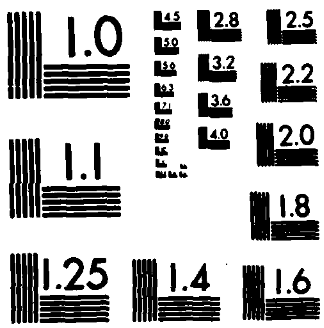
AD-A149 051 ARRAY PROCESSOR UTILISATION IN THE COMPUTATION OF REAL-TIME IMAGES(U) AERONAUTICAL RESEARCH LABS MELBOURNE (AUSTRALIA) L N LESTER AUG 84 ARL-SYS-TM-73 1/1

UNCLASSIFIED F/G 17/7 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNCLASSIFIED

ARL-SYS-TM-73

AR-003-953

DEPARTMENT OF DEFENCE

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION

AERONAUTICAL RESEARCH LABORATORIES

MELBOURNE, VICTORIA

Systems Technical Memorandum 73

ARRAY PROCESSOR UTILISATION IN THE COMPUTATION OF REAL-TIME IMAGES

L.N. LESTER

Approved for public release.

84 12 31 036

AUGUST 1984

COPY No

UNCLASSIFIED

AR-003-953

DEPARTMENT OF DEFENCE

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION

AERONAUTICAL RESEARCH LABORATORIES

Systems Technical Memorandum 73

ARRAY PROCESSOR UTILISATION IN THE COMPUTATION OF REAL-TIME IMAGES

L.N. LESTER

SUMMARY

The application of an array processor to the real time generation of aircraft images in Air Traffic Control tower simulation is described. Particular emphasis is placed upon the problems which arise in achieving efficient utilisation of the array processor. It is shown that the time required to transfer data from the host to the array processor is a serious consideration in the partitioning of the algorithm between the two processors. It is also shown that array processor execution speed is about ten times faster than a host Fortran routine, provided that the array processor code is written to maximise the number of parallel computing elements used and to minimise loop lengths by pipelining of operations. These requirements result in a considerable software development effort. A timing analysis of the resultant array processor code is also presented.

POSTAL ADDRESS: Director, Aeronautical Research Laboratories,
P.O. Box 4331, Melbourne, Victoria, 3001, Australia.

CONTENTS

A-1

1. INTRODUCTION

Array processors have potentially very high computing power, extending well beyond the capabilities of conventional computers. However, problems arise in efficiently managing the combination of host and array processor so that such potential can be realised. This Memorandum examines some of the problems in relation to an application involving the real time generation of imagery, with particular emphasis on the role of the array processor.

This application arose out of the requirement for the Royal Australian Air Force to upgrade their existing Air Traffic Control training simulator, by incorporating computer graphics for generation of the out-of-tower scenes, so that controller/traffic interaction could be extended to a virtually unlimited number of scenarios. The more difficult aspect of generating these scenes is to synthesise, in real time, the motion of distant moving aircraft, since images of close proximity aircraft (e.g. on taxiways or runways), presenting the greatest visual detail and spatial extent, follow pre-determined routine manoeuvres which can be pre-computed in non-real time and stored on disk.

Sandor and Lester [1] considered the feasibility of generating the real time images with a VAX-11/780 computer in conjunction with an AP-120B array processor as an alternative to expensive special purpose hardware for the graphics computations. A novel approach was taken to synthesise the distant images in real time by employing a technique used in animation [2,3] to reduce the rate of computation. This technique involves calculating image frames at half the required rate (12.5 frames per second) and using interpolation to maintain 25 updates per second. This allows more moving aircraft to be generated, and also provides enough computation time between frames so that images can be spatially filtered and hence reduce scintillation and edge "staircasing" effects.

It was originally intended that most of the calculations would be performed in the VAX, with the array processor handling the spatial filtering. This decision was made because the internal organisation of the array processor is well suited to digital signal processing (see [4] for example). However, timing considerations led to the array processor handling a larger share of the computations, and this Memorandum examines these aspects, as well as the process of restructuring the appropriate parts of the algorithm to optimise the parallelism of the array processor architecture.

2. OVERVIEW OF THE ARRAY PROCESSOR ARCHITECTURE

The AP-120B array arithmetic processor exploits parallelism by overlapping several operations within a single processor. It consists of eight functional sub-units interconnected via multiple data paths, and is linked to the host machine (a VAX-11/780) through a direct memory access (DMA) interface (see Figure 1). The AP-120B is a synchronous machine, that is, there is a single master-clock signal which operates all sub-units at the same speed, which is one cycle every 167 nanoseconds. The sub-units will now be described.

The program memory has a capacity of 4K instruction words of 64 bits. This separation of code fetches from data fetches allows both to proceed simultaneously without mutual interference.

Table memory is used for storage of constants, and is separate from main memory to allow parallel access. It has 4.5K of 38-bit read only memory (ROM) and 1.5K of 38-bit writeable memory. (All floating point quantities are expressed as 38 bits, comprising a 10-bit exponent and a 28-bit

