Broadcasting Arrays
- A Highly Parallel Computer Architecture
Suitable For Easy Fabrication

Alan H. Bond
Electrical Engineering Systems Department
University of Southern California
Los Angeles, California 90089

January 1987
Abstract

We present a general purpose SIMD computer architecture which is instruction broadcasting, but has no global control unit. It is derived from the ICL Distributed Array Processor. It consists of an array of simple identical intelligent memory modules and should be very easy to fabricate, particularly in VLSI.

Each module has local RAM, and the program is stored in these local RAMs. The array fetches instructions, modifies operand addresses and broadcasts instructions to itself.

The computer is self-contained in that it stores and executes its own programs, and has general input-output. It runs somewhat faster than the ICL DAP, and has a richer instruction set.

The processing element is only slightly more complicated than the DAP processing element. It has two extra one bit registers, I and M, creating an instruction plane and modifier plane. The connections between PEs consist of

1. a shift to N, S, E or W neighbours
2. an OR connection to row and column extractor lines
3. a carry to N neighbours
4. an OR connection to instruction extractor lines
5. external input and output lines

We have extended the row and column enabling using latches. The next instruction is determined by an instruction field in the current instruction.

Instruction decoding takes place at compile time, producing a 64 bit instruction. Instruction fetching is simply a load operation from store to the I plane.

Fabrication is very much easier. The design has good testability properties and presents possibilities for fault tolerance.

Such an architecture requires a certain minimum critical size to be practical, which is about 64 x 64 x 64Kbits.

The design lends itself to connection into MIMD networks.
1 Introduction

1.1 SIMD computers and their fabrication

Existing SIMD square array processor supercomputers [2][3] such as the DAP [4][5] and the MPP[6], consist of two parts, the array of processing elements and the control unit. Since the processing elements are identical and very simple, they can easily be fabricated in VLSI and indeed, both the DAP and MPP, as well as CLIP[7] and others have such chips in existence. Bit-serial array processor circuits are surveyed by Fountain[1]. Usually there are 8 or 12 PEs per chip and usually the PE local store of 1K to 64K bits per PE is on separate chips. However, the control unit for the array is not easily fabricated. It is complex, having to perform a wide variety of functions:

1. Fetch, decode and broadcast instructions.

2. Do global logical and arithmetic operations in global registers in the control unit.

Data movement operations are also required for:

1. Extraction of PE bit planes into global registers.

2. Broadcasting of data from global registers into array bitplanes.

There is only one control unit for the whole array, it differs somewhat for different sized arrays and so on. The control unit is quite costly, probably costing of the same order as the entire PE array. It is usually made in lumped chip circuitry.
1.2 Our approach

This paper introduces a new design in which the control unit functions are performed in the PE array itself. We call our design a Broadcasting Array. It arose from the observation that existing PE arrays already contain almost enough logic to allow the functions to be carried out. We describe the resulting design, which is thus very easy to fabricate. It consists simply of an array of simple processing elements. Instead of a control unit, there is only a set of external connections through which the array can broadcast an instruction to itself. We also mention a variation with a set of connected Broadcasting Arrays, some of which can act only as control units and others only as processing arrays.

Our approach contrasts with that of J.K.Illiffe [8], who has explored the integration of the host functions into the serial control unit, to make a self contained array processor with a serial and a parallel part.

Our approach also contrasts with the systolic array approach of H. T. Kung and others[9]. S. Y. Kung[10] has argued that, for VLSI, broadcasting is an inappropriate technique due to time skewing of the transmission of clock impulses. However, the disadvantage of the purely local interactions of systolic arrays is that all data communication must be propagated across the array. This leads to time bounds, as explored by Beyer[11] for example.

