The Circuit Model for Parallel Algorithms

Bob Doran
Ian Thomas

Department of Computer Science
University of Auckland

Abstract

This report explores the use of circuits as practical models for the semantics of parallel algorithms. It is shown that circuits are useful for explaining the meaning of parallel algorithms, just as a textual notation is useful for comprehending the algorithms underlying circuits. The relationship between circuits and algorithms is developed to the depth where it becomes clear that they are equivalent in expressibility. In order to express algorithms that involve re-use of resources, the concept of the *data barrier* is introduced and used to extend the algorithm-circuit relationship to cover pipelined and systolic circuits, in particular.

1. Introduction

This report explores the relationship between parallel algorithms and circuits, developing the equivalence between a textual notation for one and the graphical notation for the other. This equivalence can be approached from different directions - the path selected here, because it has not been well explored, is the use of the circuit as a model for expressing the meaning of parallel algorithms.

With serial computation there is really only one universally-accepted model of computation. This, the von Neumann model, involves a single point of control with state of computation held in a random access memory. This is so widely accepted that it is possible to describe algorithms in many languages with no difficulty in comprehension or portability - there may be a babel of serial computer languages but there is only one lingua franca for meaning (there are other models, but not so widely used).

In contrast, with parallel algorithms there are many competing models available that are not readily interchangeable. Some have been developed in depth over the years (e.g. Data flow [DAVI82], SIMD [ALMA89], CSP[HOAR78]) others have been proposed in monographs (e.g. [CHAN88], [SABO88]), with new models appearing frequently (e.g. [FELD92]). In order to control this proliferation there have been proposals for "bridging models" that can stand between the expression of algorithms in programs and the details of implementation ([VALI90])

If we have read the trends correctly, it may be that the resolution of this confusion will be to sidestep the problem when writing practical programs. A primary goal of software development is to write programs that are easy for humans to understand. It does appear that humans have great difficulty with understanding multiple simultaneous activity unless it is carefully structured. There is a need for notations and models that involve explicit synchronization of parallel processes, but humans should avoid such notations as much as possible.[1]

The practical solution for software development is to avoid expressing parallelism as much as possible and to leave its extraction for compilers. The trend should be for programs to be expressed serially as in the past with parallel expression restricted to data parallel operations such as on vectors, perhaps with the occasional use of simple control barriers as the most common form of synchronization. The trend in computer architecture to shared memory (albeit virtual) and the move from SIMD to vector processing confirm this direction [BELL92] as will the promulgation of programming standards such asHighly Parallel Fortran.

Despite the likely finessing of the practical programming problem, there is still the need for other models at the "machine language" level where explicit synchronization must take

[1] There is a good analogy with structured programming, we need languages with gotos for implementation. Languages without enable us to express algorithms in a manner that makes them easier to understand.

place. There is no practical difficulty here with the existence of multiple models, just as there are many machine languages. However, there is still a difficulty with communication of the ideas and concepts that underlay very highly parallel algorithms and devices. For example, before a description can be made of a new sorting algorithm, the model in which it is described must itself be explained and understood. This report is concerned with means for expressing highly-parallel algorithms in such a manner that the model of computation does not become an additional burden.

It seems that there are some desiderata that a widely accepted model for explaining parallel algorithms should conform to:

- One is that it should be informal - it should be clear what is meant without recourse to detailed definitions, just as a rough flow or structure diagrams may be used to explain serial algorithms.

- Another is that it should not involve itself with management of limited resources. With serial algorithms we do not concern ourselves with limitations on storage when explaining our high-level algorithms; with parallel algorithms we should in addition not use models that require features that are concerned with shortage of processors.

- A third, perhaps, is that the model should extend its range into the area of limited resources simply.

These considerations lead us to explore the use of circuits as parallel algorithm model. They can certainly be used most informally, and have the extreme characteristic that we are looking for in relation to resources - no limitation on componentry is assumed and all components are active continuously. Although circuits and algorithms are known to be of the same computational power, and the relationship between algorithms and circuits has been noted before, the analogy is a strange one to programmers and circuit designers alike. It also involves a number of conceptual difficulties that we must overcome.

2. The Circuit Model of Parallel Algorithms

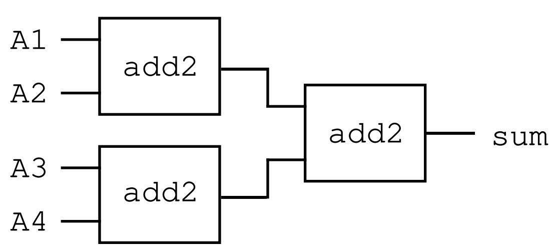
"It is possible, and even tempting, to view a program as an abstract mechanism, as a device of some sort. To do so, however, is highly dangerous." (Dijkstra, 1989)[2]

The *circuit model* defines a parallel algorithm to be the action expressed by a circuit, i.e. all programs are indeed machines. This may seem an extreme leap to take but we will gradually show that it isn't such a large step as it appears to be.

A circuit can be defined, formally or informally, as a collection of *components* connected by *wires*. Wires communicate values between components - it is simple to assume that values flow in one direction only. Circuits and components have some wires serving as inputs and others as outputs. Components respond to changes in their inputs to perhaps change values on their output wires. This definition allows components to be defined atomically or to have their action expressed by circuits in turn. There is no need at this stage for further definition or formality.

With a circuit, all components are continuously active. There is no concept of sequence except that implied by the flow of data between components. There is thus no concept of a circuit ever completing its execution. Instead we have the concept of *external stability* - if the inputs to a circuit are held constant the circuit may eventually reach a stage where the outputs change no further.

