Two-level Pipelining of Systolic Array Graphics Engines*

J. A. K. S. Jayasinghe and O. E. Herrmann

Twente University
Laboratory for Network Theory
P.O. Box 217
7500 AE Enschede
The Netherlands

In a systolic array, the maximum operating speed is determined by the most complex operation performed. In a systolic array graphics engine, capable of generating high quality images, one has to perform complex operations at a very high speed. We propose to use pipelined functional units in systolic array graphics engines as they can perform complex operations at high speeds. Due to time-varying discontinuities of operations performed by systolic array graphics engines, introduction of pipelined functional units is a complex problem. In this paper we present a methodology which solves this problem by a graph-theoretic approach. Furthermore, we characterize the architectures which can be improved by pipelined functional units.

Categories and Subject Descriptors:
B.7.1 [Integrated Circuits]: Types and Design Styles – VLSI
C.1.1 [Single Data Stream Architectures]: Pipeline Processors
C.3 [Special-purpose and Application Based Systems]: Real-time Systems
I.3.1 [Computer Graphics]: Hardware Architecture – Raster Display Devices
I.3.7 [Computer Graphics]: Three-dimensional Graphics and Realism – Color, Shading, Shadowing and Texture

Key Words & Phrases: Raster Graphics, Computer Graphics, Real-time Scan-conversion, Systolic Arrays, Two-level Pipelining.

*The work described in this paper is a part of a project for development of a graphics workstation. The project is sponsored by the Dutch Research Foundation (STW), under contract CWI77.1249, carried out by Twente University, Enschede, The Netherlands and Center for Mathematics and Computer Science, Amsterdam, The Netherlands.

1 Introduction

Raster graphics systems have become very popular over the past decade, due to the image quality achieved by rendering the complete object instead of rendering the outline of the object as in vector graphics. Though raster graphics systems are capable of rendering high quality pictures, they need several orders of magnitudes higher processing power for image generation compared to vector graphics systems. As the display must be refreshed continually, almost all raster graphics systems decouple the image generation and display refresh process by an intermediate buffer called *frame buffer*. Though a typical frame buffer needs several Mbytes of memory, cheap memories made the frame buffer concept feasible. The frame buffer is organized as a pixel array for fast screen refresh. Although, the structure of the frame buffer is extremely suitable for screen refresh, its structure is not well suited for fast interaction [12]. In [12], it has been argued that the frame buffer must be replaced by a structured object list tuned for both fast interaction and screen refresh. In this case, pixel values have to be calculated in real-time. Some graphics systems can calculate the pixels from an object description [1],[2],[3],[5],[6], which could be adapted to operate with structured object lists. Systems like Pixel-planes [5], add pixel processing logic to each pixel memory location and systems like PROOF [6] use one processor-per-object. Addition of pixel processing logic to each pixel results in a multi-fold increase in overall size of the system, since the size of the processing logic is several times the area of a memory cell. On the other hand, practical object processors can handle only very simple objects like triangles. Therefore, one needs a very large number of object processors to get a complex image. As these approaches produce very big systems, one can question the feasibility of replacing the conventional frame buffer by a structured object list.

Fortunately, the overall system size of the multi-processor systems reported in [1]-[3], is much smaller than systems based on processor-per-pixel or processor-per-object approaches. These multi-processor systems are based on a one-dimensional systolic array. They can generate the images at the display refresh speed without a frame buffer. In this paper we refer to these graphics engines as SAG (*Systolic Array Graphics*) engines. In SAG engines, pixel storage has been limited to a row of pixels, which has been distributed over the processor array. In general, a systolic array consists of an array of identical processor elements (PEs), and connections between PEs have been restricted only to near neighbors. Hence, systolic arrays are very attractive for VLSI implementation and furthermore, communication between PEs is fast and cheap.

When all the PEs are finished with their computations, a global clock enables the communication between PEs. Therefore, throughput of the systolic array is determined by the delay of the most complex operation supported. Hence, we can expect a reduction in maximum operating speed when complex operations are added to an SAG engine to provide better image quality. The maximum operating speed of the SAG engines reported in [2],[3], have been limited to 40-50ns due to (16 bit fixed-point) addition, even though advance IC technologies like 1.2μm CMOS have been used. A 50ns pixel clock cycle is enough for a 512x512 pixels display refreshed at a rate of 50Hz. A 1024x1024 pixels display refreshed at the same rate needs a 12ns pixel clock cycle. Computer graphics users have ever increasing desire for higher resolution displays, better image quality, and higher interaction speed. Therefore, SAG engines will not be feasible for tomorrow's computer graphics, unless one has some means to speed up the operations performed in these graphics engine.

In the past, speed of the ICs have been increased dramatically by scaling the device sizes. Early integrated circuits (circa 1965) had minimum feature sizes in the range of 10 to $20\mu m$. Today, sub-micron minimum feature sizes are used in the state-of-the-art IC technology. When the parasitic wiring capacitance is neglected, scaling of minimum feature sizes by factor S reduces the delay by factor S^2 (theoretically) [7]. But the feature sizes of current ICs are such that speed improvements due to scaling are lower than the above theoretical value due to wiring capacitances and other parasitic effects. The parasitic effects become more and more dominant when the minimum feature sizes become smaller and smaller. However, there is a fundamental physical limit on the transistor sizes, and according to [7] this is of 0.25 μm order. Based on the above facts (speed of scaled IC technologies, speed requirements of higher resolution displays and speed of reported SAG engines) one can conclude that bare speed-up of scaled IC technology is not enough for tomorrow's computer graphics hardware based on SAG engines.