1.3 The DAP as starting point

We took as our starting point the ICL Distributed Array Processor (DAP) [12][4][13]. Taking the PE array as given, we tried to show that the DAP control unit could be represented in the array. We found that we had to generalize the PE quite substantially, however the PE is still very simple and easy to fabricate. As will be seen, our work could just as easily have taken
some other machine as its starting point, although the DAP PE did have the advantage of being sufficiently simple to limit the explosion of design possibilities. The DAP instruction set is very large and comprehensive and we have not attempted a completed enumeration of detailed implementations of every instruction. We have however demonstrated that this can be done straightforwardly. By showing that such a complex machine can be implemented, we have demonstrated that our design does provide a parallel computer which is practical and upon which the vast variety of SIMD parallel programs can be run.

The basic point is that the DAP is a proven SIMD processor. It has been the subject of intense design activity at ICL for several years. It has a high level language - DAP Fortran, a large subroutine library including mesh calculations, sorting, image processing functions etc.[13]. We have demonstrated its suitability for computer vision [14][15][16]. Thus if we can demonstrate that our design can do at least what the DAP can do, at least as fast, but be fabricated much more easily, then we shall have shown its practical relevance.

1.4 Fault Tolerance and Testability

Arrays of processing elements of the type described can easily be configured for fault tolerance, by putting an extra m rows and n columns, but only using say 64 x 64 at a time. In the event of a fault in one PE, either the row or column containing this PE are replaced by a spare row or column by simply adjusting the row or column masking.

Since the design is very highly modular, with just one small module, this lends itself to a testable design. In addition to the ease of testing of the very simple PE circuits, there are testability techniques for regular arrays of elements [17]. Testing could possibly be performed at run time, leading to automatic reconfigurability.
2 The DAP

2.1 Capabilities of the DAP and the DAP PE

Let us take stock of the means at our disposal - the DAP PE array and the end we wish to achieve - the DAP control unit. The DAP processing element is diagrammed in Figure 1.
Figure 1: DAP Processing Element
It is obtained from a store chip by adding three one bit registers and an ALU. The processing elements are arranged in a square array, giving N, S, E, and W connections. The entire 64X64 array has row and column lines also. The DAP control unit is diagrammed in Figure 2. It has eight 64 bit registers. It fetches instructions, decodes them and broadcasts them to the array, so that every processing element executes the same instruction.
Figure 2: DAP Master Control Unit
One can imagine the DAP as made up of a serial part - the control unit, and a uniformly parallel part - the PE array. The instructions fall into about seven categories:

1. **PE instructions.**
   Boolean operations upon the Q, A, or C registers and the selected store bit S. There can be one, two or three operands and one or two resulting bit values. Operands can also be taken from N,S,E or W neighbors. Since all PEs perform the same operation, this amounts to a N,S,E or W shift of a bit plane. The shift can be for an arbitrary distance, taking a proportional amount of time.

2. **Extraction from array to control unit.**
   Either a selected row or column into a global register, or else an entire selected bit plane with ANDing of corresponding row or column bits to form a 64 bit number.

3. **Broadcasting of data from control unit to array.**
   From a global register either to a selected row or column or to an entire bit plane by replicating the register contents over rows or columns. Also a single selected bit of a selected global register can be broadcast over a selected plane.

4. **Arithmetic and Logic Operations within the Control Unit.**
   The usual register operations amongst the 64 bit registers.

5. **Control Instructions.**
   Unconditional and conditional branching upon global register contents. There is also a hardware DO loop with instruction cache.

6. **Input-Output.**
   This is performed by the host processor for the DAP and uses the row lines.
7. Row Operations.
Parallel adding of two bit planes viewed as 64 pairs of 64 bit integers. this uses the C register as a carry, and uses a carry ripple.

2.2 Implementation of the DAP PE

Let us now look at how these instructions are implemented, in order to see the capabilities of the PE array. The parts of the DAP instruction that are broadcast to each PE are:
(i) the PE operation code, which controls the various multiplexors in the PE.
(ii) the inversion bit, which inverts the operands.
(iii) the direction selection bits for N,S,E,W or self.
(iv) the store address to be used, which is the result of modifying the address with the contents of a specified global register.

Row and column selection are sent to the row and column logic of the array, likewise the geometry selection bits.