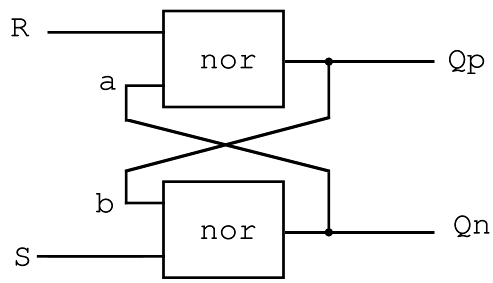
The meaning of any particular circuit, in higher-level terms, must be determined by logical reasoning. However the meaning will usually involve the state of the outputs from the circuit when it reaches external stability. Sometimes we are interested in the relationship between particular inputs and outputs, for example the following algorithm for adding four numbers using components that add two numbers:



However, there are cases when the meaning of the algorithm can only be determined by considering the history of the inputs. For example, the R-S flip/flop algorithm:

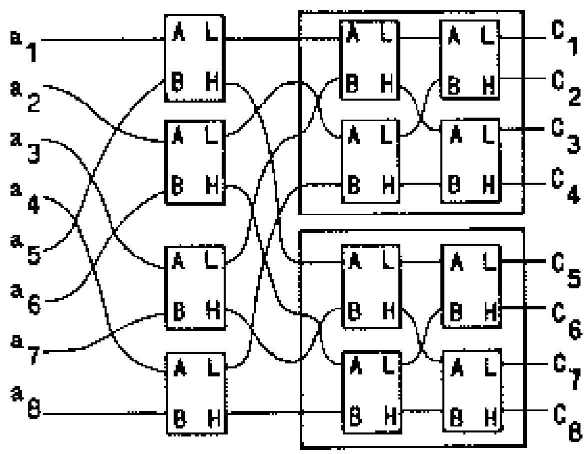
---

[2] It must be admitted that this challenge is taken out of context [DIJK89].



There are other cases where we cannot define any sensible meaning to a circuit-algorithm because it does not reach a state of external stability ever (just as a serial algorithm may loop endlessly), or because we cannot reason unambiguously about the output states without invoking machine-level details of timing and delay (as in the 11 -> 00 transition on inputs to the flip/flop).

However, the cases where the meaning is clear (which we will soon explore) are so extensive that the circuit model is immediately useful. In fact, it is already in widespread use informally. For example, the following circuit for sorting numbers is given in the seminal paper by Batcher [BATC68]:



Given that it is of such use, why hasn't the model been explored further? To start with, graphic notation has until now been difficult to use directly for programming. The recent spread of "wimps" has lead to increase in interest in visual programming for which the circuit model is a natural choice [SHU88]. However, the graphic notation suffers from the defect of being verbose and bulky, so may never be a preferred approach to practical programming.

In practice, we need the compactness of a familiar textual notation for expressing algorithms in order to avoid graphic tedium when it is inappropriate. What we will do now is develop a textual equivalent without losing the semantics or generality of the circuit model.[3]

[3]The relationship between the circuit model and the data flow model should be made clear. Structurally, a data flow diagram and circuit look very similar and data flow and circuit languages may be implemented in

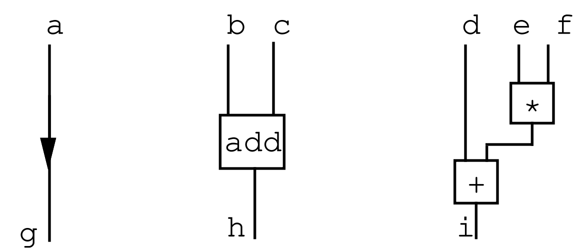
3. Textual Language for Circuit-Algorithms

There have been many textual languages developed for expressing circuits [ULLM84]. However, these tend to have the goal of describing the behaviour of *real* circuits, including physical details. The goal here is somewhat different, to express the algorithms that underlay circuits in a succinct and informal manner. We will try to make the correspondence between text and circuit as unsurprising as possible. The text notation will be close to the Algol-Pascal style.

Variables, Expressions and Assignments

In a circuit,wires may be distinguished by means of identifiers. These names correspond to variables in the text notation. The one variable name may correspond to different wires when used at different places in the text. Wires carry data in one direction only which may be indicated by arrows but will be omitted when there is no confusion caused.

Simple expressions correspond to atomic components in a circuit. Production of an output corresponds to an assignment. The following examples make the relationship plain:




```
g := a          h := b+c          i := d+(e*f)
```

Blocks

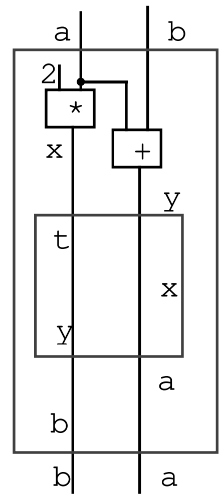
The next step is to show textually the connections between atomic elements. Rather than assume that circuits are structured arbitrarily, which would require that all connections be listed explicitly, the structure of the text is used to imply connections for circuits that are themselves well-structured.

A block is a list of statements (including blocks) analogous to the Algol block. Each block has as inputs each variable that is used in the block but not declared to be local to the block and as outputs all (and only) those non-locals that are assigned within the block. There are two extreme types of blocks, the *vertical* and *horizontal*. With vertical blocks, the named inputs to each statement are obtained from the textually closest preceding statement within the the block, or are inputs to the block itself - this corresponds to the usual rules for data

similar ways. However, traditional dataflow is very concerned with conservation of resources which it achieves with strict firing rules and the concept, derived from Petri nets, of tokens carrying data. Dataflow is thereby limited in applicability - it is quite impractical to express even the well-defined meaning of real circuits and parallel devices such as the flip/flop using dataflow. However, much of what is to follow applies equally to the dataflow model.

