8 Hardware Outline Character Rasterization

Marc Morgan
Roger D. Hersch

ABSTRACT This paper presents the design and implementation of an application specific integrated circuit (ASIC) for real-time rasterization of characters described by their outline based on vertical scan-conversion and flag fill algorithms. The chip acts as a coprocessor which rasterizes outline fonts given by Bezier splines and straight line segments. It generates high quality fonts at a rate 30 times higher than the equivalent assembly language code on a 16 MHz M68020.

8.1 Introduction

For several years, engineers and typographers have been working on a new generation of fonts. The old bitmap description of a character is giving way to characters defined by their outlines. This new description of a character makes a font independent of the size of the desired bitmap character. This in turn makes the font independent of the resolution of the hardware which will be used to render the text.

Studies undertaken at the Peripheral Systems Laboratory (LSP) at the Swiss Federal Institute of Technology (EPFL) have led to an algorithm for the rasterization and filling of outline characters. This algorithm is the base of an intelligent font rasterization program [3]. The fonts are given by outlines described by cubic Bezier splines and by line segments. The quality of the resulting black and white bitmaps depends on two important aspects of the program: the sub-pixel precision used during the rasterization and an appropriate control of the phase of the contour relative to the pixel grid.

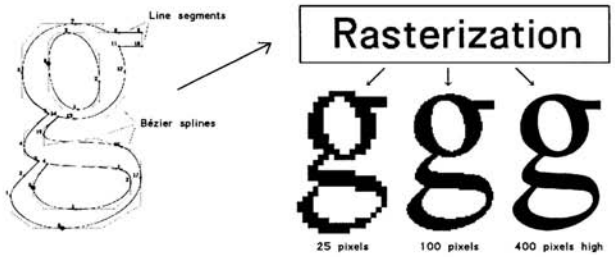

FIGURE 8.1. Outline fonts can be transformed into bitmaps of any scale without redesigning the characters

8.2 Motivation

The generation of a whole font (ASCII codes 32 to 126) by the intelligent character rasterization program [4] at a resolution of 300 dpi has been evaluated on a 16MHz M68020. Figure 8.2 shows that, in the program, grid outline adaptation (phase control) is independent of the size of the characters. However, the scan-conversion and filling times become dominant for large characters. Characters are generated at a rate of 50 characters per second at 600 dpi. This is clearly not sufficient for a high performance raster image processor. Since time consuming parts of the program are already written in assembly language, further performance improvements can be obtained by designing an application specific IC (ASIC).

Given the complexity of intelligent character rasterization, it is not possible to implement all its functionalities into a circuit. It was chosen to assign the ASIC the task of the highly repetitive part of the algorithm leaving the more complex work to the main processor. In other words, only scan-conversion and filling were integrated on the ASIC. Phase control is much more complex and requires the programming capabilities of a general-purpose processing unit.

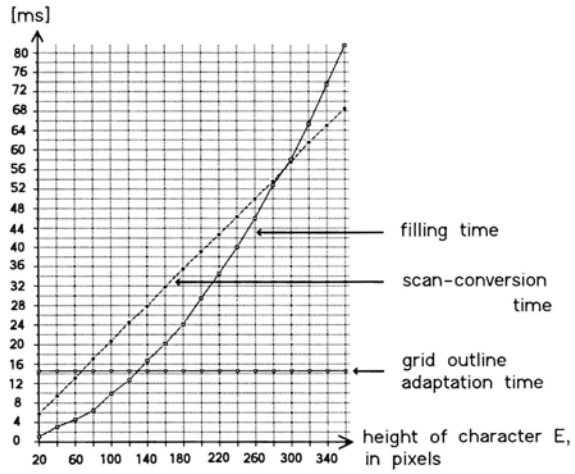

FIGURE 8.2. Average generation time of a character by software

This ASIC is mainly intended for high resolution (600 dpi). At such a high resolution, only simplified phase control is required. This ASIC can also be used for average resolution printers (300 dpi) whereby good quality results can only be achieved with complete phase control.

8.3 The Environment

The ASIC which has been designed is a simple coprocessor. It has a circular input buffer to which the main processor writes the character outline data. Once the coprocessor is ready to return the bitmap character, it sends an interrupt signal to the main processor which can then read the bitmap words sequentially. Synchronisation between the processor and the ASIC is achieved by standard interrupt request and handshake signals (CircularBufferEmpty, Full, and HalfFull).