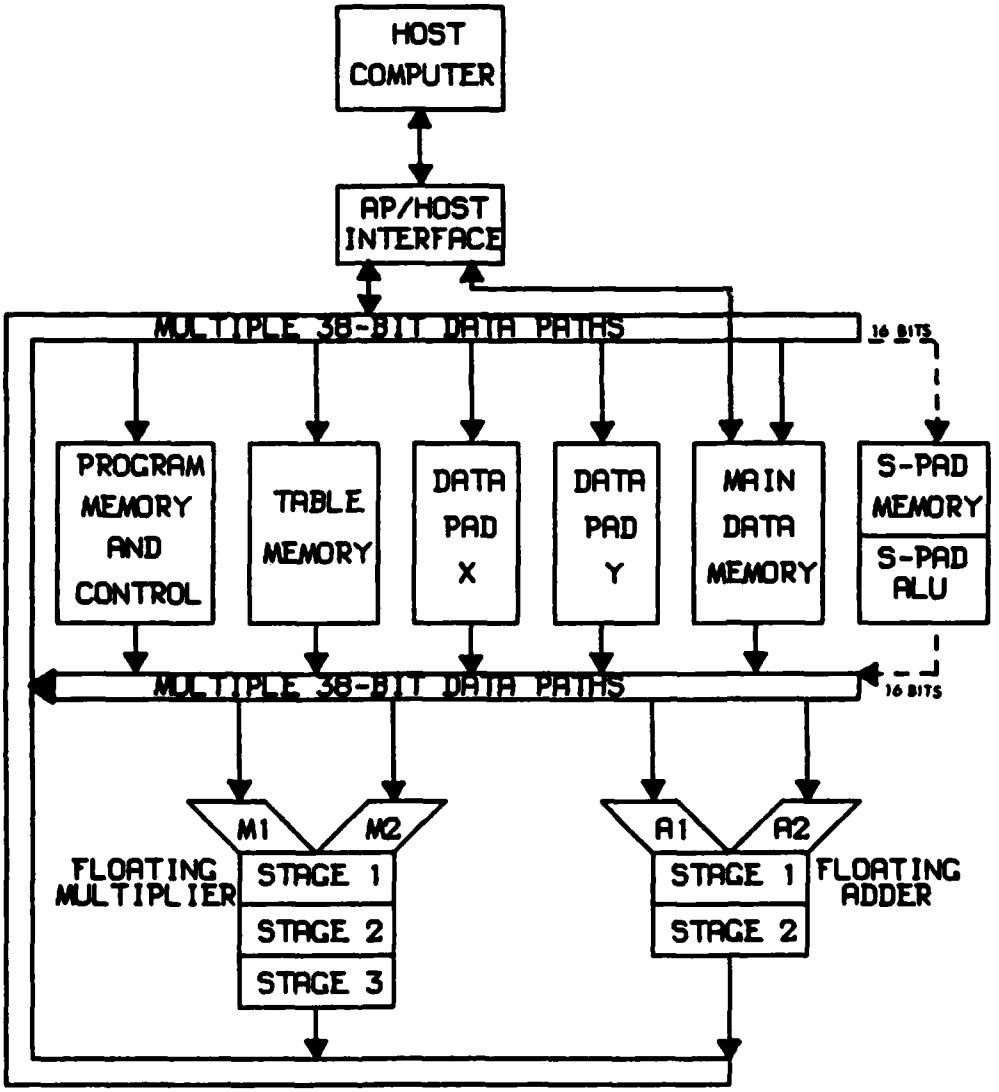


FIGURE 1: ARRAY PROCESSOR ARCHITECTURE

mantissa.) The ROM contains useful constants (such as 1.0, 0.5, π, etc.) and the writeable memory is available for storing often-accessed values or for use as a memory which can be accessed in parallel with main memory.

The data pads each contain thirty-two 38-bit registers. One register can be read and another written from both pads on every cycle, allowing a total of four simultaneous accesses.

The main data memory contains 64K words of 38-bits, and is the primary store for data. It can also function as a secondary store for instruction words when a program is too big to fit completely into program memory and has to be divided into overlays.

The "S-pad" or scratchpad contains sixteen 16-bit registers and an arithmetic and logical unit. It is used for address calculations, and for integer arithmetic. The S-pad also has a hardware bit reverse function to accomplish the bit swapping necessary to access data in scrambled order after a fast Fourier transform.

The floating point adder and multiplier handle the arithmetic. The adder also does data format conversion, such as floating integers.

Not all the functional sub-units can complete a given operation in a single cycle - a floating multiply or a main memory access take 3 cycles; an add or table memory access take 2 cycles. These units operate as pipelines, so that by keeping the pipelines full, a result can be obtained on every cycle.

The AP-120B thus allows a maximum of ten distinct operations on each cycle - 1 add, 1 multiply, 1 address calculation, 2 memory accesses, 4 data register accesses and 1 conditional branch. However, it is not always possible to specify all ten, because the 64-bit instruction words are not wide enough so that certain fields of the instruction word have different meanings dependent upon other fields. In order to gain maximum speed from the array processor, it is sometimes necessary to manipulate desired operations within a sequence of instructions to overcome this inability to specify all operations simultaneously. For further information on the AP-120B, see [5] or [6].

3. ALGORITHMS FOR RENDERING DISTANT AIRCRAFT IMAGES

In order to render aircraft images, it is necessary to have a data base describing them in machine-readable terms. For the present study, aircraft are modelled as sets of polygons in three dimensional space. Each polygon is stored in the data base by specifying the (3-D) coordinates of its vertices, which are ordered so that traversing the edges of a polygon in vertex order leaves the interior to the left when viewing the polygon from outside the aircraft. The colour of the aircraft is specified by assigning a colour to each polygon.

The visual environment of the Air Traffic Control tower simulator comprises five window sectors (each spanning 45° in azimuth and 33° in elevation), so that given the polygon description of an aircraft, the following steps are taken to synthesise its image:

1. For each aircraft modelled as a set of polygons in 3-D space:

   1.1 Determine the aircraft's position and orientation by integration of its translational and angular velocities

   1.2 Compute the vertex coordinates in 3-D space from the aircraft's position and attitude

   1.3 Determine on which window sector(s) the aircraft will be visible

2. For each window sector on which the aircraft is visible:

   2.1 Transform the appropriate polygon vertex coordinates to the coordinate system of the window sector

   2.2 Eliminate polygons which belong to parts of the aircraft that are obscured by the aircraft's own volume (back-face elimination) and adjust the colour of each polygon to take account of the position of the sun

   2.3 Transform the 3-D coordinates of the in-view polygon vertices into a two-dimensional representation as would be seen by an observer (perspective transformation)

   2.4 Remove polygons which are outside the window sector boundary and truncate those which are partially outside (clipping)

3. For each in-view polygon of the transformed and clipped aircraft:

   3.1 Convert vertex coordinates to high resolution space

   3.2 Draw polygons, in back-to-front order, into the picture buffer array (scan conversion)

   3.3 Low pass filter the high resolution image by performing a discrete two-dimensional convolution on it with an appropriate filter function

   3.4 Decimate the high resolution image back to display resolution

At this point, the picture buffer array will contain the synthesised image ready for display. The above sequence constitutes the algorithm for producing each "key" frame. Each interpolated frame is produced by linearly interpolating between two key frames to obtain an intermediate high resolution picture buffer array which is then filtered and decimated as described above in steps 3.3 and 3.4.

4. DISTRIBUTION OF WORKLOAD BETWEEN HOST AND ARRAY PROCESSOR

The assembly language for the AP-120B is called APAL, and provides great flexibility for utilising the parallel and pipelining features of the hardware. It is a difficult language in which to program or make program modifications. However, Floating Point Systems provides a library of mathematical and signal processing functions which are written in APAL and are callable from a host-resident Fortran program. One such APAL routine is available to perform the two dimensional convolutions required for the spatial filtering, and hence it was decided originally to process the low pass filtering and decimation (algorithm steps 3.3 and 3.4) in the array processor.