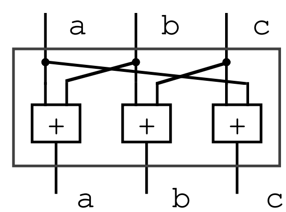
flow. These rules can be defined very formally and carefully but we will often not use declarations when showing algorithms informally. An example:

```
begin {global variables a, b}
     var x, y: integer;
     x := 2*a;
     y := a+b;
     begin
          var t;
          t := x;
          x := y;
          y := t;
     end;
     a := x;
     b := y
end;
```



In the horizontal block, which we tend to flag with the keyword **all**, every statement inside the block obtains its inputs from the inputs to the block. Multiple assignments to the same variable are not permitted in a horizontal block. Example:

```
begin all
      a := a+b;
      b := b+c;
      c := c+a;
end;
```



The use of nested blocks is adequate to express many parallel algorithms, but not all. To allow for feedback and other connections that break the textual nesting rules, it is possible to override the textual structure with a "**from**" tag on any variable coupled with labels to indicate the source. For example, the RS flip/flop can be expressed by either of the following algorithms[4]:

```
begin {R,S, Qp, Qn global}
      var a, b: Boolean;
      a  := Qn(from L);
      Qp := R nor a;
      b  := Qp;
      Qn := S or b;
L:end;
```

```
begin  {R,S, Qp, Qn global}
     begin all
          Qp := R nor a(from L);
          Qn := S nor b(from L);
     end;
     begin all
          a := Qn;
          b := Qp
     end
L:end;
```

[4] The correspondence here with serial algorithms is evident. The control *goto* and the data *from*both allow terrible algorithms to be expressed, but we cannot rule out the **from** if we do have the goal of being able to express *all* parallel algorithms.

Procedures, or parameterised blocks

So far, there is a one-one relationship between the text notation and the corresponding circuit. By parameterising blocks, giving them names, and allowing re-use, the text notation becomes much more succinct and powerful.

We will use the notation of Pascal for procedures, but the meaning ascribed to the use of a procedure has to be quite different to the usual. In the circuit model we assume no shortage of resources, hence each use of a component is a separate instance of that component. Applying this to textual notation, we have to regard each use of a procedure as a complete expansion of that procedure with the usual replacement of formal with actual names. For example, the following algorithm to add four numbers (declarations reduced to a minimum):

```
procedure add4(A[1:4], sum);
begin
     add2(A[1:2], sum1);
     add2(A[3:4], sum2);
     sum := sum1 + sum2
end;
procedure add2(A[1:2], sum);
begin
     sum := A[1] + A[2]
end;
```

The parallel algorithm:

```
add4(B[1:4], total)
```

represents exactly the algorithm:

```
begin
     add2(B[1:2], sum1);
     add2(B[3:4], sum2);
     total := sum1 + sum2
end
```

which is:

```
begin
     sum1 := B[1] + B[2];
     sum2 := B[3] + B[4];
     total := sum1 + sum2
end
```

which is the same circuit as in the example above.

Conditionals

Conditionals are a conceptual problem if one is concerned about shortage of resources. The condition must be evaluated first to avoid unnecessary work, yet this often doesn't give the fastest-executing result. The interpretation of conditionals with the circuit model is plain, however. The condition and the true and false parts correspond to circuits that are continuously active and the result of the condition is used to select the results from either the true or false section. The conditional is a partially horizontal and partially vertical block. That is, if we write:

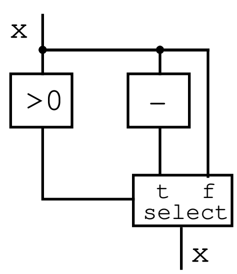
```
if cond(INc, c) then T(INt,OUTt) else F(INf,OUTf)
```

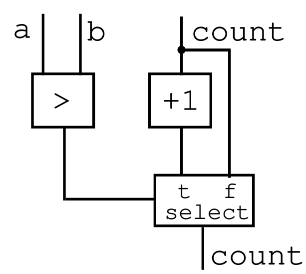
we mean:

```
begin
  begin all cond(INc, c); T(INt,OUTt); F(INf,OUTf) end
  select (c, OUTt, OUTf)
end
```

where "select" is an array of gates that directs through to outputs of the conditional the outputs of the true or false section or the unchanged inputs if the true and false sections have different outputs. That rule is pretty hard to express formally or to describe informally, but the intention should be clear from examples[5]:

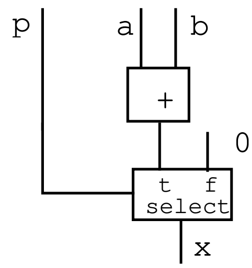
```
if x>0 then x := -x;        if a>b then count:=count+1;
```



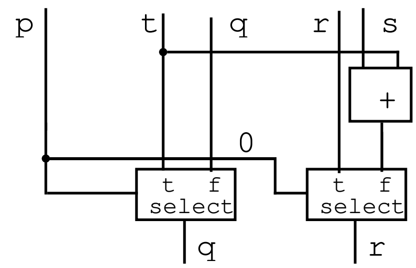


[5]One could, not too seriously, describe this behaviour, which is the opposite of lazy evaluation, as "hyperactive evaluation".

```
if p then x := a+b
    else x := 0
```




```
if p then q := t
    else r := s+t
```



Optimization

As with any mechanical translation process, our rules for translation of text to circuit will often give rise to circuits that can be greatly optimised. An optimised circuit represents a different algorithm to the original and in practice we would be interested in defining the meaning of "equivalent behaviour" and performing optimization in general.

However, we will not discuss optimization other than a very special case. A simple programming optimization is compile-time evaluation of constant expressions. Because of our definition of conditionals, the evaluation of a constant condition can allow major optimizations by omitting the true or false section. We will refer the reduction possible after constant expression evaluation as *constant reduction* and term the reduced circuit and the original *constant-equivalent*.