2.3 DAP control unit functions we wish to implement

We have now some idea of the logic available in the PE array. The functions performed by the master control unit that we need to implement by logic in the set of processing elements are as follows:

1. Fetching the current instruction from store
2. Forming the store address by adding the contents of a global index register to the address in the instruction

3. Broadcasting the instruction to the array of processing elements

4. Instruction sequencing i.e. computing the address of the next instruction to be fetched

5. Global index registers

6. Arithmetic and logical operations upon global index registers

7. Broadcasting from a global register to a bit plane

8. Extraction from a bit plane into a global register

In addition our array must be able to carry out all the other DAP functions including PE operations and shifts.

Also we include Input/Output, even though this is a host function in the existing DAP, so that we have a self-contained general purpose SIMD architecture.

3 The Architecture of Broadcasting Arrays

3.1 The main idea

We introduce two more one bit registers, I and M, into each PE, forming two more 64 x 64 bit planes. We call these the instruction plane and the modifier plane. The basic action of the system is to fetch a plane of instructions from the S store into the I plane and to execute out of there. The fetching of a plane of instructions is not automatic, it is done by executing
a machine code instruction to load from S into I. An instruction plane fetch is only needed every 64 instructions unless a branch occurs. Sequencing through instructions is done by a 6 bit instruction selection field in every instruction, which specifies the location in the I plane of the next instruction. The selected row in the I plane contains the current instruction and its I registers are connected to the 64 bit PE control bus. This control bus carries the current instruction to the PEs. It contains the 16 bit address field for the S store as well as a completely decoded instruction, directly to the various gates and multiplexors in the PEs, which select the instruction that occurs, see Figure 3.

![Figure 3: The main execution pathway](image)
The decoding of the instruction is assumed to be done at compile time, so the form of machine instruction is longer viz. 64 bits, instead of the 32 bits used in the DAP.

Branching instructions are implemented by fetching a whole new plane of instructions.

Address modification is done by the usual parallel rowwise add with ripple carry, adding a given M row into the instruction. To obtain speed here, an additional row line and column line register were added to the PE, and the activity control was extended to a row enable and column enable line. In this way, one row can read directly from any other row in one cycle.

For input-output, this can use the row or column lines, but in addition, we add two (binary) lines directly to each PE, for external input and output respectively.

It is a basic assumption that the registers and logic can be clocked faster than the store S, hence the PE logic and registers are to be made in one technology and the store in another.

### 3.2 Basic PE operations

The resulting PE and its data paths are shown in Figure 4. Figure 5 gives a logic diagram.
Figure 4: Our processing element, showing data paths
Figure 5: Logic diagram description of our processing element
The basic operation is to take two operands, selected from:
(ii) The store S.
(iii) The row and column lines.
(iv) The rowshift and columnshift lines.
(v) The external input line.
(vi) The constants 1 and 0.

The two operands may then be inverted. They are input to the ALU, which performs a full add. The output is possibly inverted and is sent to any subset of:
(ii) The store S.
(iii) The row and column registers.
(iv) The rowshift and columnshift registers.
(v) The external output line.

The instruction bit allocation is shown in Figure 6.
ADDRESS 16 BITS FOR 64K BITS

SELECT OPERAND 1, 4 BITS
A REGISTER INPUT TO ALU

SELECT OPERAND 2, 4 BITS
B REGISTER INPUT TO ALU

CARRY ENABLE, 1 BIT
INVERT OPERANDS & RESULTS, 4 BITS
(A, B, S_i, C_i)

SELECT S_i DESTINATION, 4 BITS

SELECT C_i DESTINATION, 4 BITS

RS SHIFT CONTROL, 2 BITS

CS SHIFT CONTROL, 2 BITS

ROW ENABLE, 7 BITS

COLUMN ENABLE, 7 BITS

IS INSTRUCTION SELECTION, 6 BITS

Figure 6: The instruction bit allocation
There are thus two row lines and two column lines, one for row to row and column to column transfers using the row and column registers, and one for shifts, using the rowshift and columnsshift registers. The row and column registers are ORed onto the row and column lines. Only one place shifts are used, since these are by far the most common in our system, and we didn’t want the extra complication of representing and decoding arbitrary length shifts. N-S and E-W shifts can occur simultaneously however.