Pipelined functional units have been used in computer hardware to achieve higher throughputs. A systolic array built from pipelined functional units has been first reported in 1982 [11], which has been referred to as *two-level pipelined systolic array.* Hence, in this paper we propose to use pipelined functional units in SAG engines to increase the operating speeds. The introduction of pipelined functional units into an SAG engine is a complex problem, as functions performed by PEs in an SAG engine are subject to time-varying discontinuities. (This is explained in detail in the last paragraph in Section 2.) But, as we can see in this paper, the complexity can be handled by a systematic approach.

The primary goal of this paper is to make a framework to derive two-level pipelined SAG engines from a non (two-level) pipelined SAG engine. In the next section, we present a survey of working principles of SAG engines reported in literature. In Section 3, a formal approach is presented, which can be used to derive two-level pipelined SAG engines from a non (two-level) pipelined SAG engine. In Section 4, some issues on pipelinability are discussed. An architecture of a two-level pipelined SAG engine is presented in Section 5. Finally some conclusions are drawn in Section 6.

2 Principles of SAG Engines

The SAG engines are built from a one-dimensional identical processor array as shown in Figure 1. According to the directions of data movements, we can categorize the SAG engines into two groups which will be referred to as *contra-flow* and *contra-flow free* architectures. In contra-flow architectures, video information travels in the opposite direction to data and instruction flow. Video, data and instructions travel at the speed of one processor per cycle. The difference in contra-flow free architectures is that video information is sent to the display through a global bus. The architectures presented in [1],[2] are contra-flow architectures whereas the architectures presented in [3],[4] are contra-flow free architectures.

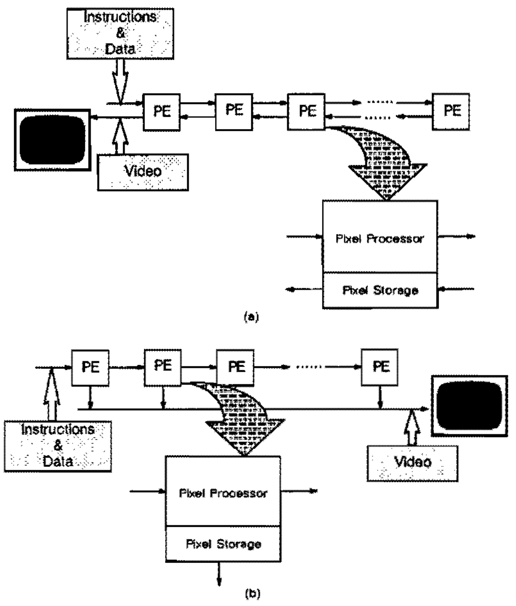


Figure 1: Two Different SAG Engins: (a) Contra-flow, (b) Contra-flow free.

In contra-flow architectures only half of the processors contain the instructions at any cycle. If processors at even locations contain the instructions on cycle N, then processors at odd locations idle on cycle N because only the processors containing instructions are active. As instructions travel one processor per cycle, the states are reversed on the next cycle. However, in contra-flow free architectures, all the processors are active at any cycle. Apart from these differences, contra-flow and contra-flow free architectures have the following common behavior: each processor array is fed by instructions, which contain the information of a span of pixels, along with some processor addresses indicating some important points in the span. Different functions are performed by sending different instructions. For example, constant shading or Gouraud shading is done by sending different instructions. When the instructions pass through each processor, the pixel values are calculated by the processors when they are addressed by the address field and stored in a local pixel storage. Furthermore, when the processor is addressed by the instruction, some data associated with each instruction are also modified before the instruction is sent to the next processor.

Due to real-time requirements, the processors in the array work in two modes: *refreshing and non-refreshing*, which is determined by the instruction on each processor. At any time only one processor can be in refreshing mode. When a processor is in the refreshing mode, the content of the pixel storage is sent to the display. Non-refreshing processors do the manipulations on the data stream, if they are addressed by the address field of the incoming data. Furthermore, the non-refreshing processors on the right of the refreshing processor operate on the current pixel row, while the non-refreshing processors on the left of the refreshing processor operate on the next pixel row. As instructions travel one processor per cycle, in each cycle the processor in refreshing mode switches back to non-refreshing mode while its right neighbor becomes the next refreshing processor. Therefore, SAG engines process the pixel information in a pipelined fashion without any exclusive pipeline filling or draining. Table 1 shows the instructions supported by different SAG engines reported in literature.

The functions performed by PEs in an SAG engine are subject to time-varying discontinuities. For example when the Z-buffer algorithm is executed on an SAG engine, the Z value of the polygon closest to the eye so far encountered is stored in a register for a variable number of clock cycles. The number of clock cycles depends on conditions such as whether the next instruction has a pixel span which overlaps with the pixel location assigned for the processor, or whether the new pixel span is closer than the previous spans. As the instructions for the Z-buffer algorithm pass through each PE, the Z value associated with each instruction is updated only by those processors that are assigned to a given pixel span, which introduces another form of functional discontinuity. Note that in a systolic array designed for convolution, matrix multiplication, etc., no functional discontinuities can be seen as all the PEs perform the same function. When pipelined functional units are introduced, one must make sure that no empty cycles, no exclusive pipeline filling or draining cycles are introduced to handle the functional discontinuities, because they reduce the number of advantages we have in an SAG engine.