The first prototype of the circuit has been designed to generate raster characters up to 64 pixels high and wide. This prototype was realised to demonstrate the feasibility of such a circuit. The final version of this circuit would need to generate larger characters (256 x 256 pixels for example).

8.4 The Algorithms

The principal task of the ASIC is to scan-convert the outline of the character which it will then fill before sending the resulting bitmap back to the main processor.

8.4.1 The Flag Fill Algorithm

The flag fill algorithm used by the ASIC is one [1] which was improved for the purpose of accurate shape filling [4]. The basics of the algorithm will quickly be exposed here.

A pixel is considered as being inside an outline if more than 50% of its surface lies inside the outline. For characters, the element of the outline which crosses a pixel can be considered as a line segment. Therefore, a pixel is inside an outline if its center is inside.

The bitmap which will be generated by the flag fill algorithm can be considered as a set of black horizontal spans for the inside of the outline and white horizontal spans for the outside. The first pixel of each span is marked by a flag. Once all the flags corresponding to an outline have been set, the flag fill algorithm scans the flag image memory from left to right. Each flag encountered indicates the start of a new horizontal interior or exterior span.

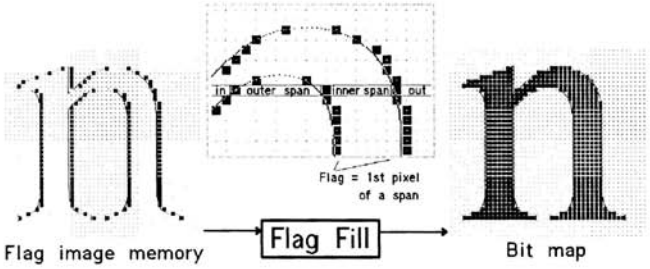

FIGURE 8.3. Example of the flag fill algorithm applied to a character

8.4.2 The Vertical Scan-Conversion Algorithm

The Bezier splines and line segments which make up an outline have to be converted into flags for the filling algorithm. Two strategies can be adopted to scan-convert a Bezier spline: recursive subdivision or forward differencing [7]. Both strategies have been developed in order to reduce the number of required operations without reducing the precision of the scan conversion.

Ordinary forward differencing had one main drawback: the incremental step of the parameter used to describe the curve was a constant. Adaptive forward differencing (AFD) corrected this problem [5]. AFD insures that most of the points which are generated will be used to trace the curve. Integer AFD further improved the algorithm by using fixed point or pseudo floating point arithmetic instead of floating point arithmetic [6] [2]. The resulting algorithm is yet faster.

Recursive subdivision has also been optimized [4]. It presents several advantages over forward differencing. First, computation errors aren't amplified in the same way as in AFD. In order to get the same quality result, recursive subdivision requires a significantly smaller amount of bits of precision than AFD. Second, recursive subdivision can be carried out with the control points of a Bezier curve rather than its polynomial equation. This allows for a better understanding and monitoring of the algorithm.

8.4.3 Scan-Conversion by Recursive Subdivision

Recursive subdivision of Bezier splines is based on DeCasteljou's theorem. As figure 8.4 shows, a Bezier spline:

$$P(u) = V_0 \cdot (1-u)^3 + V_1 \cdot 3u(1-u)^2 + V_2 \cdot 3u^2(1-u) + V_3 \cdot u^3 \qquad \text{with} \quad u \in [0,1]$$

represented by its control polygon (V_0, V_1, V_2, V_3) can be subdivided into two smaller Bezier splines, (V_0, S_1, S_2, S_3) and (S_3, T_1, T_2, V_3). The smaller splines will have their control polygons closer to the spline. Therefore, if a spline is subdivided enough times, the resulting control polygons become a sufficient approximation to the spline. One of the delicate points of the algorithm is the criterion for stopping subdivision. It is based on the convex hull property of Bezier curves: a Bezier curve always lies within the convex hull formed by its control polygon.