Limited Recursion and Repetition

If the extent of recursion if limited then it is easy to give a constant-equivalent interpretation to a recursive algorithm. For example, if we generalise the parallel summer above to:

```
procedure addn(A[1:2**n], sum, n);
begin
   if n=1 then sum := A[1] +A[2] else
       begin
       split(A[1:2**n],A1[1:2**(n-1)],A2[1:2**(n-1)])
       addn(A1[1:2**(n-1)], sum1,n-1);
       addn(A2[1:2**(n-1)], sum2,n-1);
       sum := sum1 + sum2
       end
end;
```

Then `addn(A[1:8], sum, 3)` is constant equivalent to:

```
addn(A[1:4], sum1,2);
addn(A[4:8], sum2,2);
sum := sum1 + sum2
```

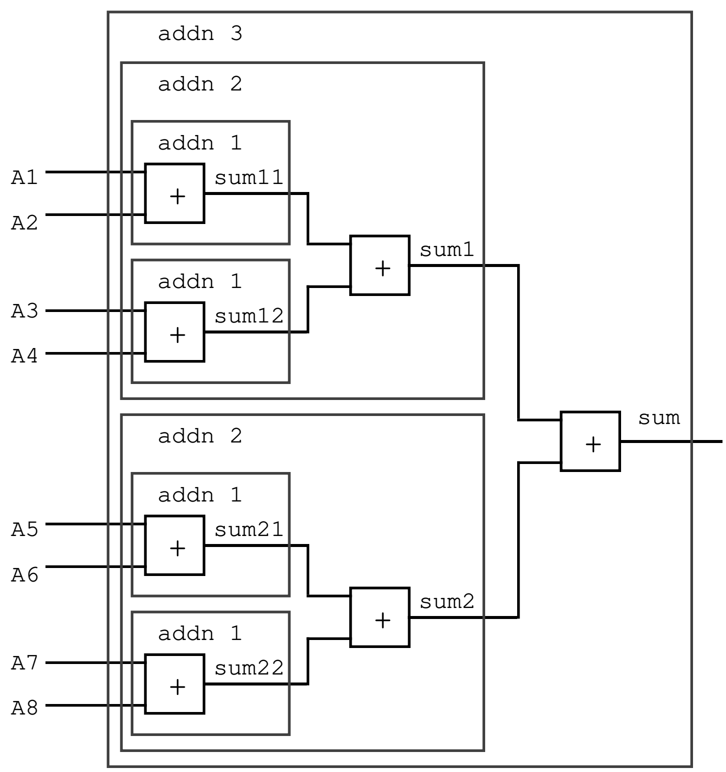
which is constant equivalent to

```
addn(A[1:2], sum11,1);
addn(A[3:4], sum12,1);
sum1 := sum11 + sum12
addn(A[5:6], sum21,1);
addn(A[7:8], sum22,1);
sum2 := sum21 + sum22
sum := sum1 + sum2
```

which is constant equivalent to

```
sum11 := A[1] + A[2];
sum12 := A[3] + A[4];
sum1 := sum11 + sum12
sum21 := A[5] + A[6];
sum22 := A[7] + A[8];
sum2 := sum21 + sum22
sum := sum1+sum2
end;
```

which is the circuit:

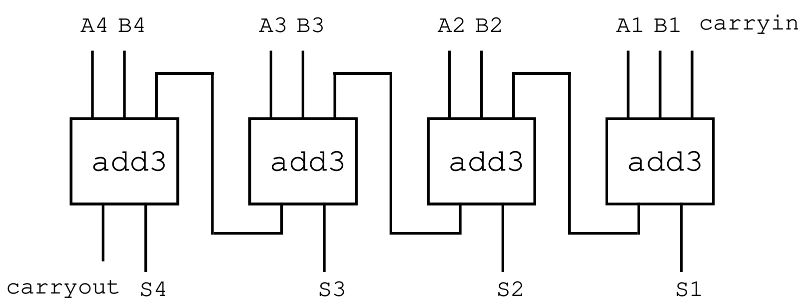


(The procedure "split" is not defined in the above example because it is assumed that it is constant equivalent to a mere relabeling of the wires.)

The same reasoning applies to limited repetition which is equivalent to loop unrolling. Thus the following is the algorithm for a 4-bit ripple adder:

```
procedure add3(a,b,c:Boolean; var sum, carry: Boolean);
begin
   sum := .....;
   carry := ....;
end;
ci := carryin
for i from 1 to 4 do
   begin
        ci := co;
        add3(A[i],B[i],ci,S[i],co)
   end;
carryout := co
```

which is constant equivalent to the following circuit:



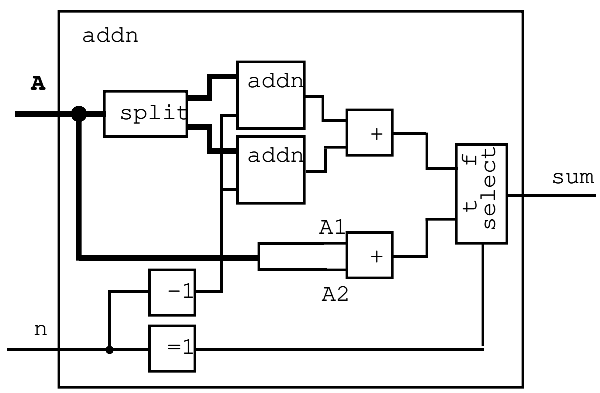
Unbounded Recursion and Repetition

So far the correspondence between text and circuit has been rather natural. The behaviour that is obtained from the model is just what one would expect the text to mean. However, if we want to give a circuit equivalent for all apparently meaningful textual algorithms, we will have accept that it is possible for some to have quite tricky meanings.