3 Designing Two-level Pipelined SAG Engines

3.1 Related Work and Their Strengths and Weaknesses for Two-level Pipelining of SAG Engines

Designing a two-level pipelined systolic array can be done in two ways: transforming a non (two-level) pipelined design systematically into a two-level pipelined design, or designing the two-level pipelined systolic array from scratch. The later approach has been demonstrated in literature [10] in applications such as computing inner products. To our opinion this approach is not well suited for SAG engines due to the complexity introduced by time-varying functional discontinuities. The former approach was first introduced by Kung et al. [8]. In fact, in two-level pipelining, one tries to improve the clock period by introducing pipeline registers into critical paths. A technique called *re-timing* which could optimize the clock speed of a synchronous circuit was introduced by Leiserson et al.[9]. In re-timing, the clock speed is optimized by re-locating the registers in a given circuit while the logical behavior of the system is kept intact. Graph-theoretic approaches have been used for re-timing and two-level pipelining by Leiserson et al. and Kung et al. In these graph-theoretic approaches, combinational logic and communication links are denoted by the nodes and edges of a graph. The number of registers on a communication link is denoted by a weight on the corresponding edge. The weights on nodes represent the propagation delay of the corresponding combinational logic, which has been omitted by Kung et al. In both approaches, registers introduce fixed latencies. In an SAG engine, the latencies introduced by registers are subject to time-varying discontinuities. Therefore, both approaches cannot be used to improve the clock speed of an SAG engine.

In Reference	Instruction	Description
1	REFRESH( )	Send the local pixel storage to the display and reset the processor.
	EVAL0(X,DX,I)	Store value I in processors X,X+1,...,X+DX . (Constant shading)
2,3	REFRESH( )	Send the local pixel storage to the display and reset the processor.
	EVAL0(X,DX,I,Z,DX)	Calculate the intensity and depth by zero and first order interpolation at processors X,X+1,..X+DX and store the calculated intensity and depth if stored depth is larger than calculated depth. (Constant shading and hidden surface removal by the Z-buffer algorithm, supported only in 3.)
	EVAL1(X,DX,I,DI,Z,DX)	Calculate the intensity and depth by first order interpolation at processors X,X+1,..X+DX and store the calculated intensity and depth if stored depth is larger than calculated depth. (Gouraud shading and hidden surface removal by the Z-buffer algorithm.)
4	REFRESH( )	Send the local pixel storage to the display and reset the processor.
	EVAL0(X,DX,I)	Accumulate value I in processors X,X+1,...,X+DX until next refresh. (Constant shading)
	EVAL1(X,DX,I,DI)	Calculate the intensity by first order interpolation at processors X,X+1,...,X+DX and accumulate the calculated intensity until next refresh. (Gouraud shading)
	EVAL2(X,DX,I,DI,DDI)	Calculate the intensity by second order interpolation and accumulate the calculated intensity until next refresh. (for Phong Shading)
	SETI(X,DX,I)	Change the intensity at processor locations X,X+DX,X+2DX,... to I during next interpolation.
	SETDI(X,DX,DI)	Change the first derivative of the intensity at processor locations X,X+DX,X+2DX,..., to DI during next interpolation.
	SETDDI(X,DX,DDI)	Change the second derivative of the intensity at processor locations X,X+DX,X+2DX,... to DDI during next interpolation.
	DIS(X,DX)	Disable the accumulation between processor locations X and DX.
	NOP( )	Do nothing

Table 1: Instructions Supported in Different SAG Engines.

3.2 A New Methodology

In this section, we present a new graph-theoretic approach which can be used to convert a systolic array (built from non-pipelined functional units) with time-varying functional discontinuities into a two-level pipelined version. In fact this approach can be used not only for SAG engines but also for any systolic array with or without functional discontinuities.

In our methodology, the original systolic array is represented at bit-level by a finite, vertex-weighted, edge-weighted, directed graph $G = \langle V, V', E, d_V, d_{V'}, w \rangle$, (from now on, for simplicity we say *graph.*) where V, V' and E denote the nodes and edges of the graph. The labels $d_V, d_{V'}$ and w denote the weights on the nodes and edges respectively. The graph G which is constructed by replicating the graph of a PE, $G_{PE} = \langle V_{PE}, V'_{PE}, E_{PE}, d_{V_{PE}}, d_{V'_{PE}}, w_{PE} \rangle$ and connecting vertices according to communication between PEs as the entire array is built from identical PEs.

In order to construct the graph G_{PE}, the bit-level functional units are represented by vertices V_{PE} and bit-level storage by vertices V'_{PE}. The communication between nodes are denoted by the edges and for each edge $e \in E_{PE}$, $w_{PE}(e)$ denotes the earliest communication time slot for all legal combinations of instructions. The edges which communicate in the $i^{th}(i = 0, 1, 2, ...)$ time slot are weighted by i. Each vertex $v \in V_{PE}$ is weighted by the numerical propagation delay of the functional unit $d(v)$. Each vertex $v' \in V'_{PE}$ is multiply weighted by $d(v'_{a \to b})$ for each input edge a and output edge b connected to node v', when there is a direct information flow[1]. The quantity $d(v'_{a \to b})$ indicates the minimum latency (which is under control of instructions) between information flow from edge a to b through the storage node. If there is no direct information flow from edge a to b then $d(v'_{a \to b})$ is undefined.