However, this turned out to be costly in terms of the real time available for such calculations. The computation "cycle time" is 80 millisec, during which time one interpolated frame and one key frame are computed and displayed 40 millisec apart. For the present study, distant moving aircraft were rendered onto a 50 x 50 picture buffer array, and double resolution was selected since it involves the minimum additional storage and execution time. The high resolution (100 x 100) picture buffer therefore comprises 10000 elements, and in order to perform the convolutions in the array processor, it is necessary to transfer the buffer across the DMA interface - a time-consuming operation. A graph of measured time as a function of the number of words transferred is shown in Figure 2. As can be seen, the relationship is linear, and can be summarised in the following equation:

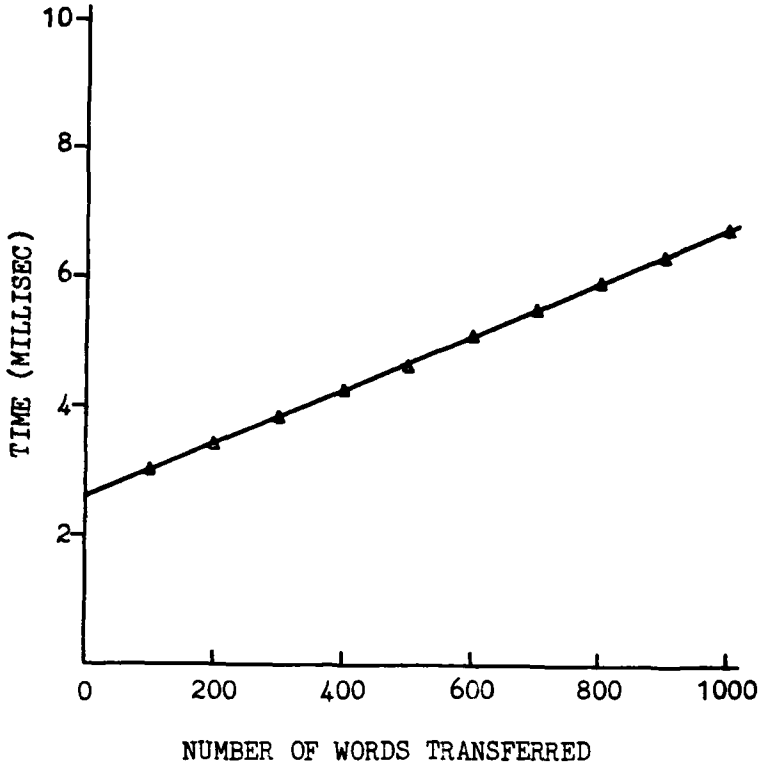


FIGURE 2: GRAPH OF TIME TO TRANSFER DATA TO THE ARRAY PROCESSOR

$$T(n) = 2.6 + 0.004n$$

where T(n) is the number of millisec required to transfer n words. Note that in measuring these times, the test program and all its data were locked into the physical memory of the VAX to ensure that there was no paging or swapping, so that these times represent the best achievable. So, the transfer of a 10000 word array from the VAX to the array processor would require approximately 42 millisec, and since a picture frame is required to be computed and displayed every 40 millisec, application of the array processor in this manner would be inappropriate.

It was therefore necessary to restructure the process so that data transfer time could be reduced. One approach considered was to pack the components of the picture buffer array. Since each element requires only 8 bits, these can be packed four to a word and transmitted as such, so that transfer time would be reduced to 12 millisec. There would also be an overhead of 9 millisec for the array processor to unpack the picture buffer, so that the effect would be to halve the original transfer time. This was still considered to be excessive, and it was therefore necessary to reduce the amount of data that was to be transferred, rather than compacting it. The method for doing this was to put more steps of the algorithm into the array processor since in earlier steps the picture data exists as polygon vertices rather than as individual pixel colours. It was considered that an appropriate choice of algorithm division would be at the point where the VAX has removed all occulted polygons, so that only the vertices of the visible polygons are transferred. That is, the VAX would perform the computations as far as determining the polygons for each picture update (steps 1. and 2. in section 3) and the array processor would complete the process (step 3.). It turned out to be expedient for the array processor program to work directly with the high resolution vertex coordinates, and hence that transformation (step 3.1 in section 3) was included in the VAX program.

5. DETAILS OF ARRAY PROCESSOR WORKLOAD

As discussed above, the array processor handles the scan conversion and filtering parts of the algorithm for rendering distant aircraft images in key frames, and also produces the interpolated frames. These parts of the algorithm will now be described in detail.

5.1 Scan Conversion

The array processor receives a list of polygons from the VAX with each polygon represented by its colour, number of vertices and vertex coordinates (in high resolution space). The polygons are ordered so that those near to the head of the list are behind those further down the list. Thus, the hidden surface problem is solved by overwriting the polygons behind with those in front. The method of scan-converting each polygon will now be described.

Let $V_1, V_2, \ldots, V_N$ be the polygon vertices, where each vertex has coordinates $(V_i.x, V_i.y)$, $i = 1,2, \ldots, N$, with the vertices arranged in counter-clockwise order. Further let $PRED(V_i)$ and $SUCC(V_i)$ denote the predecessor and successor of V_i in the counter-clockwise ordering. Note that $PRED(V_1)$ is V_N and $SUCC(V_N)$ is V_1. Let PIC be a matrix whose (i,j)-th element contains an integer representing the colour of pixel (i,j).

1. [Initialise.] Determine MAX and MIN, the indices of the vertices with highest and lowest y-coordinates (raster scan is from top to bottom). Note that since the vertices are in counter-clockwise order,

$SUCC(V_{MAX})$, $SUCC(SUCC(V_{MAX}))$, ..., $PRED(V_{MIN})$ will be on the left of the polygon, and $PRED(V_{MAX})$, $PRED(PRED(V_{MAX}))$, ..., $SUCC(V_{MIN})$ will be on the right. Set LEFT and RIGHT edge pointers to MAX, and set SCANLINE to $ROUND(V_{MAX}.y - 1)$ where ROUND(x) is formed by rounding x to the nearest integer. Set DECISIONLEVEL to SCANLINE + 0.5 (a pixel will be completely filled in if its centre is not outside the polygon).

2. [Check that left edge is still current.] If $V_{LEFT}.y <$ DECISIONLEVEL then decrement XLEFT by LEFTSLOPE and go to Step 4.

3. [Set up new left edge.] Set LEFT edge pointer to index of the next vertex below the current SCANLINE. The new left edge is the line joining V_{LEFT} to $PRED(V_{LEFT})$. Set LEFTSLOPE to the inverse of this line's gradient, and set XLEFT to the intercept of this line with the line Y = DECISIONLEVEL.