3.3 Timing

The basic timing cycle is as shown in Figure 7.

![Figure 7: The basic timing cycle](image-url)
Thus any shifting is done first, and the cycle ends with outputs appearing on row and column lines.

There are four different cases, depending upon whether the S store is used as an input, an output, both or not at all.

i.e. Register to Register

Store to Register

Register to Store

Store to Store

These cases would be recognized by the timer by the settings of the S in bit and S out bit in the instruction.

3.4 Control of activity

3.4.1 Control of activity in PEs

The instruction is only executed if

Row enable AND Column enable AND local A register

are all on.
3.4.2 Control of row and column lines

As shown in Figure 8, there are 1 bit registers at the ends of each row and column and these receive the masking part of the instruction. The row and column masks either select 1 row (6 bits), 1 column (6 bits) or all rows and columns (1 bit), requiring 13 bits.

![Diagram of row and column enable registers](image)

Figure 8: Row and Column Lines

3.4.3 Bypassing a row or column

A shift bypasses any disabled PEs, by the enable line and combinational logic.

This same bypassing logic allows fault tolerance, by allowing any rows or columns to be made invisible by simply disabling them.
3.4.4 Sending data from one selected row to another

This is needed for rapid address modification. We enable the source row and select the row register as an output. The next instruction, we enable the target row and select the row line as an input for the instruction.

It is necessary for a disenabled PE to have its output registers zeroed, at the end of the instruction.

3.4.5 Hardware program loops

We can implement program loops in hardware. The entire loop should ideally fit on one plane, however multiple planes wouldn’t cause much problem.

In the DAP, the hardware DO loop modifies the fetched instructions in the loop by incrementing, decrementing or leaving their address fields. The increments can only be 1 however.

In our case, a plane of loop modifiers would be set up by the compiler and added in each time round the loop. Thus all incrementation for the entire loop is performed in one cycle.

3.5 I/O and External Connections

3.5.1 Execution of instructions

The broadcasting array generates a 64 bit instruction along the instruction lines. It also accepts a 64 bit instruction along the PE bus. An external circuit can connect these together and clock the execution of the system.
3.5.2 Input/Output

Input can be along row and column lines to either a selected row or column or else to all rows or all columns, giving broadcast input.

Input can also be along the 64 x 64 external input lines.

Input and Output are effected by an appropriate instruction selecting the appropriate inputs.

Programs would be loaded similarly.

Output can be from a selected row or column, or can be an OR extraction from all rows or all columns.

Similarly, it can be along the 64 x 64 external output lines.

Thus, we have vector I/O of 64 bit words or array I/O of 64 x 64 bit planes.

For video I/O it would be probably best to have a shift register about 8 bits wide i.e. with width equal to the pixel grey level resolution.

4 The Operation of Broadcasting Arrays

We can now show how the broadcasting array architecture we have defined can implement all of master control unit functions of the DAP. This is what we set out to do, and justifies the design as a useable general purpose SIMD parallel computer.
4.1 Data operations

4.1.1 Global index registers

The M plane with CL lines provide 64 64 bit fields, which can be used as modifiers as well as combined in arithmetic and logical operations.

4.1.2 Loading global registers from RAM store

We can define the macro

LOAD ADDR,REG

to mean

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>A</th>
<th>B</th>
<th>CA</th>
<th>INV</th>
<th>S</th>
<th>C</th>
<th>RS</th>
<th>CS</th>
<th>RE</th>
<th>CE</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>REG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

i.e. ADDR -> REG.

where REG can be Q, A, C, I or M.