The maximum clock speed of the circuit is determined by the propagation delays of the critical paths. A critical path in our graph G can be identified as a directed path activated in the same time slot such that the sum of the node weights on that path is maximum over the entire graph. (If a critical path goes through a storage node, the critical path is terminated at the storage node whenever the vertex weight corresponding to that path is non-zero, as the vertex weights of storage nodes denote the latency whereas the weights of other nodes denote the propagation delay.) In order to improve the clock speed the graph is re-timed. In re-timing, pipeline registers are added to critical paths to meet the given speed requirements, and then some more additional pipeline registers are added to some other edges and/or latencies at some storage vertices are changed such that the following theorem is satisfied.

Two-level Pipelining Theorem: If a two-level pipelined design is obtained by adding pipeline registers to some edges and/or changing the latencies of storage nodes, the logical behavior of the system will be kept intact if the differences in latencies through all pairs of paths between any two nodes are equal in the original and re-timed graphs.

[1]In Figure 5 the information flow from edge a_i to a_{i+1} is direct whereas the information flow from edge a_i to d_i is indirect. There is no information flow from edge a_i to b_{i+1}

Proof: Let m, n be two nodes and p_1, p_2 be two paths from m to n. Let path p_1 be activated on $i_{p1_1}, i_{p1_2}, \dots, i_{p1_{N1}}$ time slots ($i_{p1_1} < i_{p1_2} \dots < i_{p1_{N1}}$) and p_2 be activated on $i_{p2_1}, i_{p2_2}, \dots, i_{p2_{N2}}$ time slots ($i_{p2_1} < i_{p2_2} \dots < i_{p2_{N2}}$). If we introduce a latency of k cycles in path p_1, by introducing pipeline stages and/or changing storage latencies, node m gets the data from node n through path p_1 in time slot $i_{p1_{N1}} + k$. In the original graph node m gets the data from node n through paths p_1 and p_2 in time slots $i_{p1_{N1}}$ and $i_{p2_{N2}}$ respectively. In order to keep the logical behavior of the graph intact we must get data through path p_2 in time slot $i_{p2_{N2}} + k$. Now, we can see the difference in latency through path p_1 and p_2 is equal to $(i_{p1_{N1}} - i_{p2_{N2}})$ cycles in both original and re-timed graphs. This argument can be applied to all paths between any node pair to prove the theorem. □

It can be proven that the application of the two-level pipelining theorem produces $|E| - |V| + 1$ linearly independent equations, where $|E|, |V|$ are the number of edges and vertices in the graph [13]. These equations contain $|E| + |V'_{\geq 1}|$ unknowns where $|V'_{\geq 1}|$ is the number of non-zero weights in the storage nodes. As we have more unknowns than equations, the re-timing problem can be solved by linear programming, for example minimizing the total register count as Leiserson proposed [9].

3.2.1 An Example

Figure 2(a) shows a systolic array finite impulse response (FIR) filter[2] whose output is given by $y_i = \sum_{k=0}^{k=1} x_{i-k}$. In this filter input x_i and output y_i travel to the left and right respectively at a speed of one processor per cycle. As y_i travels, it collects the partial sums and hence the complete sum is formed at the rightmost PE. For the proper operation, the input x_i must supplied at every other cycle. Hence, output y_i is also produced at every other cycle. Figure 2(b) shows the graph of the systolic FIR filter when a carry ripple adder is used for addition. For simplicity, we have assumed that calculations are performed by 2 bit binary numbers. The nodes v'_1, v'_2 (v'_3, v'_4) represent the delay D which stores x_i in PE1 (PE2). Similarly, nodes v'_{11}, v'_{12} (v'_7, v'_8) represent the delay D which stores y_i in PE1 (PE2). The adder in PE1 (PE2) is represented by nodes v_9, v_{10} (v_5, v_6).

Let us assume that $k_{i,j}$ denotes the number of pipeline registers placed on edge (v_i^*, v_j^*) [3] and k_i denotes the non-zero latency at storage node v'_i for a two-level pipelined design. Let us place a single pipeline register in each edge $(v_9, v_{10}), (v_5, v_6)$ as these edges belong to the critical paths, i.e. $k_{9,10} = k_{5,6} = 1$. Note that these edges represent the carry propagation in the adders. Application of the two-level pipelining theorem generates 5 linearly independent equations as follows.

[2]In fact this systolic array has no functional discontinuities. But we use this example for simplicity.

[3]Here v_i^* stands for v_i or v'_j, as nodes in the example are numbered such that v_i and v'_j exists only for $i \neq j$.

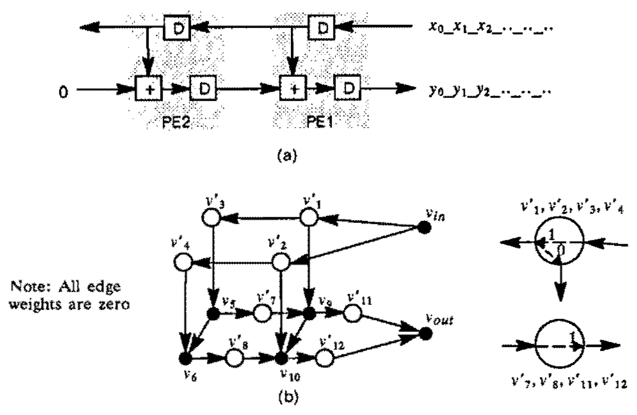


Figure 2: A Systolic FIR Filter (a) Non (two-level) Pipelined Implementation, (b) Graph for (a).