4. [Check that right edge is still current.] If $V_{RIGHT}.y <$ DECISIONLEVEL then decrement XRIGHT by RIGHTSLOPE and go to Step 6.

5. [Set up new right edge.] Set RIGHT edge pointer to index of the next vertex below the current SCANLINE. The new right edge is the line joining V_{RIGHT} to $SUCC(V_{RIGHT})$. Set RIGHTSLOPE to the inverse of this line's gradient, and set XRIGHT to the intercept of this line with the line Y = DECISIONLEVEL.

6. [Fill in the scan line.] Set PIC[I,SCANLINE] to the polygon colour for
   I = ROUND(XLEFT), ROUND(XLEFT) + 1, ..., ROUND(XRIGHT) - 1.

7. [Check for end of polygon.] If SCANLINE = $ROUND(V_{MIN}.y)$ then terminate the process; otherwise decrement SCANLINE and DECISIONLEVEL both by 1 and return to Step 2.

5.2 Key Frame Interpolation

As discussed in Section 1, key frames are calculated at the rate of 12.5 per second, and apparently smooth visual motion of distant aircraft is achieved by interpolating between these to produce 25 frames per second. This approach is justified by noting that visual images of distant aircraft will not change significantly from one display instant to the next (frame-to-frame coherence), so that an observer will, in general, be unaware of the fact that alternate frames are not computed exactly. The interpolation process is executed by maintaining two high resolution picture buffer arrays, K_n and K_{n+1}, in the array processor, where K_n represents the scan-converted image at key frame n. The interpolated image, I_n, is computed as $[K_n + K_{n+1}]/2$, and the final (display resolution) image is displayed such that its centre is halfway between the positions of the aircraft centre at the n-th and (n+1)-th sampling intervals.

5.3 Filtering and Decimation

In this part of the process, the high resolution image is filtered to attenuate high frequencies by a two dimensional convolution of it with an appropriate filter function. The result is then decimated for display at normal resolution. Since the filter convolution need only be evaluated at those high resolution sampling points that correspond to display pixels, filtering and decimation can be accomplished simultaneously, thus saving

significant computational time. Formally the convolution can be expressed as

$$P[i,j] = \sum_{r=-M}^{M} \sum_{s=-M}^{M} a_{rs} H[2i+r, 2j+s]$$

where $H[\cdot,\cdot]$ is the high resolution image array (with boundary conditions $H[m,n] = 0$ for m or n outside the high resolution image), a_{rs} is a symmetric low pass filter kernel with finite support specified by M, and $P[\cdot,\cdot]$ is the filtered image at display resolution. In the present case a Bartlett filter with M = 1 and

$$a_{rs} = 0.5^{|r|+|s|+2} \qquad r,s = -1,0,1$$

was found to produce satisfactory results. The filtering and decimation used can thus be summarised in the following equation:

```
P[i,j] = ROUND(
      0.0625*H[2i,2j]   + 0.125*H[2i+1,2j]   + 0.0625*H[2i+2,2j]
    + 0.125 *H[2i,2j+1] + 0.25 *H[2i+1,2j+1] + 0.125 *H[2i+2,2j+1]
    + 0.0625*H[2i,2j+2] + 0.125*H[2i+1,2j+2] + 0.0625*H[2i+2,2j+2]
           )
```

Note that in the case of interpolated frames, this equation can be modified so that all coefficients are halved thus eliminating the division by 2 in the interpolation. The two sets of coefficients are stored in the writeable table memory of the array processor, with each being used on alternate frames.

6. ALGORITHM IMPLEMENTATION ON THE ARRAY PROCESSOR

The implementation of the array processor's part of the algorithm involves two aspects - the restructuring of the algorithm code to utilise the array processor architecture most efficiently, and the construction of the software interface between the two machines. These aspects will now be considered.

6.1 Algorithm Coding on the Array Processor

To write fast code for the array processor, it is necessary to maximise the utilisation of the parallel and pipelining features of the architecture. To illustrate this, consider the following program fragment which copies x-coordinates from an argument list and then computes differences:

```
for i := 1 to N do
begin
   x[i] := arglist[ptr];
   ptr := ptr + 1
end;
for i := 1 to N-1 do
   deltax[i] := x[i+1] - x[i]
deltax[N] := x[1] - x[N];
```

This calculation is used in the scan conversion of polygons when the inverse gradient of an edge is required. To reduce loop overheads, the two loops can be gathered into a single loop in the following manner:

```
x[1] := arglist[ptr];
ptr := ptr + 1;
for i := 1 to N-1 do
begin
   x[i+1] := arglist[ptr];
   ptr := ptr + 1;
   deltax[i] := x[i+1] - x[i]
end;
deltax[N] := x[1] - x[N];
```

A straightforward translation of this into the machine language of the array processor is given below. The following symbols are used:

MD - "register" into which main data memory contents is read
MI - "register" from which data is written into main data memory
FA - the contents of the floating point adder output stage
< - indicates replacement of the item on the left with that on the right
; - denotes the next instruction is in the same cycle as the current one
" - is used to introduce comments

The machine language translation now follows, with the numbers on the left indicating machine cycles:

```
"Initialise before entering the loop - fetch x(1) from memory,
"set up the loop counter to 1 less than the number of x values
"and save x(1) for the first and last Δx computations.

 1. LDSPI XREG; DB=XBASE              "Get base address of x into s-pad reg
 2. MOV ARGREG,ARGREG; SETMA          "Fetch x(1) (ready at cycle 5.)
 3. LDSPI DELTAXREG; DB=DELXBASE-1    "Put base address of Δx in s-pad
 4. LDSPI COUNTREG; DB=N-1            "Set up loop counter in s-pad reg
 5. MOV XREG,XREG; SETMA; MI<MD;      "Put x(1) where it can be found
       DPX<MD;                        "Save x(1) in data pad X
          DPY<MD                      "And again in data pad Y for last calc

    "Enter the loop - compute Δx(1), Δx(2), ..., Δx(n-1),
    "where Δx(i) = x(i+1) - x(i)

LOOP:
 6. INC ARGREG; SETMA                 "Fetch x(i+1) (i=1 initially)
 7. NOP                               "Wait for memory fetch
 8. NOP                               "To be finished
 9. INC XREG; SETMA; MI<MD;           "Put x(i+1) where it can be found
       FSUBR DPX,MD;                  "Start x(i+1)-x(i)
          DPX<MD                      "Save x(i+1) in data pad X
10. FADD;                             "Push subtraction through adder
       DEC COUNTREG                   "Decrement loop count
11. INC DELTAXREG; SETMA; MI<FA;      "Store Δx(i)
    BGT LOOP                          "Branch back if more to do

     "End of loop - compute Δx(n) = x(1) - x(n)

12. FSUB DPY,DPX                      "Start x(1)-x(n)
13. FADD                              "Push subtraction through adder
14. INC DELTAXREG; SETMA; MI<FA       "Store Δx(n)
```