There are certainly recursive algorithms where it is not known before execution what the depth of the recursion will be - an example is the recursive addition program above for which the parameter n is variable. One could approach this by defining separate expansion and execution phases to the interpretation of the text - reintroducing the concept of dynamic macro expansion. This seems to us to be an unfortunate choice because we wish to be able to map parallel algorithms to circuits, not to processes for producing circuits. We take the alternative approach of regarding such a circuit as having an *infinite* extent, only part of which has to complete for the circuit to reach external stability. Thus the algorithm:

```
procedure addn(A[1:*], sum, n);
begin
   if n=1 then sum := A[1] +A[2] else
       begin
       split(A[1:*],A1[1:*],A2[1:*])
       addn(A1[1:*], sum1,n-1);
       addn(A2[1:*], sum2,n-1);
       sum := sum1 + sum2
       end
end;
```

we choose to regard as an infinitely expanded circuit as follows:



Visualize this in three dimensions where the recursive calls are in the background and the same size as foreground addn, or, alternatively, expand the inner calls in reduced size. Then the circuit, although of infinite depth, is of finite width. If n is set to any definite non-negative integer value the circuit will reach a state of external stability in finite time and only a finite section of the circuit "matters".

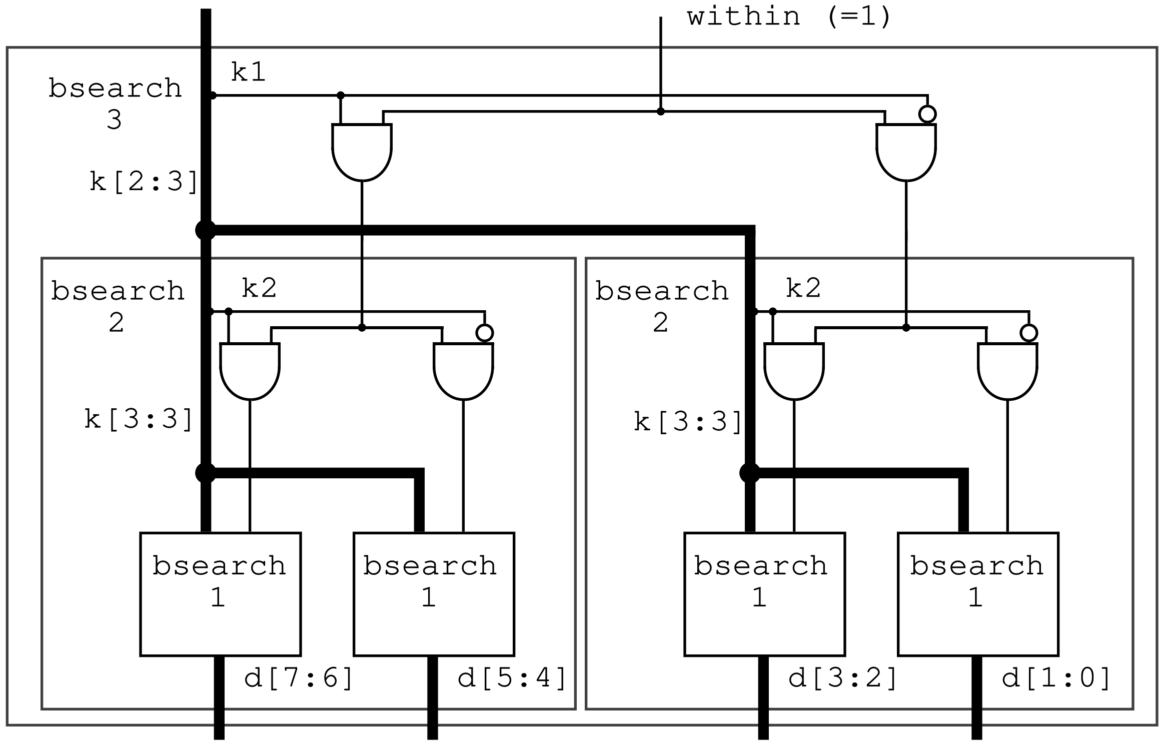
This use of infinities may seem to be unfortunate. However, we are quite happy with serial programs that operate for an infinite amount of time. We wish to give meaning to parallelism with no shortage of resources so it is not surprising that potential infinite time is modelled by potentially infinite extent. Of course, if n is specified as being limited then the circuit is finite and if n is ever set to a specific constant such as 3 then the circuit becomes constant-equivalent to the simple example given above.

As another example of the utility of recursion when describing parallel circuit algorithms, here is a parallel version of binary search. Searching does not usually result in interesting parallel algorithms, but, in this case the task is to determine the location of the n-bit key k[1:n] regarded as a binary integer (k[1] most significant) in the list of integers 0 - 2**n-1 -

the result to be indicated by setting all bits of the array d[0:2**n-1] and where only bit comparisons may be performed.

```
procedure bsearch(within, k[1:*], var d[0:*], n);
begin
   inbot := within and (not k[1]);
   intop := within and (k[1])
   if n=1 then d[0:1] := inbot, intop
   else begin
       bsearch(inbot,k[2:*], var d[0:*],n-1);
       bsearch(intop,k[2:*], var d[2**n:*],n-1);
       end;
end;
```

`bsearch(true, k[1:3], d[0:7],3)` is clearly constant equivalent to a 3-bit tree decoder. The first two levels of expansion would be:



Repetition

This can now be approached by regarding it as a special form of recursion:

For example:

```
for i := 1 to n do body(INbi,OUTbi);
```

could be regarded as:

```
procedure forloop(i,INli,OUTli);
begin
    if i<n then begin
        body(INi,OUTi);
        i:=i+1;
        forloop(i,INLip1,OUTip1)
        end;
    end;
```