$$\begin{aligned}
k_{in,1} + k_1 + k_{1,3} + k_3 + k_{3,5} + k_{5,6} - k_{4,6} - k_4 - k_{2,4} - k_2 - k_{in,2} &= 0 \\
k_{1,3} + k_3 + k_{3,5} + k_{5,7} + k_7 + k_{7,9} - k_{1,9} &= 2 \\
k_{2,4} + k_4 + k_{4,6} + k_{6,8} + k_8 + k_{8,10} - k_{2,10} &= 2 \\
k_{5,6} + k_{6,8} + k_8 + k_{8,10} - k_{9,10} - k_{7,9} - k_7 - k_{5,7} &= 0 \\
k_{9,10} + k_{10,12} + k_{12} + k_{12,out} - k_{11,out} - k_{11} - k_{9,11} &= 0
\end{aligned}$$

These equations have been obtained by taking the difference of latency through two paths between node pairs $(v_{in}, v_6), (v_1, v_9), (v_2, v_{10}), (v_5, v_{10})$ and (v_9, v_{out}) respectively. Any other equation obtained by other paths can be expressed as a linear combination of the above 5 equations. Now,

$$\begin{aligned}
k_i &= 1, \quad \text{for } i = 1,2,3,4,7,8,11,12 \\
k_{i,j} &= \begin{cases} 1 & \text{for } (i,j) = (v_{in}, v_2), (v_{11}, v_{vout}), (v_9, v_{10}), (v_5, v_6) \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

is a feasible solution. This means we have to place pipeline registers to the carry path of each adder and edges $(v_{in}, v_2), (v_{11}, v_{out})$ and keep the latencies of the storage nodes unchanged to get a two-level pipelined design for the systolic FIR filter. This pipelined design can be operated twice as fast as the original design. In general, such a pipelined design can be made to operate m times faster than the original design when an m bit adder is used. The area overhead will be insignificant compared to speed improvements as the pipeline registers need very small silicon area.

4 A Study on Pipelinability

In this section, we study the pipelinability of SAG engines reported in literature [1], [2], [3], [4]. Due to regularity of the graph, only part of the graph will be drawn for the

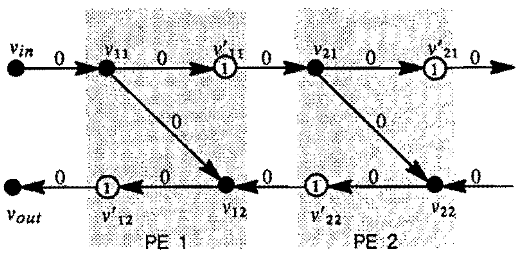


Figure 3: A Partial Graph for Contra-flow Architecture.

discussions in the next sub-sections. Furthermore, we simply omit unnecessary vertex weights.

4.1 Contra-flow vs. Contra-flow Free Architectures

First, let us study the contra-flow architecture given in [1] where the pixel stream travels in opposite direction to instruction and data streams. The relevant partial graph is depicted in Figure 3, where nodes $v_{11}, v_{12}, v'_{11}, v'_{12}$ are nodes in the 1st PE and $v_{21}, v_{22}, v'_{21}, v'_{22}$ are nodes in the 2nd PE. Let $v_{11}, v'_{11}, v_{21}, v'_{21}$ and $v'_{12}, v_{12}, v'_{22}, v_{22}$ be on the instruction and pixel streams respectively. Nodes v_{in} and v_{out} represent the instruction source and display respectively. Let us consider the latencies between nodes v_{11} and v_{12}. Latencies through paths $v_{11}v_{12}$ and $v_{11}v'_{11}v_{21}v_{22}v'_{22}v_{12}$ are 0 and 2 cycles respectively. Now let us assume k_1, k_1, k_2 and k_2 pipeline registers are placed between (v_{11}, v'_{11}), (v_{21}, v'_{21}), (v_{12}, v'_{12}) and (v_{22}, v'_{22}) to get a regular design[4]. Now latencies through paths $v_{11}v_{12}$ and $v_{11}v'_{11}v_{21}v_{22}v'_{22}v_{12}$ are 0 and $k_1 + k_2 + 2$ cycles respectively. Therefore in order to satisfy the two-level pipelining theorem, $k_1 + k_2 + 2 = 2$. When $k_1 \geq 1$ we get an unrealizable design. Hence we conclude that an SAG engine having a contra flow architecture is not two-level pipelinable[5].

Now let us turn our attention to the contra-flow free architecture presented in [3]. The partial graph is depicted in Figure 4, where nodes v_{11}, v_{12}, v'_{11} are nodes in the 1st PE and v_{21}, v_{22}, v'_{21} are nodes in the 2nd PE. Let $v_{11}, v'_{11}, v_{21}, v'_{21}$ and v_{12}, v_{22} be on the instruction and pixel streams respectively. Nodes v_{in} and v_{out} represent the instruction source and display respectively. Let us consider the latencies between nodes v_{11} and v_{22}. Latencies through paths $v_{11}v_{12}v_{22}$ and $v_{11}v'_{11}v_{21}v_{22}$ are 0 and 1 cycles respectively. Now let us assume k_1, k_1, k_2 and k_2 pipeline registers are placed between (v_{11}, v'_{11}), (v_{21}, v'_{21}), (v_{12}, v_{22}) and (v_{22}, v_{out}) to get a regular design. Now latencies through paths $v_{11}v_{12}v_{22}$ and $v_{11}v'_{11}v_{21}v_{22}$ are k_2 and $k_1 + 1$ cycles respectively. Now, in order to satisfy the two-level pipelining theorem we must have $k_1 - k_2 + 1 = 1$. As the condition $k_1 - k_2 + 1 = 1$ can be satisfied, we can conclude that the direction of the information flow in the contra-flow free architecture does not impose any constrains for two-level pipelining. But we can

[4]In the next section, the necessity for pipeline registers in the instruction streams will become clear.