This code implements the above fragment with the copy and calculation being executed in a single loop. In so doing, the program takes 8 + 6N cycles, where N is the number of times the loop is executed (one less than the number of x-values). This approach to the problem is inefficient because it does not fully exploit the pipelining capabilities, nor does it fully utilise the parallel elements of the array processor. To make better use of pipelining, it is important to note that it is not necessary to perform all the calculations associated with x(i) in iteration i of the loop, as illustrated in the following code:

```
    "Initialise before entering the loop - fetch x(1) and x(2) from
    "memory, set up the loop counter to 1 less than the number of
    "x values, store x(1) and save it for the first and last Δx
    "computations.

 1. LDSPI XREG; DB=XBASE              "Get base address of x into s-pad reg
 2. LDSPI DELTAXREG; DB=DELXBASE-1    "Put base address of Δx in s-pad
 3. MOV ARGREG,ARGREG; SETMA          "Fetch x(1) (ready at cycle 6.)
 4. INC ARGREG; SETMA                 "Fetch x(2) (ready at cycle 7.)
 5. LDSPI COUNTREG; DB=N-1            "Set up loop counter in s-pad reg
 6. MOV XREG,XREG; SETMA; MI<MD;      "Put x(1) where it can be found
       DPX<MD;                        "Save x(1) in data pad X
          DPY<MD                      "And again in data pad Y for last calc

    "Enter the loop - compute Δx(1), Δx(2), ..., Δx(n-1),
    "where Δx(i) = x(i+1) - x(i)

LOOP:
 7. INC XREG; SETMA; MI<MD;           "Put x(i+1) where it can be found
       FSUBR DPX,MD;                  "Start x(i+1)-x(i) (i=1 initially)
          DPX<MD                      "Save x(i+1) in data pad X
 8. INC ARGREG; SETMA                 "Fetch x(i+2)
 9. FADD;                             "Push subtraction through adder
       DEC COUNTREG                   "Decrement loop count
10. INC DELTAXREG; SETMA; MI<FA;      "Store Δx(i)
    BGT LOOP                          "Branch back if more to do

    "End of loop - compute Δx(n) = x(1) - x(n)

11. FSUB DPY,DPX                      "Start x(1)-x(n)
12. FADD                              "Push subtraction through adder
13. INC DELTAXREG; SETMA; MI<FA       "Store Δx(n)
```

This code allows x(i+2) to be fetched during the loop iteration which computes Δx(i). This pipelining of the program has reduced the loop length from six cycles to four cycles, which is a significant saving in time, particularly in the real-time context. Reducing loop length in this way is referred to as "loop-folding", and is probably the most important technique for writing fast code.

A further saving of one loop cycle can be achieved by better use of the parallel elements of the array processor. Since every cycle of the loop involves an s-pad instruction, whereas the floating adder is used for only two of the four cycles, the loop length can be reduced by one cycle by performing the loop counting in the adder, so that only three s-pad references are needed. This is achieved by using the constant 1.0 from table memory and subtracting it off the loop count in the adder, as follows (TM refers to the table memory contents from the most recent fetch):

```
    "Initialise before entering the loop - fetch x(1) and x(2) from
    "memory, store x(1) and save it for the first and last Δx
    "computations. It is assumed that the loop count has been set up
    "as a floating point number in data pad DPY(2).

 1. LDTMA; DB=!ONE                          "Fetch 1.0 from table memory
 2. LDSPI XREG; DB=XBASE                    "Get base address of x into s-pad reg
 3. MOV ARGREG,ARGREG; SETMA;               "Fetch x(1) (ready at cycle 6.)
       FSUBR TM,DPY(2)                      "Start calculating N-1 in adder
 4. INC ARGREG; SETMA;                      "Fetch x(2)
       FADD                                 "Push N-1 through adder
 5. LDSPI DELTAXREG; DB=DELXBASE-1;         "Get base address of Δx into s-pad
       DPY(2)<FA                            "Store N-1 in data pad Y
 6. MOV XREG,XREG; SETMA; MI<MD;            "Put x(1) where it can be found
       DPX<MD;                              "Save x(1) in data pad X
          DPY<MD;                           "And again in data pad Y for last calc
             FSUBR TM,DPY(2)                "Start decrementing loop count

    "Enter the loop - compute Δx(1), Δx(2), ..., Δx(n-1),
    "where Δx(i) = x(i+1) - x(i)

LOOP:
 7. DPX<MD;                                 "Save x(i+1) (i=1 initially)
       FSUBR DPX,MD;                        "Start x(i+1)-x(i)
          INC ARGREG; SETMA                 "Fetch x(i+2)
 8. INC XREG; SETMA; MI<DPX;                "Put x(i+1) where it can be found
       DPY(2)<FA;                           "Save loop count
          FSUB DPY,DPX                      "Start x(1)-x(i+1) in case i+1=n
 9. INC DELTAXREG; SETMA; MI<FA;            "Store Δx(i)
       FSUBR TM,DPY(2);                     "Start decrementing loop count
          BFGT LOOP                         "Branch back if more to do

    "End of loop - save Δx(n)

10. INC DELTAXREG; SETMA; MI<FA             "Store Δx(n)
```

Note that this final version of the code has only one cycle following the loop whereas the previous versions have three. This further reduction results from the fact that the floating adder is needed for only two of the three cycles, so the available cycle is used to start the calculation of $\Delta x(N)$. This means that $\Delta x(N)$ is actually computed N-1 times during the execution of the loop.

Thus, the original program fragment time has been reduced from $8 + 6N$ machine cycles to $7 + 3N$ cycles by making full use of the machine architecture. However, it is a difficult and tedious task for the programmer, and leads to much longer software development times than would be expected for other high speed computers (see [10] for example, for a table comparing software development times on various machines).

The manufacturer has alleviated this problem to some degree by providing a Fortran-to-AP-120B-machine-language compiler and a set of machine language routines to perform common functions. The AP-Fortran compiler has been found to produce code about three to four times slower than tightly-coded machine language (for example, [10],[11]) and is therefore unsuitable for real time applications such as that considered in this Memorandum.