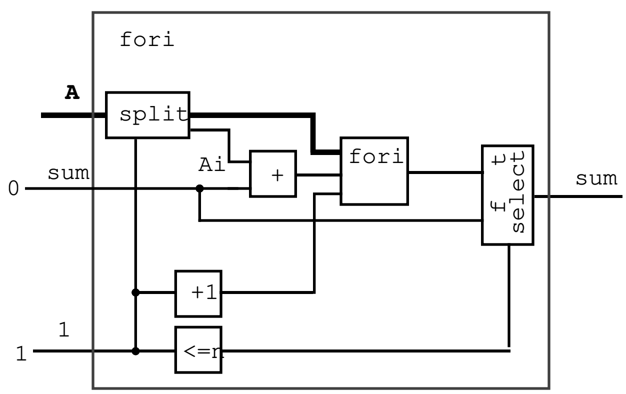
Note that there can be two versions of a loop, horizontal or vertical. A vertical example:

```
sum := 0;
for i := 1 to n do sum := sum + A[i];
```

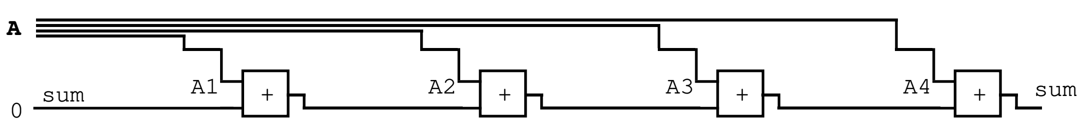
which is constant equivalent to:

```
sum := 0;
i := 1;
if i <= n then begin
    sum := sum+A[i];
    for i := 2 to n do sum := sum + A[i];
    end
```

which is the circuit:



Of course, if n is ever fixed then the general infinite circuit immediately reduces to what is expected, eg, n=4:



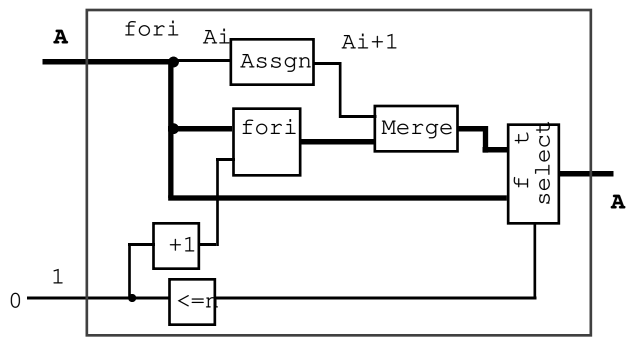
A horizontal example:

```
for i := 0 to n do all A[i+1] := A[i];
```

which is:

```
i := 0;
if i<n do begin all
    A[i+1] := A[i];
    for i:= 2 to n do A[i+1] := A[i];
    end;
```

Which is the circuit:



Again, if n is fixed this is constant equivalent to what is expected.

Arrays

We have to ascribe sensible meaning to array elements when the index cannot be expanded to a constant. It seems reasonable to regard A[v] in such cases as a multiplexor that receives all A and a selector v as input but produces but one element as output. A multiplexor may be defined in terms of simpler circuits although it may be better considered as an element.

The use of a variable index as a destination is likewise a demultiplexor. However, the use of a demultiplexor in a loop can give rise to some problems. For example, the following cannot be regarded as well defined in the circuit model (although there is a sensible interpretation possible in this particular case):

```
procedure reverse (A[1:*],B[1,*],n)
begin
    for i from 1 to n do all
        B[n+1-i] := A[i];
end;
```

because the lhs of the assignment is repeated on every iteration. The versions of the above without "all" or with the assignment reversed do have well defined meanings. For example:

```
procedure reverse (A[1:*],B[1,*],n)
begin
    for i from 1 to n do all
        B[i] := A[n+1-i];
end;
```

The above, although a simple algorithm serially, and a mere wire relabelling when n is constant, still represents a serious algorithm when regarded as expressing a general parallel concept (and would be expensive to implement as a circuit in practice).

4. Relationship to Other Models

Let's summarise where we are at present. We have a model for parallel algorithms which is that they are circuits. We also have a textual notation for expressing such algorithms. This gives us a compact way of describing the algorithms underlying circuits. It also allows us to write parallel algorithms, with no constraint on resources, and have a well defined way of figuring out what such algorithms mean.

Another way of looking at things is that we have one class of object that we are dealing with, the *algorithm-circuit* say. Each algorithm-circuit may be regarded as an algorithm or as a circuit. Some are more naturally regarded one way or the other but there is one class only.

Although we claim that the circuit model is useful, we are not proposing that the textual language is a useful one in which to program in practice. It would certainly be possible to use the language for programming (implemented in a manner similar to data flow languages) but there would be many practical difficulties in handling the generality involved. We may construct many circuits that appear to have properties that are too bizarre to be algorithms, for example algorithms with values that never stop changing yet have well-defined external stability.

From the programming point of view, the set of objects that we are dealing with is perhaps too large. A much better behaved class of circuits are those that are called *combinational*. Taking this to mean circuits that have no feedback[6], it is clear that such circuits have much more restricted properties. In particular, combinational circuit-algorithms may be executed/simulated in such a way that each wire is assigned a value only once.

Single Assignment Model

This relates the class of combinational circuits to the single-assigment algorithms, which have been claimed as being particularly simple and worthy of attention [AMBL90]. A single assignment algorithm is one that is written as if it were to be executed serially but all calls on procedures are initiated as parallel operations. The algorithm must obey the rule that any individual variable may be assigned to once, only. A variable may be referenced many times but where a procedure encounters a variable that has not yet been assigned to, it will be held pending until the variable receives its value.

For example, the insertion sort, taken from [THOR92], but expressed in an informal notation :

[6]No feedback implies combinational but not the other way round (HUFF71). However, in practise there is little use for circuits that are combinational and have feedback.