4.1.3 Storing from global registers into RAM store

We can similarly define the macro

STORE ADDR,REG

to mean

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>A</th>
<th>B</th>
<th>CA</th>
<th>INV</th>
<th>S</th>
<th>C</th>
<th>RS</th>
<th>CS</th>
<th>RE</th>
<th>CE</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR</td>
<td>REG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

24
4.1.4 Arithmetic and logical operations upon global registers

These instructions are of the form:

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>A</th>
<th>B</th>
<th>CA</th>
<th>INV</th>
<th>S</th>
<th>C</th>
<th>RS</th>
<th>CS</th>
<th>RE</th>
<th>CE</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R1</td>
<td>R2</td>
<td>CA</td>
<td>INV</td>
<td>R3</td>
<td>R4</td>
<td>RS</td>
<td>CS</td>
<td>RE</td>
<td>CE</td>
<td></td>
</tr>
</tbody>
</table>

which takes input from the R1 plane and R2 plane, performs the operation and places the result in the R3 and R4 planes. R1, R2, R3 and R4 can be any of Q, A, C, I or M.

The primitive operations are simply full adding with selectable inversions of inputs and outputs. All other operations have to be achieved by sequences of such primitives. This is how the DAP works also, and is well documented, see for example [5] or [8].

INV specifies any inversions of operands and results. RS and CS specify any shifting of rows or columns, by one place either E, W, N or S. RE and CE specify which row or column is involved in the operation, with the value of 64 meaning use all (rows or columns).

Numbers are usually represented and processed serially, i.e. the successive bits of a plane of integers occupy successive address planes. Then addition of two 64 x 64 planes of integers is performed serially by a sequence of full add operations, to place the result in another set of bit planes.

4.1.5 Row adding of register planes

It is also possible to use more parallel representation of integers, and this is required for address modification during instruction execution.

When we have two planes of 64 64-bit integers, we can add corresponding rows in parallel, i.e. in one cycle, using the ripple carry mechanism provided. We use the carry-in input and
the carry-out is rippled.

The operation $R1 + R2 \rightarrow R3$ is

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>A</th>
<th>B</th>
<th>CA</th>
<th>INV</th>
<th>S</th>
<th>C</th>
<th>RS</th>
<th>CS</th>
<th>RE</th>
<th>CE</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R1</td>
<td>R2</td>
<td>CI</td>
<td>0</td>
<td>R3</td>
<td>CO</td>
<td>0</td>
<td>0</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

4.1.6 Broadcasting data from a global register to a bit plane

This takes two instructions:

(i) M register $\rightarrow$ CL lines.

The M register involved is selected using the row enable lines.

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>A</th>
<th>B</th>
<th>CA</th>
<th>INV</th>
<th>S</th>
<th>C</th>
<th>RS</th>
<th>CS</th>
<th>RE</th>
<th>CE</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>CL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>M</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

(ii) CL lines $\rightarrow$ ADDR address

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>A</th>
<th>B</th>
<th>CA</th>
<th>INV</th>
<th>S</th>
<th>C</th>
<th>RS</th>
<th>CS</th>
<th>RE</th>
<th>CE</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR</td>
<td>CL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

4.1.7 Extraction from a bit plane into a global register

Again two instructions are required.

(i) The bit plane is put onto the RL or CL lines.

ADDR $\rightarrow$ RL (or CL)
This ORs all the corresponding bits in the same column or row respectively.

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>A</th>
<th>B</th>
<th>CA</th>
<th>INV</th>
<th>S</th>
<th>C</th>
<th>RS</th>
<th>CS</th>
<th>RE</th>
<th>CE</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>RL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

(ii) The RL (or CL) lines are then selected as input for the next instruction, and put into the selected row of the M plane, by using the operand M and enabling that row.

**RL (or CL) -> M**

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>A</th>
<th>B</th>
<th>CA</th>
<th>INV</th>
<th>S</th>
<th>C</th>
<th>RS</th>
<th>CS</th>
<th>RE</th>
<th>CE</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>RL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>M</td>
<td>64</td>
</tr>
</tbody>
</table>

### 4.2 Instruction operations

#### 4.2.1 Instruction fetching

This is done by an explicit load operation

LOAD ADDR,I

This fetches a plane of 64 instructions from the bit plane ADDR. The instruction executed first is given by the IS field in the fetch instruction.