The manufacturer-supplied machine language routines fully utilise the pipelining capabilities of the array processor. However, they cannot always fully exploit the parallel processing features, simply because of

their generality. For example, the above program fragment could be coded using two library routines, VMOV, which copies a vector from one area of memory to another, and VSUB, which subtracts each element of a vector from a corresponding element of another, storing the result as a third vector. VMOV takes 12 + 2N cycles, and VSUB takes 16 + 3N cycles, so that the combination (neglecting initialisation cycles) requires 28 + 5N cycles. This is not much of an improvement over the original straightforward translation which used 8 + 6N cycles. For real-time applications such as that of the current study, it is therefore essential to produce software which fully exploits the parallel and pipelining features of the array processor architecture.

6.2 Software Link between Host and Array Processor

Communication between the VAX and the AP/120B is achieved through an interface which comprises a simulated front panel and a DMA control. The simulated front panel consists of three registers called the "switch", "lites" and "function" registers whose functions closely parallel those of the switches and lights on the console of a stand-alone computer. There are a further four registers for controlling the DMA. The host uses the front panel registers for loading and starting programs, and for examining and modifying array processor memories and internal registers during debugging. A manufacturer-supplied package of Fortran callable routines ("APEX") for manipulating these registers facilitates communication.

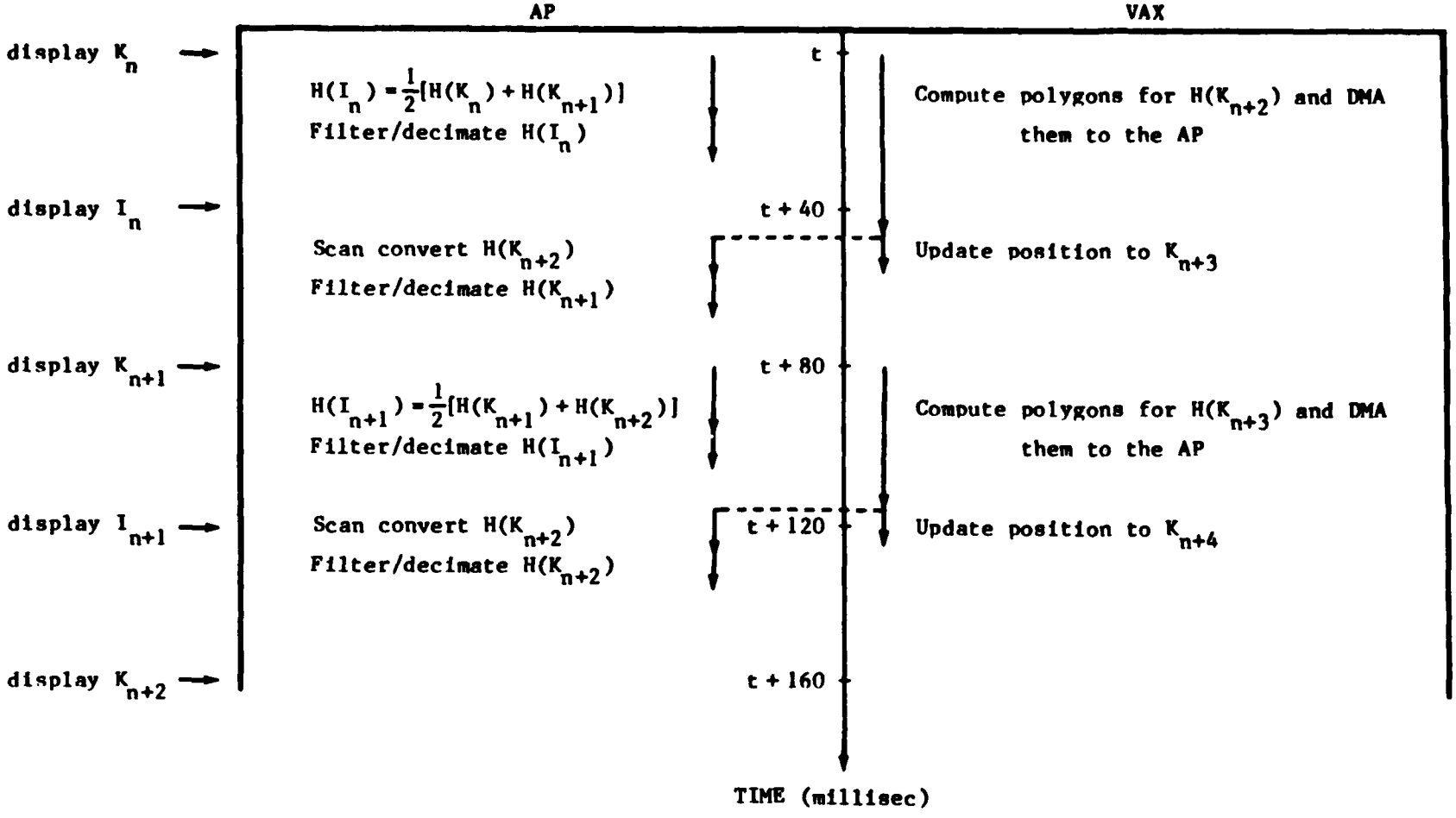
Thus, apart from the development of code within the array processor, a link must be established within the host program so that data can be transferred and array processor programs can be set up and executed. This can be achieved using another manufacturer-supplied program called APLOAD. This links and relocates separate APAL routines, and produces a load module and an interface routine ("HASI"). The load module is a Fortran subroutine which contains the array processor code in a form appropriate for DMA transmission, and the relevant APEX calls required to effect the transmission. The HASI ("host-array processor software interface") is a group of Fortran routines which invoke the execution of the array processor modules. The HASI as produced by APLOAD is very general, in that it checks for and handles various situations, such as array processor modules being overwritten in program memory by others and needing reloading, and transfer of data to s-pad registers prior to each module execution. For the present study, a different software link was written so that array processor routine execution could be initiated as fast as possible.

A further consideration in the real time context is the time required for transferring data to the array processor. As discussed in section 4, the overhead for initiating a DMA transfer is of the order of 2 millisec. Since integer and floating-point quantities must be transferred separately, the most efficient method of transfer is to convert all quantities to one type and hence use only one DMA transfer. A routine to do this and initiate the transfer was written as part of the software link.

7. ALGORITHM TIMING ANALYSIS

The evaluation of the algorithm was conducted using a typical jet aircraft which traversed the display screen under oscillatory rotation about one axis at a simulated distance of approximately 1000 metres. This distance was close enough to allow the vehicle to be discernible as an aircraft, yet far enough to ensure that dynamic modelling could be employed rather than invoking pre-computed routine manoeuvres. Under these conditions, the aircraft was satisfactorily represented by twenty polygons, with, in general, half of them visible at any one instant.