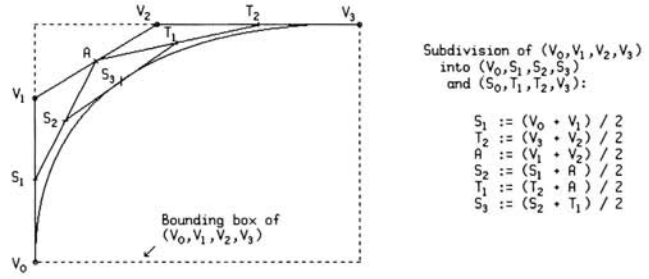

FIGURE 8.4. DeCasteljou's subdivision of Bezier splines

Repeated subdivision of Bezier splines can result in three types of Bezier splines:

- splines which don't intersect any scan line, and which can be discarded since they won't generate any flag,
- splines which don't intersect any vertical grid lines, and can be replaced by vertical line segments,
- splines which still intersect a scan line and a vertical grid line.

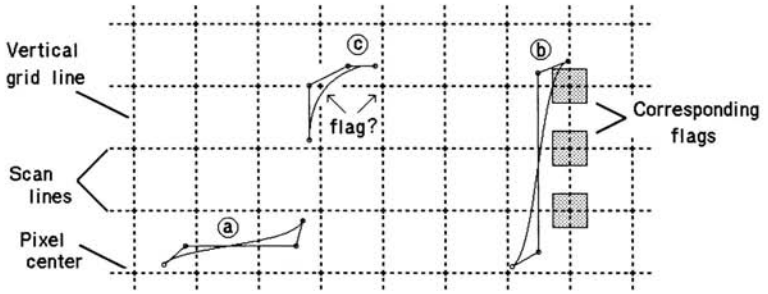

FIGURE 8.5. The three possible final outcomes of a Bezier subdivision

The third case sets the problem of knowing when to stop subdividing a spline and how to choose the position of the corresponding flag from where the subdivision left off. The choice of stopping a subdivision has to take into account the amount of errors which are tolerated and the precision used in the computations. The criterion we chose tests whether the spline's bounding box is smaller than the parameter *MinFracLength* along both the x and y axis. When this criterion is met, the small spline is replaced by the three line

segments which make up its control polygon. Experience shows that a good trade off for *MinFracLength* is 1/8 pixel.

The ASIC implementation of the subdivision-stopping criteria is slightly simplified by an additionnal hypothesis: all the splines which make up an outline are monotone both along x and y. This hypothesis implies that the font designer will have to be careful to divide any splines which aren't monotone into two or more splines which are. Note however that it is very rare that non-monotone cubic splines be used. A simple filter can also be applied to divide non-monotone splines into several monotone splines.

8.4.4 Comparison of Recursive Subdivision and Forward Differencing

The same criteria which we developed for recursive subdivision could be applied to adaptive forward differencing (AFD). This however would require precomputing the next point on the curve before deciding whether the step size is correct or needs adjusting. This corresponds to an extra addition for each adjust up or down operation. Even so, when *MinFracLength* has been reached, the curve would have to be replaced by a line segment, which is less accurate than the control polygon given by recursive subdivision.

Recursive subdivision requires 6 adds and 6 shifts to generate 2 sub-splines (figure 8.4). Integer AFD requires 3 adds, 2 shifts and an increment to generate each new step and for each adjust up or adjust down operation. Both algorithms therefore require approximately the same amount of arithmetic operations. However, integer AFD requires dynamical shift operations which are avoided by recursive subdivision. This tends to make AFD slower than subdivision. On the other hand, the recursive aspect of subdivision has to be implemented with a stack. Stack access will slow down subdivision. This problem can be partially eliminated by accessing the stack in parallel with other operations. This is true for the ASIC presented here which writes into the stack at the same time as it computes other control points or as it tests the locations of spline control points.

Another important difference between the two methods comes from the number of bits which are worked on. For example, for a maximum curve length of 64 pixels and with *MinFracLength* = 1/8 pixel, recursive subdivision requires 14-bit operators. Ordinary forward differencing would require 35 bits to get the same precision [6] while AFD would need 20 bits. AFD therefore would work on 43% more bits than recursive subdivision. Having more bits slows down arithmetic operations and requires more place on the integrated circuit.

8.4.5 Realization

The ASIC was designed to optimise the scan-conversion of Bezier splines. The processing of line segments only uses part of the resources which are required by the splines. It is also based on recursive subdivision.

The last step in the processing of a character consists in returning the filled bitmap to the main processor. The ASIC reads the flag image memory byte by byte and fills the outline on the fly. At the same time, it clears the flag image memory for the next character.