#### 4.2.2 Address modification

This consists of two instructions in addition to executing the instruction whose address field is being modified.
<table>
<thead>
<tr>
<th>Operand</th>
<th>0</th>
<th>1</th>
<th>Q</th>
<th>A</th>
<th>C</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>RL</th>
<th>CL</th>
<th>RS</th>
<th>CS</th>
<th>EI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>Result</td>
<td>0</td>
<td>1</td>
<td>Q</td>
<td>A</td>
<td>C</td>
<td>I</td>
<td>M</td>
<td>S</td>
<td>RL</td>
<td>CL</td>
<td>RS</td>
<td>CS</td>
<td>EO</td>
</tr>
<tr>
<td>Code</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 9: An assignment of coding for operands and result destinations

(i) M -> CL

This is row-enabled at the M register being used.

(ii) CL + I -> I

This is row-enabled at the current instruction being used.

The most complex sequencing occurs when we use a store address which is modified by the contents of an index register. We give a complete example to explain how the machine works in this case. The instruction bit allocation is shown in Figure 6. We have assigned the coding operands and result destinations as in Figure 9.

The example three instruction sequence is given in Figure 10.

4.2.3 Broadcasting the instruction

The external circuitry connects the selected IS row of the I plane with the PE bus.
<table>
<thead>
<tr>
<th>I</th>
<th>ADDRESS</th>
<th>A</th>
<th>B</th>
<th>CA</th>
<th>INV</th>
<th>S</th>
<th>C</th>
<th>RS</th>
<th>CS</th>
<th>RE</th>
<th>CE</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>53</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>64</td>
<td>64</td>
<td>3</td>
</tr>
</tbody>
</table>

LOAD ADDR(M) Q

i.e. load from address ADDR modified by contents of global register M, into global register Q.

Assume that ADDR is 53 and that M contains the value 7.

Assume instructions are at locations 0, 1 and 2 in the I plane.

0. PICK UP MODIFIER

   TAKE INPUT FROM M

   ENABLE ROW 5, ALL COLUMNS

   PUT INTO RL

1. ADD INTO INSTRUCTION ADDRESS FIELD

   TAKE FROM RL AND I

   ENABLE ROW 2, ALL COLUMNS

   PUT INTO I

2. EXECUTE STORE INSTRUCTION

   LOAD FROM STORE ADDRESS 60, SAY

   ALL ROWS AND COLUMNS

   PUT INTO Q

Figure 10: Example instruction sequence
4.2.4 Instruction sequencing

As previously discussed, this is done by an explicit next instruction address field, for instructions in the I plane, and by loading further planes of instructions as required.

5 Economics, Performance and Fabrication

5.1 Compile Time Expense

We are advocating a complete instruction decoding to be generated at compile time, except instruction plane fetches which are generated at load time (or more strictly, storage allocation time).

Isn’t this expensive in compile time?

Well, most of compile time is spent in processing other than code generation. Further, code generation is just done with skeletons, so this is easy, with only a few substitutions required anyway.

5.2 Space Taken By Code

In most array processing applications, for example mesh calculations and computer vision, the storage required for the code is very small compared to that required for data. There are after all 4096 data sets, but only one program in SIMD operation. In the 2Mb DAP (64 x 64 x 4K bits), the program may be say 10,000 machine instructions. In the DAP with 32 bit instructions, this occupies about 40K bytes i.e. 2% of store. In our design, we need 64
bits per instruction and also explicit address modification instructions, hence this is up to 128 bits per equivalent DAP instruction. However this would still be only 8% of the total program size.

The expense of storing on backing store media of programs in symbolic source form is the same as before. For programs in machine code form, this is greater but still a small expense.

5.3 The price of memory, compile time decoding and code size

Memory is becoming cheaper. Also special cpus are becoming relatively more expensive. Hence presumably we must have economic pressure towards simpler and more general purpose cpus. Hence we can argue that compile time decoding is a likely future trend. This will:
(i) Increase code sizes by up to 4X.
(ii) Decrease cpu sizes greatly.
(iii) Make little difference to compile time costs.
(iv) Move cpu operation into the machine level of description.