[5]Note that the systolic FIR filter in our example also has a contra flow architecture, but we were able to design a two-level pipelined systolic FIR filter.

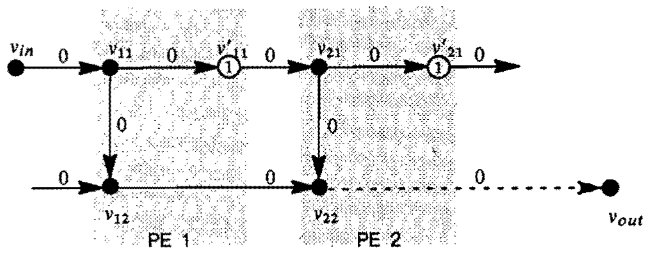


Figure 4: A Partial Graph for Contra-flow Free Architecture.

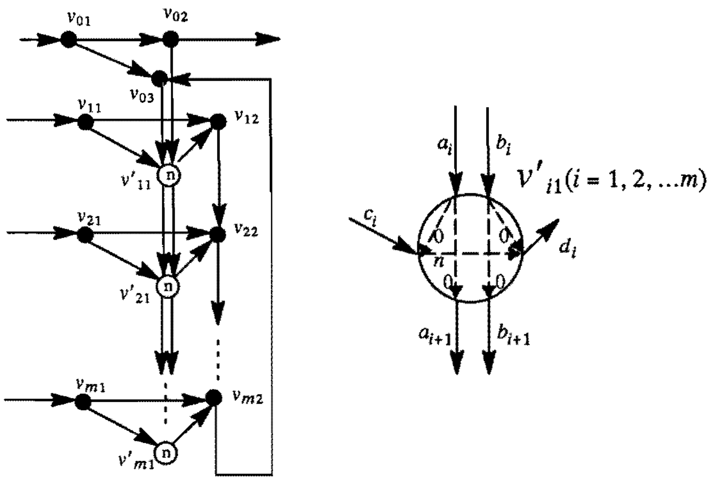


Figure 5: A Partial Graph for a PE Capable of Hidden Surface Removal.

not guarantee that contra-flow free architectures as two-level pipelinable, because in this study we have neglected the internal details of a PE to a great extent. Hence, we classify the contra-flow free SAG engines as potential designs for two-level pipelining.

4.2 Different Instruction Sets

In this sub-section let us study the impact of different instruction sets. First, let us consider the instruction set proposed in [3]. This instruction set is capable of Gouraud shading with hidden surface removal by the Z-buffer algorithm. Let $v_{01}, v_{02}, ..., v_{m2}$ be nodes on a PE, see Figure 5, such that nodes $v_{12}, v_{22}, ..., v_{m2}$ make the comparison between the current Z value and stored Z value for hidden surface removal. Nodes $v'_{11}, v'_{21}, ..., v'_{m1}$ store the Z value to be used with the next instruction according to the results of the comparison. Nodes v_{01}, v_{02}, v_{03} are on the instruction stream such that v_{02} and v_{03} generate the write and read signals for the depth storage. The graph has been drawn by assuming that the depth is calculated by m bit fixed-point numbers and the instructions for hidden surface removal are issued at least n cycles apart as done in [3]. Note that, in this graph there are several loops passing through the critical path. (We have assumed that the critical path lies on the comparison circuit). Application of the two-level pipelining theorem reveals that pipeline registers can be inserted into the critical path if we reduce the latencies of the nodes $v'_{11}, v'_{21}, ..., v'_{m1}$. The reductions in storage latencies are necessary due to directed loops such as $v'_{11}v_{21}v_{22}...v_{m2}v_{03}$. As the nodes $v'_{11}, v'_{21}, ..., v'_{m1}$ represent storage logic in the design[6], the comparator unit can only be pipelined into maximum depth of $n-1$. In general $m \gg n$ ($m = 32$, $n = 4$ for the design in [3]) and hence the speed improvement is limited. When $n = 1$, the comparator can not be two-level pipelined.

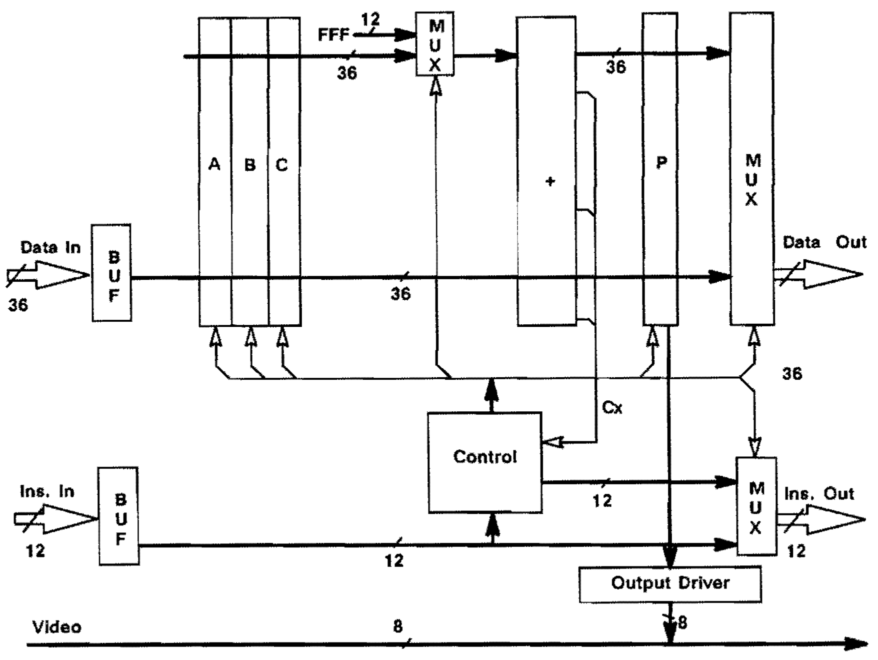
The instruction set presented in [4] has been proven to be two-level pipelinable to any depth. Due to the complexity of the graph the proof is omited and in the next section we will present a two-level pipelined architecture.