8.5 The Architecture

The architecture of the ASIC is fairly simple (figure 8.6). It can be decomposed into the following blocks:

- an input interface,
- a circular input buffer to store incoming bytes until they can be sent to the data processing units,
- two nearly identical data processing units for the X and Y coordinates,
- a stack to store intermediate results,
- the sequencer which is based on a small ROM,
- a RAM for the flag image memory,
- the flag set/fill unit,
- an output interface.

8.5.1 The Circular Input Buffer

The circular input buffer is used to store each curve sent by the processor. It can store upto 512 words which corresponds, for instance, to 64 Bezier splines; each spline being described by 4 control points requiring 2 14-bit coordinates each.

8.5.2 The Data Processing Units

The architecture of the data processing units has been carefully optimized for the Bezier subdivision algorithm. The data processing unit (figure 8.7) is nearly identical for the x and y coordinates. It is made up of 7 registers, one adder/subtractor, one adder, four busses, and a condition code tester.

The four registers (R0, R1, R2, R3) in the center of a DPU contain the coordinates of the four control points of a Bezier spline. The results of the average operations are stored in the two outer registers (A0, A1). The seventh register (PL) latches the last value pushed on the stack as a temporary buffer. An average operation is generated by an adder followed by a division by two which has been implemented as a hard wired shift right. The subtractor is used to test a curve to determine whether to keep subdividing it or not. The test logic is the only part of the DPU which differs for the x and y coordinates.

Each register contains 14 bits: 6 integer bits and 8 fractional bits. No extra sign or overflow bits are required since the only operation performed is an average. The 6 integer bits allow for characters up to 64 pixels high. The 8 fractional bits are sufficient in order to keep the required precision.

The bus configuration has been designed to enable parallel access to both adders and to limit the capacity (pF) of each bus. The busses also give access to the stack and the input circular buffer. The stack is used to save curves from one level of subdivision to the next.