5.4 Speed Comparison to the DAP

The DAP cycle is (i) fetch 4 instructions (ii) decode (iii) broadcast (iv) do PE operation. If a (machine code) DO loop then fetch DO loop into special cache.

In our case, we fetch 64 instructions at a time also. As a rough measure, assuming a branch every 48 instructions of which 20 involve the store and 28 the registers only, to pick numbers out of a hat, then the DAP would need 12 fetches from store and 20 store operations = 32
cycles. Our machine would need 1 fetch and 20 store operations = 21 cycles. So it would up to 50% faster than the DAP.

5.5 VLSI Fabrication

A Broadcasting Array is made from 4096 identical PEs with very little else.

Each PE would be made from two chips
(i) The PE logic chip, based on Figure 5.
(ii) A normal memory chip, 4K, 16K or 64 K bits.

The PE chip is simple to design and layout. The number of pins would be quite high, about 54 for the PE chip. The memory chip would take the 12 - 16 bit address lines of course.

One could put several PEs on one chip, as this wouldn’t increase the number of pins too much as they are all on a common bus. There would be extra pins for each new row and column line.

6 Advantages and disadvantages

In this section, we list advantages and disadvantages of our self-broadcasting approach in comparison to other design methods for parallel machines. We discuss hardware and software issues separately.
6.1 Hardware

6.1.1 Advantages

1. It is much easier to fabricate. The complete design consists of a regular array of identical PE units, and each unit is very simple. This highly modular design has advantages in ease of design and design checkout. In production, the simple design is easy to implement and easy to test.

2. It can use current technology i.e. 64Kbit chips with 100ns clocking.

3. It runs faster than other implementations, since it uses fewer instruction fetches, and has a more powerful instruction set.

4. Loops of less than 64 instructions are executed without any instruction fetches, and this can be arranged by the compiler since it can determine instruction-fetch instructions and address allocation of instructions.

5. 64 modifier registers are available; the DAP only has 8.

6.1.2 Disadvantages

1. It uses a broadcasting bus. For an array of size $64 \times 64 \times 64K$, this could require one or two repeaters.

2. It needs several clock cycles (2 short and 1 long) to execute and instruction using a modified address.

3. It needs more memory to store the program.
4. It needs more PE hardware
   Two extra registers, I and M
   Two extra lines CL and RL
   Extra flip-flops for RL etc.

6.2 Software

6.2.1 Advantages

1. It has a considerably more powerful instruction set.
   (i) any two values from Q,A,C,I,M,RL,CL,RS,CS etc. can be used as operands.
   (ii) direct input and output from external peripherals.
   (iii) instruction fetches can be controlled in software, allowing code optimization.

2. Existing DAP software can be used, including the APAL macro-assembly language and
   the DAP Fortran compiler.

6.2.2 Disadvantages

1. It is necessary to do more work at compile time, in instruction decoding and in creating
   the next instruction field.

2. It is necessary to do slightly more work in allocating storage at load time, so that
   instruction planes are allocated with corresponding instruction plane fetches at the
   ends of planes, and possibly elsewhere.
7 General Discussion

7.1 Completeness of the instruction set

Our instruction set is in some sense complete, since any combination of control lines produces some effect and all registers and lines are treated equally as sources of operands and as destinations for results.

We have a sort of micro language of PE boolean operation.

This to the author is reminiscent of the advance made by Algol over Fortran, in providing a structure which was much more regular in providing elements and rules for their combination and allowing the entire closure of the set as legal programs.

7.2 Critical Size

Because any interesting processing within a store element requires a certain minimum number of bits, we are lead to a critical size phenomenon.

The broadcasting array concept only makes sense, and sets of intelligent store elements can only produce a self contained processor, if above a certain critical size, which is about 64 x 64 x 64.