5 A Two-level Pipelined SAG Engine

First let us study a non two-level pipelined SAG engine capable of executing the instruction set given in [4] (see Table 1). Figure 6 shows a non-pipelined architecture of a PE which can execute the above instruction set. The registers A,B and C are used to store intensity, its first derivative and its second derivative for interpolations. The register P holds the accumulated intensity (i.e. local pixel storage), which is sent to the output driver and then cleared by the 'REF' instruction. The 'SETI', 'SETDI', 'SETDDI' and 'DIS' instructions disable the registers from being over-written by the next 'EVAL*' instruction. The multiplexers at the right side are used to handle some functional discontinuities. For example, when the PE is (not) within X and X+DX, interpolation is (not) performed. Hence, the right multiplexer on the data stream selects the correct data to the output according to the location of the PE. Furthermore, at some PE locations the state of the instruction is changed. For example, at PE locations X and X+DX the 'EVAL*' instructions are activated and converted into a 'NOP' instruction respectively. This is achieved by the multiplexer on the instruction stream. The discontinuities of the operations performed by each PE depend on the value of X, DX and instruction type. Therefore, the control signals for the multiplexers are generated in conjunction with the carry signals (Cx in Figure 6) from the adder and the instruction in the processor.

The target architecture in Figure 6 can be converted into two-level pipelined design as outlined below. First the graph of the architecture is constructed. Then the critical paths are identified. In our target architecture, the critical paths pass through the carry ripple path in the adder. The places where the pipeline registers must be placed in the critical paths are then decided to meet the given speed requirements. Then locations of other pipeline registers are determined by solving the linearly independent equations obtained by the application of the two-level pipelining theorem. The graph of the architecture is not given due to its complexity and hence more details of the steps we outlined above are

[6]The storage nodes must store data at least for one cycle, otherwise we convert the synchronous design into an asynchronous design.

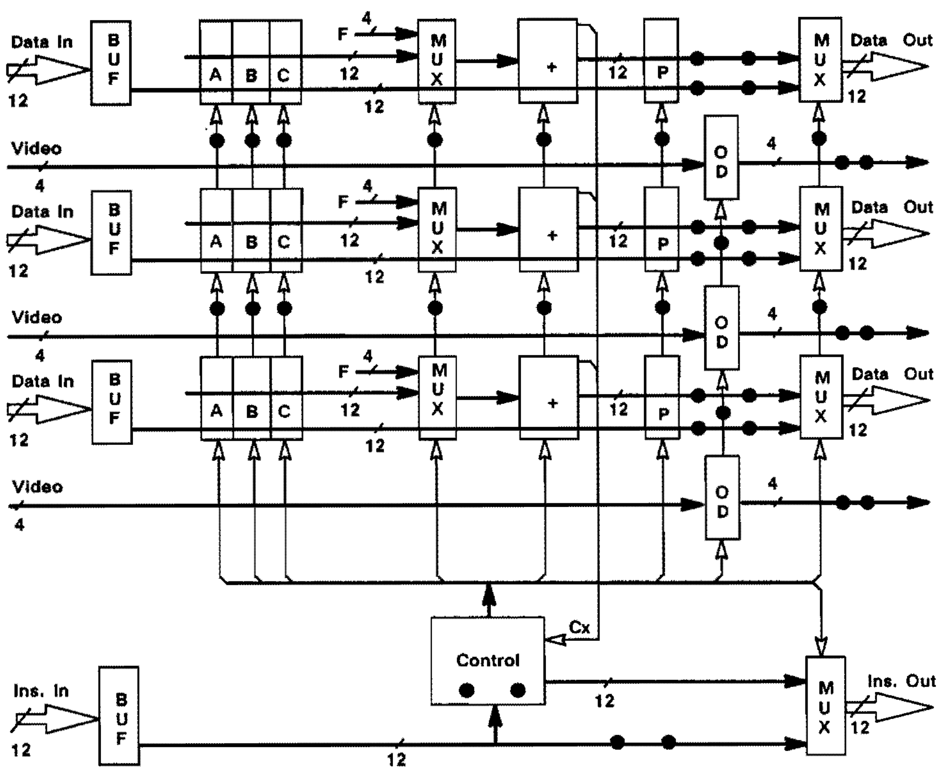


A,B,C	: 36 Bit Internal Registers	MUX	: Multiplexer
P	: 24 Bit Pixel Storage	BUF	: Buffer

Figure 6: Architecture of a Non (two-level) Pipelined SAG Engine for the Instruction Set in [4].

also omited. A two-level pipelined architecture derived by this approach is depicted in Figure 7. Though the architecture in Figure 7 is pipelined to depth 2, it can be pipelined to the bit-level.