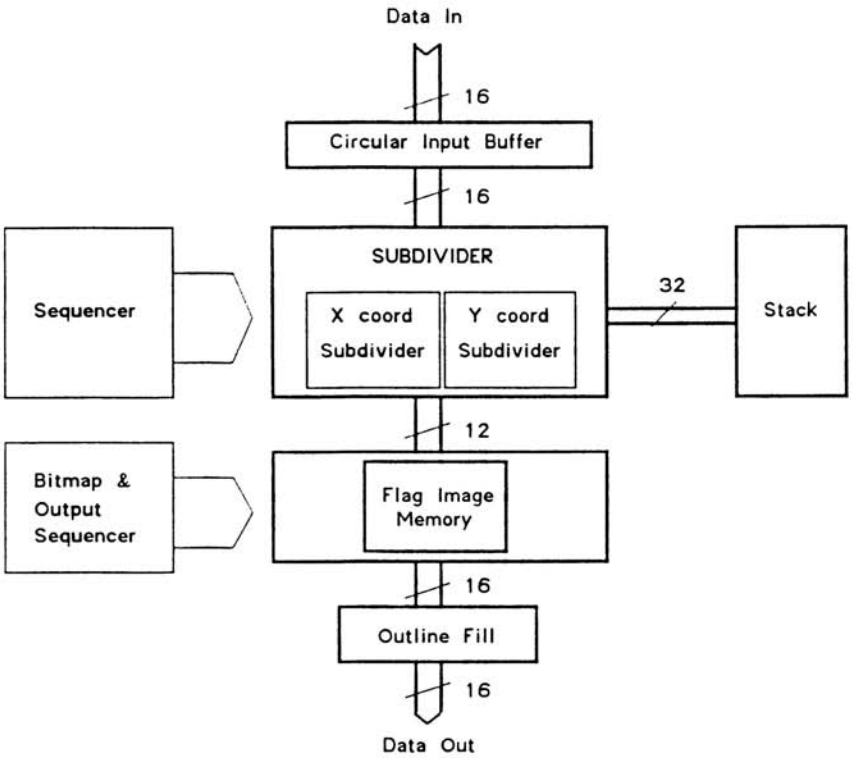

FIGURE 8.6. General architecture of the ASIC

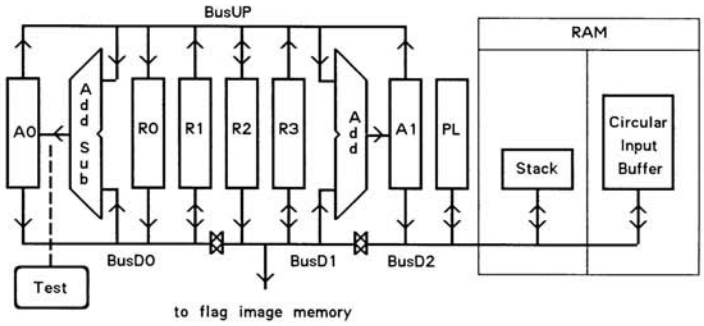

FIGURE 8.7. Architecture of a data processing unit

8.5.3 The Stack

The stack is 16 bits wide to contain sequentially the x and y coordinates of a point and the type of curve it belongs to (Bezier spline or line segment). To calculate the height of the stack, one has to take into account the range of accepted coordinates (0.00 to H'3F.FF in our case) and the chosen value for *MinFracLength* (1/8). In our case, a maximum of 6 + 3 = 9 levels of recursive subdivisions can occur. Since each subdivision requires that 4 control points (4 * 2 coordinates) of a new spline be stored on the stack, the stack has to contain at least 9 * 4 * 2 = 72 words. Note that no new curve is processed before the last one has been completely scan-converted; this keeps memory requirements for the stack to a strict minimum. The stack was assigned 128 words.

8.5.4 The Sequencer

The sequencer is based on a ROM which contains the microcode for the recursive subdivision of Bezier splines and line segments. The ROM is followed by the necessary combinatorial logic to generate the DPU's control signals. It is preceded by the ROM address generator which takes into account the results of the tests and the increment/jump bit from the ROM. The jump addresses are not coded inside the ROM since this would demand too large a ROM or would demand a lot of extra jumps. The algorithm contained in the ROM is represented in figure 8.8.

8.5.5 The Flag Set/Fill Unit

This unit is used to set each flag in a word of the flag image memory. An exclusive OR is used to conserve the parity of flags in complex cases. After all the flags of an outline have been computed, this same unit takes each word of the flag image memory and fills it. This is done at the same time as the processor requests each word. To avoid having to insert wait states during the processor's read cycle, the filling unit uses a 'fill look ahead' scheme (by analogy to the carry look ahead for an addition).

8.5.6 Implementation

During the implementation of the circuit, the size of the RAMs required became an important factor. About half the surface of the circuit is RAM. In an effort to reduce the RAM surface, the stack, the flag image memory and the circular input buffer were grouped into a single RAM. This does not have any effect on the speed of the circuit since no two memories ever had to be accessed at the same time. The result was a denser RAM. Since the size of the final circuit was still too large, it was chosen to use an external RAM for the first prototype. The ASIC is connected to the RAM via a 16 bit data bus and a 10 bit address bus. The RAM contains 256 words for the flag image memory (ie 256*16 bits = 64*64 bits), 512 words for the circular input buffer, 128 words for the stack, and 128 free words which were used for the test of the circuit. In order to further facilitate the debugging, a dual port RAM (DPRAM) was chosen instead of a RAM.