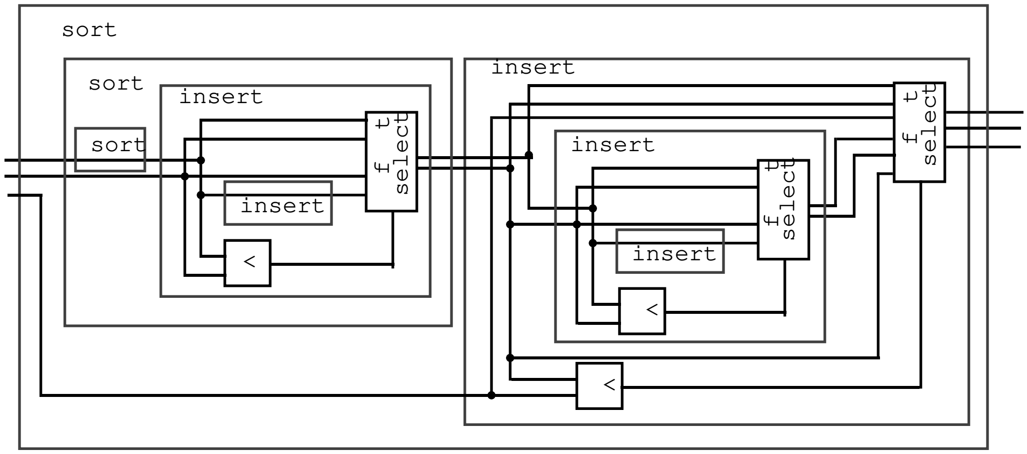
```
procedure sort(A[1..n],var B[1..n]);
begin
   if n>1  then
       begin
           sort(A[1..(n-1)],C[1..(n-1)]);
           insert(A[n], C[1..(n-1)], B[1:n]);
       end
   else B[1] := A[1];
end;

procedure insert(x, A[1..m], var B[1:(m+1)])
begin
   if m=0 then B[1] := x
   else
       if x<=A[1]  then B[1..m+1] := x, A[1..m]
                   else begin
                        B[1] := A[1];
                        insert(x, A[2..m], B[2:m+1]);
                        end;
end;
```

The actual course of parallel execution depends on the timing involved, but if sort(A[1:3]) were to be initiated and applied to [6,5,7], for example, the following sequence of events could result. Time flows from left to right and the vertical direction represents parallel activity - each procedure is asumed to be taken in one instant to the next procedure call or to where it is stuck waiting for an assignment (shown by ?).

```
[6,5,7]->A[1:3]
   sort(A[1:3],B[1:3])
      sort(A[1:2],C[1:2])
         sort(A[1:1],D[1:1])
            6->D[1:1]
            insert(A[2], D[1:1], C[1:2])
               ?D[1]..[5,6]->C[1:2]
      insert(A[3],C[1:2],B[1:3])
         ?C[1].................... 5->B[1]
                                     insert(A[3],C[2:2],B[2:3])
                                        6->B[2]
                                        insert(A[3],C[2:1],B[3,3])
                                           7->B[7]
```

The algorithm as written may equally be regarded as a combinational algorithm circuit, for the case of length 3 arrays, constant equivalent to:



Although the models are poles apart, the single assignment model gives the same meaning and also shows one of the possible ways that the circuit could be simulated reasonably efficiently, i.e how a text algorithm with variable parameters could be executed in practice.

The combinational circuit algorithm and the single assignment language are of the same extent. However, it is somewhat easier in our notation to express algorithms in a natural form. For example, the body of the sort example could be expressed as:

```
for i := 2 to n do insert(A[i],A[1..(i-1)],A[1..i]);
```

With further massaging, the whole recursive algorithm is identical to:

```
for i := 2 to n do
   {insert A[i] in A[1..(i-1)]}
       begin
       X:=A;
       for j from i-1 downto 1 do
           X[j..i] := select(  A[i]<A[j],
                               (A[i],A[j..i-1]),
                               X[j..i]);
       A:=X;
       end;
```

So the recursive form of [THOR92] has an inherently sequential aspect as a circuit, a rather slow O(nxn) algorithm. No doubt the single assignment model could be adapted to allow expression in a similar form but it is doubtful whether it would be regarded as being *the* identical algorithm.

However, because the circuit model does not need to rely on recursion, it could express a parallel insertion sort as a better and more straightforward algorithm as follows:

```
for i := 2 to n do
    {insert A[i] in A[1..(i-1)]}
        begin
        for j from 1 to i-1 do
            cond[j] := A[i]<A[j];{cond[0] initlzed to false}
        for j from 1 to i-1 do
            if cond[j] then
                A[j] := (if cond[j-1]  then A[j-1] else A[i]);
        end;
```

In this case the fully expanded inner loops have no connections between the iterations so the algorithm is clearly O(n), when n is a constant (ignoring fan-in and fan-out).

Functional Programming

Another model of parallel algorithms is functional evaluation [HUDA89]. With this the example above would have to be expressed something like:

```
function sort(A[1..n]):B[1..n];
    select( n>1, insert(A[n], sort(A[1..(n-1)]), B[1:n]),A[1];

function insert(x, A[1..m]):B[1:(m+1)];
    select( m=0,
            x
            select( x<=A[1],
                    [x, A[1..m]],
                    [A[1],insert(x, A[2..m], B[2:m])]))
```

The semantics here is that all arguments are evaluated in parallel, called recursively, until finally a value is obtained. Whenever an array is encountered as an argument or a result of a function an entire array is assigned.

It is pretty clear that any functional program may be modelled as one of our circuits; our circuits, like the course of functional evaluation, may be infinite in extent[7]. Although the functional notation is fairly restrictive it could be extended to give means of expression like can be used with circuits and single assignment. So it does appear that the functional model, single assignment, and combinational ciruit-algorithms have the same expressive power (as does the general data flow model).