Referring to Figure 11, we consider the determination of the depth, width and length of a Broadcasting Array.

Such processing power requires certain data storage requirements to prevent severe I/O limitation. It was found with the DAP that 8K bits per PE caused I/O problems, and even
Figure 11: Determination of Size of the Broadcasting Array

16K per PE also. This is an average figure. Roughly speaking, the processing power of the system is large, and in many applications the processing can be completed quite quickly compared to the time taken to input data. Hence a depth of 16, 32 or 64 K bits is indicated.

In order to specify a boolean instruction on one two or three bits, we need about 4 or 8 bits. To specify a store address, we need 8, 10 or 16 bits, say. To specify routing along row and column lines, a further 5 bits, and to specify row and column masking, about 14 bits. We are thus lead to an instruction length of > 40 bits, requiring an array width of > 40. Complexity of instruction depends upon

(i) number of basic operations and complexity of PE logic
(ii) address space size and complexity
(iii) number of modifiers and registers per instruction.

The array *length* is really determined by two properties:

(i) how many global modifier registers, say 4 minimum.

(ii) the number of instructions to be fetched at a time. This relates to the branching rate of the program i.e. after how many instructions we normally branch. This only applies to significant branches i.e. where each arm of the branch has a probability > 10% say, making instruction fetching important economically. An instruction fetch only takes one instruction to perform. Hence with length *l* instructions, the instruction fetch occurs approximately once out of *l* of the time. For length 64, this is about 2%. Hence, the length is not so critical, even length 16 would be quite useable, giving 16 global modifier and registers, with instruction fetching taking 6% of the time.

The amount of storage for a 64x64x64K broadcasting array is $2^6 \times 2^6 \times 2^{10} = 2^{28}$ bits which is 32 megabytes.

### 7.3 Each element is complete, and minimal

As shown in Figure 12, each memory element contains all the features for a complete computer. However, it is not a complete computer; only when connected into a set of identical such elements does it form a computer.
Figure 12: Each element contains the features of a complete computer
Its contribution to the computer is a single bit towards each datum. It is probably close to the \textit{minimal size element}.

8 Connecting Broadcasting Arrays Together

In the author’s opinion, MIMD operation is best sought not in modifying the internal structure of the array but in connecting arrays together into networks.

These would be strongly coupled networks. Instruction and data streams could be multiplexed and sent to other arrays.

Following on from section 3.5, we can connect arrays together in several ways.

8.1 Data Connection

(i) Data Vector Connection
Row or column of one to row or column of another. Remember that an input operation can broadcast to all rows or columns, and that a vector output operation can be an OR extraction.

(ii) Data Array Connection
Either a 1:1 connection between two arrays, or a multiplexed connection to several
8.2 Data and instruction connection

(i) Instruction Connection
One array could send instructions to another or receive instructions from another. Some data connections would usually also be required to allow branching conditional upon the results of computation. If branching were only conditional upon external inputs, then a purely instruction connection could be used.

(ii) A Separate Specialized Broadcasting Array as a Control Unit.
One could use one array to hold the program and to fetch instructions, and another to hold data and to execute instructions. In this case, a simpler PE could be used with only 3 planes, Q,A and C in the data array and I and M in the control array.

Data connections would also be needed, since the I and M plane would be in the control array and data broadcasting from and extraction into these would be needed. Hence row and column data vector connections would be required.

The system would run faster since instruction preparation could proceed in parallel with instruction execution. The control array could be made of higher speed technology if desired, since it would be smaller. We might even be able to implement this from existing DAP or MPP chips.

9 Summary

We have demonstrated that a highly modular and easily fabricated array processor can be designed. This array processor includes all the DAP logic as well as many extensions.
It could thus run all the DAP software, including the DAP Fortran compiler, application program libraries and computer vision software. Probably in practice 64 x 64 x 64 Kbits is a minimum size, and it could be run at about 200ns per machine cycle.

10 Acknowledgement

The author would like to thank Don Beal, Chris Hudson, Mark Burton and Mark Witkowski for criticism of this research.
References