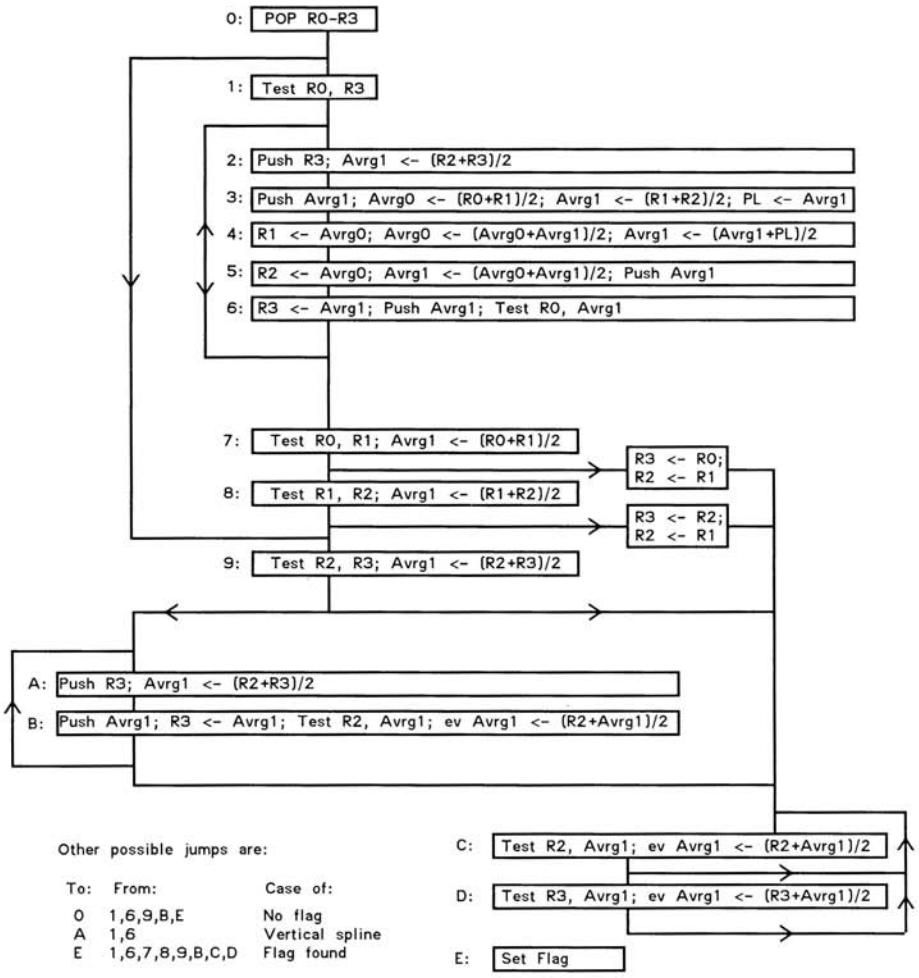

FIGURE 8.8. The organigram

8.5.7 Subdivision of a Bezier spline

Figure 8.9 demonstrates the subdivision of a Bezier spline using the DPU described above. (NB: Notation is the same as in figures 8.4 and 8.7.) The four control points of the spline are loaded either from the stack or from the CIB in four steps. Then, the endpoints of the spline are tested to determine whether or not the spline needs a further subdivision step. The rest of the sequence represents the case where the spline has to be subdivided. At the end, one sub-spline is in the four registers and is tested for a new subdivision. The other sub-spline is on the stack in the same order as the first spline (V0 first, V3 last).

