smart Image Sensor. Evaluation and Description of PASIC.*, Lic Thesis, Linköpings University 1990.

[2] Åström A., Forchheimer R., *MAPP2200 Smart Vision Sensor. Programmability and Adaptivity.*, Internal Report, Linköping, Sweden, 1992.

[3] Chen K., *Fast Algorithm for the Calculation of Image Moments in a Linear Processor Array*, Internal Report, Linköping, Sweden, 1988.

[4] Danielsson P.E., *Generalized and Separable Sobel Operator*, Internal Report LiTH-ISY-I-0975, Linköping, Sweden, 1989.

[5] Forchheimer R., Ingelhag P., Jansson C., *MAPP2200, a second generation smart optical sensor*, SPIE Vol 1659 (1992)