The relative timing of the VAX and array processor computations is shown in Figure 3. In each 80 millisec computation cycle, one interpolated frame



K_n = nth key frame
I_n = nth interpolated frame
H(X) = high resolution form of frame X

FIGURE 3: RELATIVE TIMING OF VAX AND ARRAY PROCESSOR COMPUTATIONS

and one key frame are computed and displayed 40 millisec apart. During each "cycle", the array processor interpolates between two key frames, $H(K_n)$ and $H(K_{n+1})$ (where $H(\cdot)$ refers to the high resolution form of the frame), to produce the high resolution interpolated frame, $H(I_n)$. This is then spatially filtered and decimated to produce I_n, which is displayed at the next 40 millisec point. In the meantime, the VAX performs its calculations for key frame $H(K_{n+2})$, and transfers the visible polygon details to the array processor. The DMA transfers take place in parallel with the processing of both the VAX and array processor, although there could be some slight degradation of processing speed if a processor and the DMA attempt to access the same memory location simultaneously. Once all polygons are transferred, the array processor scan converts them to produce $H(K_{n+2})$, and spatially filters and decimates the previous frame, $H(K_{n+1})$, ready for display. While this is proceeding, the VAX updates the position of the aircraft ready for the next key frame calculation. Thus, there is a fair degree of overlap in the VAX and array processor operations.

Timing of the array processor code was achieved with the aid of a manufacturer supplied simulator, which is a Fortran program that simulates the instruction by instruction execution of array processor code. Since the array processor is a synchronous machine, such simulated execution accurately reflects the states and timing of the AP-120B. However, it is extremely slow - for example, it takes approximately 10 minutes of computer processing to simulate 5 millisec of array processor computation. Timing of the VAX processing enlisted the aid of the VAX interval clock which is accurate to 1 microsec.

TABLE 1

TYPICAL TIMES FOR IMAGE GENERATION

Function	Typical time (msec/frame)
Computation of polygons in VAX and DMA transfer to AP-120B (without clipping)	22
Computation of polygons in VAX and DMA transfer to AP-120B (when clipping required)	34
Update of aircraft position	4
Scan conversion in AP	2
Interpolation in AP	5
Spatial filtering and decimation in AP	4

Typical times as measured are listed in Table 1. The VAX times are listed for cases with and without clipping (step 2.4 in section 3) since this is a time-consuming event which is rarely required, so these times represent best and worse estimates for VAX processing. Table 1 also shows that the VAX provides a considerable part of the computation required. However, transferring more of the computational load to the array processor has to be weighed against the time required to transfer more data from the VAX.

It is of interest to look more closely at the scan conversion times, since these are the only component of the array processor operation that is data dependent. Interpolation and filtering/decimation perform the same set of operations on constant-sized arrays, and hence there is no variation in their execution times. On the other hand, scan conversion time depends upon the input polygon, apart from a constant overhead of 1.7 millisec per frame (10200 array processor cycles) to zero the 10200 element picture buffer array. Table 2 lists the times required to scan convert certain regular

TABLE 2

POLYGON SCAN CONVERSION TIMES

	octagon		"rotated" square		"parallel" square	
area	dimension (pixels)	time (μsec)	dimension (pixels)	time (μsec)	dimension (pixels)	time (μsec)
2	1.68	73	2.00	55	1.41	52
10	3.76	91	4.47	66	3.16	64
25	5.95	104	7.07	87	5.00	69
50	8.41	119	10.0	104	7.07	93
75	10.3	134	11.2	120	8.66	109
100	11.9	147	14.1	135	10.0	109
150	14.6	166	17.3	158	12.2	127
200	16.8	185	20.0	180	14.1	146
250	18.8	204	22.4	202	15.8	166
500	26.6	286	31.6	286	22.4	235
750	32.6	359	38.7	367	27.4	316
1000	37.6	432	44.7	440	31.6	377
2000	53.2	678	63.2	703	44.7	591
4000	75.2	1125	89.4	1166	63.2	1054

polygons (assuming a clear picture buffer). Three polygons are considered - a regular octagon with one vertex uppermost, a square with one vertex uppermost and a square with its edges parallel to the screen boundaries. Each row of the table compares polygons with equal areas (unit dimension is one pixel). The table shows that the dimensions of the polygon play a significant part in determining the scan conversion time. This is a direct consequence of the scan conversion process - the polygon is scanned from top to bottom, filling in the pixels in each row which are inside the polygon. The height of the polygon is of greater importance than the width in timing, since the filling of a row requires only one array processor cycle per pixel (step 6 of the algorithm as described in section 5.1), whereas steps 2 to 7 must be executed for each row occupied by the polygon. Table 2 also demonstrates the effect of the number of vertices - more processing is required each time a vertex is encountered (steps 3 and 5 of section 5.1). This can be seen by comparing the times for the two square orientations. The interaction between vertex count and dimension can be seen by comparing the octagon and rotated square times - the octagon always has the smaller dimension, but for the smaller areas the increase in processing due to the larger vertex count is not offset by the decrease due to smaller dimensions.

The gains made by performing scan conversion in the array processor are illustrated in Table 3, which compares array processor and VAX scan conversion times for the octagons of Table 2. The VAX routine was written in

(optimised) Fortran, as were all the other VAX routines. Table 3 shows that

TABLE 3

COMPARISON OF VAX AND ARRAY PROCESSOR SCAN CONVERSION TIMES

area	VAX time (μsec)	AP-120B time (μsec)
2	440	73
10	640	91
25	730	104
50	920	119
75	1100	134
100	1270	147
150	1530	166
200	1820	185
250	2100	204
500	3470	286
750	4760	359
1000	6040	432
2000	10900	678
4000	20400	1125

the array processor is significantly faster at scan conversion, even for the smaller polygons.

8. CONCLUSION

In this Memorandum, the use of an array processor in the computation of real time images for an Air Traffic Control tower simulation has been described, together with some of the problems that arose in the implementation. It has been shown that in this application, the time required to transfer data from the host to the array processor is significant to the extent that it dictates the partitioning of the algorithm between the two processors. The coding of the array processor program is also an important factor in achieving high-speed execution. Hand-coded software, which fully exploits the parallel and pipelining architecture of the array processor, has been shown to be about ten times faster than a host Fortran routine, although at considerable expense in software development time. The processing efficiency derives from the utilisation of the maximum number of parallel computing elements together with the pipelining of operations to minimise loop lengths ("loop-folding"). Without such considerations, the resulting code can be much slower (for example, half the speed for a simple loop computing first-order differences). Other methods of developing array processor code (AP-Fortran compilation or stringing together a sequence of calls to manufacturer-supplied library routines) are also shown to be inferior, in terms of execution speed, to careful hand-coding in the machine language.