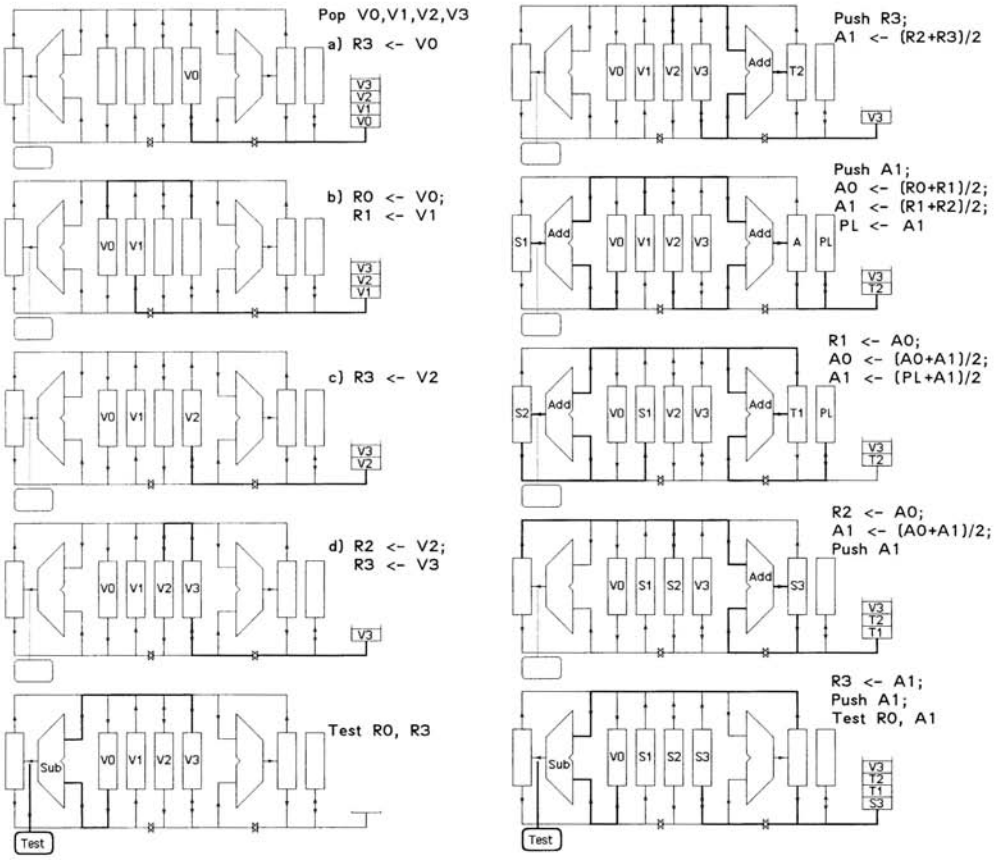

FIGURE 8.9. Sequence of operations for the subdivision of a Bezier spline

8.5.8 The Input/Output Interface

The ASIC has been designed as a coprocessor for a M68020. Its input/output interface can be decomposed into:

- the RAM interface: a 16 bit data bus, a 10 bit address bus, and the usual output enable and Read/Write signals,
- the circular input buffer status flags: CIB_Full, CIB_Half_Full and CIB_Empty,
- the CIB_Size_Increment inputs which update the ASIC's count of the CIB size. There are two signals to increment the count by 4 (Bezier spline with 4 control points), by 2 (line segments with 2 extremities), or by 1 (end of character command word),
- an interrupt request IReq output to indicate that the bitmap is ready to be sent to the processor,
- a bitmap word request input to indicate that the processor is waiting for the next word of the bitmap,
- and a Memory acknowledge output to indicate that the next bitmap word is on the processor's data bus or that wait states have to be inserted in the processor's read cycle.

8.6 Technology

The choice of the technology used for the ASIC was based on ease of conception and former experience. The solution adopted for the first prototype was VTI's 2 μm CMOS (CMN20a) standard cell technology.

8.7 The Results

The prototype circuit was sent to the foundry in May 1991. According to the simulation work which has been done, we can estimate the rate of generation of characters at around 2500 characters per second, for capital character sizes of 32 pixels. This estimation doesn't take into account any phase control application time which might be added by the main processor.

The size of the ASIC is just under 6 by 6 mm.

8.8 Conclusions

This paper has presented the algorithm developed for the rasterization of characters described by their outlines. A comparison between recursive subdivision and forward dif-

ferencing shows that recursive subdivision is easier to implement into an application specific integrated circuit. The proposed architecture guarantees the correct rasterization of outline characters due to precise criteria for stopping recursive subdivision.

A simple architecture is proposed for the implementation of the rasterization algorithm. The simulations show that the ASIC can generate 2500 characters/second. It can be used as the core of a high performance raster image processor.

8.9 References

[1] B.D. Ackland, N.H. Weste, "The Edge Flag Algorithm - A Fill Method for Raster Scan Displays",*IEEE Trans. on Computers*, Vol. 30, No 1, pp. 41–48. (1981).

[2] J.Gonczarowski, "Fast Generation of Unfilled and Filled Outline Characters", *Raster Imaging and Digital Typography*, Cambridge University Press, pp. 97–110. (1989).

[3] R.D.Hersch, "Introduction to Font Rasterization", in Andre, Hersch (eds.), *Raster Imaging and Digital Typography*, Cambridge University Press, pp. 1–13. (1989).

[4] R.D.Hersch, "Efficient Rendering of Outline Characters", *1990 SID Symposium Digest of Technical papers*, Vol. 21, Society for Information Display, pp. 392–394. (1990).

[5] S.L.Lien, M.Schantz, V.Pratt, "Adaptive Forward Differencing for Rendering Curves and Surfaces", *Proceedings SIGGRAPH'87, ACM Computer Graphics*, Vol. 21, No. 4, pp. 111–118. (1987).

[6] S.L.Lien, M.Schantz, R.Rocchetti, "Rendering Cubic Curves and Surfaces with Integer Adaptive Forward Differencing", *Proceedings SIGGRAPH'89, ACM Computer Graphics, Vol. 23*, No. 3, pp. 157–166. (1989).

[7] W.M.Newman, R.F.Sproull, *Principles of Interactive Graphics*, McGraw-Hill. (1979).