Some explanations are given in this paragraph to understand the operation of the pipelined engine. As the critical paths of this circuit lie through the adder, 2 pipeline registers have been placed on the carry path. As control signals for the multiplexers on the right side are generated in conjunction with the carry signals from the adder, the (data) inputs to these multiplexers are delayed by pipeline registers, which can be seen in Figure 7. As the carry signal from the low order bits to higher order bits travels in different time slots, the input data to the adder must also be delayed and hence pipeline registers are placed on the control signals of registers A,B,C,D and multiplexer in the middle. Furthermore, the latency of the carry introduces latency in the adder output. Due to this reason, pipeline registers are introduced on the right hand multiplexer control signals. Pipeline registers on the pixel stream and on the control signals of the output driver are also placed due to similar reasons.



A,B,C	: 36 Bit Internal Registers	MUX	: Multiplexer
P	: 24 Bit Pixel Storage	BUF	: Buffer
OD	: Output Driver	●	: Pipeline Register

Figure 7: A Two-level Pipelined Architecture.

6 Conclusions

The operating speed of the systolic array is determined by the most complex operation performed. Hence the maximum operating speed of an SAG engine tends to reduce as more complex operations are introduced to improve the image quality. Furthermore, high resolution displays also demand high speed operation. In order to improve the speed of SAG engines, the use of pipelined functional units were proposed. Even though pipelined functional units need more hardware for pipeline registers, we think two-level pipelining of SAG engines will be very attractive for tomorrow's computer graphics hardware due to the following major reasons. First, SAG engines generate the pixels in real-time from an object representation and hence one can improve the interaction speed by replacing the conventional frame buffer by a structured object list as proposed in [12]. The SAG engine based systems are more compact than systems based on processor-per-pixel or processor-per-object providing the same functionality, image quality and interaction speed. As the scaling of minimum feature sizes and wafer scale integration increase the device count dramatically, one has very feasible means to provide the additional hardware for two-level pipelining of SAG engines.

A graph-theoretic methodology was presented to transform SAG engines built from non-pipelined functional units into two-level pipelined designs. In an SAG engine, the number of PEs could be large ($256 \sim 4000$) and hence the number of nodes and edges in the graph could be very large ($10,000 \sim 100,000$). Hence, the execution time of the re-timing procedure on a computer will also be very large. Currently we are looking into fast implementations of re-timing by exploiting the regularities in the graph.

Acknowledgments

Thanks to A.A.M. Kuijk and E. Blake at Interactive Systems Group, CWI, Amsterdam for helpful discussions. Thanks also to F. Moelart El-Hadidy, G. Karagiannis and S. Gerez at Laboratory of Network Theory, Twente University, Enschede for proof reading and helpful discussions.

References

[1] Nader Gharachorloo and Christopher Pottle, *SUPER BUFFER: A Systolic VLSI Graphics Engine for Real Time Raster Image Generation*, Proceedings of 1985 Chapel Hill Conference on VLSI, Computer Science Press, 1985, pp. 285-305.

[2] Nader Gharachorloo, Satish Gupta, Erdem Hokenek, Peruvemba Balasubramanian, Bill Bogholtz, Christian Mathieu and Christos Zoulas, *Subnanosecond Pixel Rendering with Million Transistor Chips,* Proceedings of SIGGRAPH 88, August 1988, pp. 41-49.

[3] Teiji Nishizawa, Takeru Ohgi, Kazuyasu Nagatomi, Hiroshi Kamiyama and Kiyashi Maenobu, *A Hidden Surface Processor for 3-Dimension Graphics*, ISSCC 88, February 1988, pp. 166-167.

[4] J.A.K.S. Jayasinghe, A.A.M. Kuijk and L. Spaanenburg, *A Display Controller for a Structured Frame Store System*, in Advances in Computer Graphics Hardware III, Springer-Verlag, 1989.

[5] H. Fuchs and J. Poulton *Pixel-planes: A VLSI Oriented Design for a Raster Graphics Engine*, VLSI Design, Vol. 3, No. 3, 1981, pp. 20-28.

[6] Bengt-Olaf Schneider, *PROOF: An architecture for rendering in Object Space*, in Advances in Computer Graphics Hardware III, Springer-Verlag, 1989.

[7] D.A. Hodges and H.G. Jackson, *Analysis and Design of Digital Integrated Circuits*, McGraw-Hill, Inc. 1983.

[8] H.T. Kung and M.S. Lam, *Fault-Tolerance and Two-level Pipelining in VLSI Systolic Arrays*, 1984 Conference on Advance Research in VLSI, M.I.T., 1984, pp. 74-83.

[9] C. E. Leiserson and J.B. Saxe, *Optimizing Synchronous Systems*, Journal of VLSI and Computer Systems, Vol.1, No.1, 1983, pp. 41-68.

[10] S.Y. Kung, *VLSI Array Processors*, Prentice Hall, 1988.

[11] H.T. Kung, L.M. Ruane and D.W.L. Yen, *Two-level Pipelined Systolic Array for Multi-dimensional Convolution*, Image and Vision Computing, Vol.1, No. 1, February 1983, pp. 30-36.

[12] P.J.W. ten Hagen, A.A.M. Kuijk and T. Triekens, *Display Architecture for VLSI-based Graphics Workstation*, in Advances in Computer Graphics Hardware I, Springer-Verlag, 1987.

[13] J.A.K.S. Jaysinghe, *Speed Optimization of VLSI/WSI Array Processors*, Internal report under preparation.