Thus, although the high-speed potential of the array processor is well-suited to real time applications such as described here, realisation of this potential cannot be achieved without considerable effort.

REFERENCES

1. Sandor, J. and Lester, L.N., Computer graphics for air traffic control tower simulation. Proc. Conf. Computers and Engineering, Sydney, Sept. 1983, pp. 24-29.

2. Booth, K.S., Kochanek, D.H. and Wein, M., Computers animate films and video. IEEE Spectrum, Vol. 20, No. 2, Feb. 1983, pp. 44-51.

3. Parke, F., Parameterized models for facial animation. IEEE Comp. Graph. and Appl., Vol. 2, No. 9, Nov. 1982, pp 61-70.

4. Bucy, R.S., Ghovanlou, F., Moura, J.M.F. and Senne, K.D., Nonlinear filtering and array computation. IEEE Computer, Vol. 16, No. 6, June 1983, pp. 51-61.

5. Charlesworth, A.E., An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 family. IEEE Computer, Vol. 14, No. 9, Sept. 1981, pp 18-27.

6. Floating Point Systems, Inc., AP Programmer's Reference Manual. Publication No. 860-7319-001A, 1981.

7. Gupta, S. and Sproull, R.F., Filtering edges for grey-scale displays. ACM Computer Graphics, Vol. 15, No. 3, 1981, pp 1-5.

8. Newman, W.M. and Sproull, R.F., Principles of Interactive Graphics, Wiley, New York, 1979.

9. Sutherland I.E., Sproull, R.F. and Schumaker, R.A., A characterization of ten hidden-surface algorithms. Comp. Surveys, Vol. 6, No. 1, Mar. 1974, pp 1-55.

10. Karplus, W.J. and Cohen, D., Architectural and software issues in the design and application of peripheral array processors. IEEE Computer, Vol. 14, No. 9, Sept. 1981, pp. 11-17.

11. Forsstrom, K.S., Array processors in real-time flight simulation. IEEE Computer, Vol. 16, No. 6, June 1983, pp. 62-70.

DISTRIBUTION

AUSTRALIA

Department of Defence

Central Office

Chief Defence Scientist)
Deputy Chief Defence Scientist)
Superintendent, Science and Program Administration) (1 copy)
Controller, External Relations, Projects and Analytical Studies)
Defence Science Adviser (U.K.) (Doc Data sheet only)
Counsellor, Defence Science (U.S.A.) (Doc Data sheet only)
Defence Central Library
Document Exchange Centre, D.I.S.B. (18 copies)
Joint Intelligence Organisation
Librarian H Block, Victoria Barracks, Melbourne
Director General - Army Development (NSO) (4 copies)

Aeronautical Research Laboratories

Director
Library
Superintendent - Systems Division
Author: L.N. Lester

Materials Research Laboratories

Director/Library

Defence Research Centre

Library

Navy Office

Navy Scientific Adviser
Directorate of Naval Aircraft Engineering

Army Office

Scientific Adviser - Army
Engineering Development Establishment, Library
Royal Military College Library

Air Force Office

Air Force Scientific Adviser
Aircraft Research and Development Unit
  Scientific Flight Group
  Library
Technical Division Library
RAAF Academy, Point Cook

.../contd.

DISTRIBUTION (CONTD.)

Central Studies Establishment

Information Centre

Universities And Colleges

Adelaide	Barr Smith Library
Latrobe	Library
Melbourne	Engineering Library
Monash	Hargrave Library
Newcastle	Library
Sydney	Engineering Library
N.S.W.	Physical Sciences Library
Queensland	Library
Tasmania	Engineering Library
Western Australia	Library
R.M.I.T.	Library Dr H. Kowalski, Mech. & Production Engineering
Macquarie	Library

UNITED KINGDOM

Royal Aircraft Establishment, Bedford, Library
British Library, Lending Division

UNITED STATES OF AMERICA

NASA Scientific and Technical Information Facility

SPARES (10 copies)

TOTAL (69 copies)

Department of Defence

DOCUMENT CONTROL DATA

1 a. AR No	1 b Establishment No	2 Document Date	3. Task No
AR-003-953	ARL-SYS-TM-73	AUGUST 1984	DST 84/040

4. Title	5. Security a. document	6. No Pages
ARRAY PROCESSOR UTILISATION IN THE COMPUTATION OF REAL-TIME IMAGES	UNCLASSIFIED b. title c. abstract U U	18 7. No Refs 11

8. Author(s)	9. Downgrading Instructions
L.N. LESTER	-

10. Corporate Author and Address	11. Authority (as appropriate) a. Sponsor b. Security c. Downgrading d. Approval
Aeronautical Research Laboratories PO Box 4331, Melbourne, Vic. 3001	-

12. Secondary Distribution (of this document)

Approved for public release.

Overseas enquiries outside stated limitations should be referred through ASDIS, Defence Information Services Branch, Department of Defence, Campbell Park, CANBERRA ACT 2601

13 a. This document may be ANNOUNCED in catalogues and awareness services available to ..

No limitations.

13. b. Citation for other purposes (ie casual announcement) may be (select) unrestricted (or) as for 13 a

14. Descriptors	15. COSATI Group
Computer graphics › Real time operations › Pipelining (computers) › Image synthesis › Array processors	09020

16. Abstract

The application of an array processor to the real time generation of aircraft images in Air Traffic Control tower simulation is described. Particular emphasis is placed upon the problems which arise in achieving efficient utilisation of the array processor. It is shown that the time required to transfer data from the host to the array processor is a serious consideration in the partitioning of the algorithm between the two processors. It is also shown that array processor execution speed is about ten times faster than a host Fortran routine, provided that the array processor code is written to maximise the number of parallel computing elements used and to minimise loop lengths by pipelining of operations. These requirements result in a considerable software development effort. A timing analysis of the resultant array processor code is also presented.

Originator-supplied Keywords

PF 65

This page is to be used to record information which is required by the Establishment for its own use but which will not be added to the DISTIS data base unless specifically requested.

16. Abstract *(Contd)*

17. Imprint

Aeronautical Research Laboratories, Melbourne

18. Document Series and Number	19. Cost Code	20. Type of Report and Period Covered
SYSTEMS TECHNICAL MEMORANDUM 73	716490	

21. Computer Programs Used

22. Establishment File Ref(s)

END

FILMED

2-85

DTIC