The functional and circuit models are, in a sense, extremely pure in that they ascribe meaning but do not prescribe how a program is to be executed in practice. The single assignment model seems, like dataflow, to be much more oriented towards efficient

[7]We haven't considered the complications that arise when unevaluated functions are passed as arguments, but these also can be forced into line by extending circuits in a similar manner.

execution. This, presumably, is a plus in some circumstances, but it does mean that the model is limited in its range of applicability.

The other advantage of single assignment is that sequentiality not implied by data flow is inherent. With the other models it is difficult to ensure sequential behaviour (though with our textual notation, and similar notations for dataflow, one can regard an algorithm as being serial in order to aid understanding with no loss of meaning other than to considerations related to implied performance).

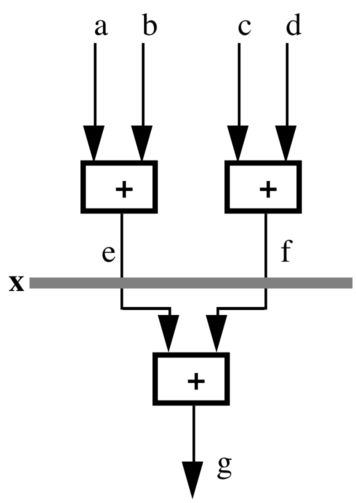
5. Barrier Circuits

This leads us to dealing with the main deficiency of the circuit-algorithm model. Although it is great for expressing parallel ideas when there is no shortage of resources, or where time is not involved, the model is of little help when encountering real limitations. Just as in practice there is a need to introduce a new element, the latch, to describe synchronous circuits, we need to also extend our model.

There are many aways that we could extend the model. However, it would be nice to have an extension that is close to extensions made for serial programming as well as close to the way that circuits are extended in practice. This leads us to consider the *data barrier* which is the analogue of the control barrier and also analogous to the latch point.

Informal Definition

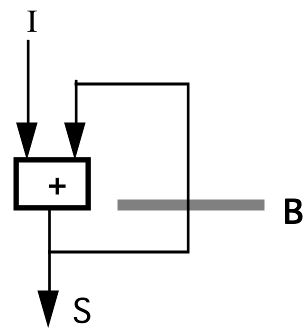
We introduce to our notation the concept of a *data barrier*. In the circuit this is shown as a line, perhaps labelled, across which some wires may pass. The corresponding text notation is straightforward.




```
begin
   e := a+b;
   f := c+d;
   barrier x(e,f);
   g := e+f;
end
```

Informally, the meaning of a barrier can be explained in terms of the circuit becoming externally stable with no signals being allowed to cross any barrier. Once the circuit has become stable the outputs may be recorded, then values may cross barriers and the inputs may be changed. Then barrier crossings are inhibited and circuit is allowed to become stable again, etc. In a sense, the barriers divide the circuit into *regions* that operate autonomously. The meaning of the circuit is extended to involve the effect of sequences of input values over time. In the above example groups of four numbers will be summed in two stages.

The introduction of barriers brings both sequentiality in time and memory. With barriers feedback may be given meaning where it otherwise would be undefined. For example, the following algorithm will produce as outputs the running sums of the sequence of inputs, assuming that the sum is initialised to zero.




```
begin
   barrier B(S from L);
   S := I + S;
   L:
end
```

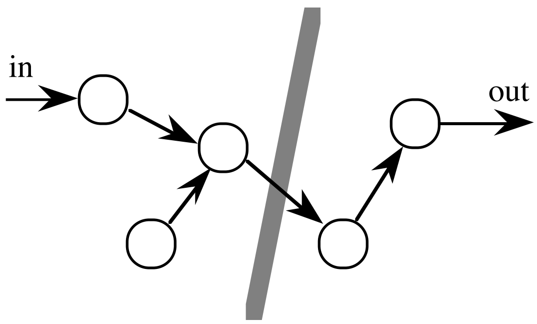
Precise definition

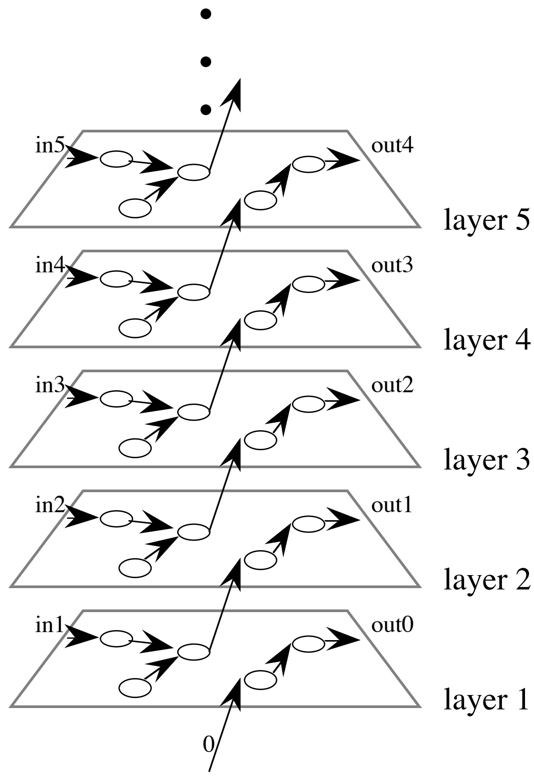
The informal definition is adequate for many purposes but it is incomplete in that there may well be circuits that have feedback both through barriers and within regions. The concept of a region is very ill defined. It is also distasteful in that it requires that we be able to recognise stability before changing inputs - the model without barriers does not guarantee stability in any limited time and it is dificult to determine external stability[8].

The solution that we propose for defining the meaning precisely is to replace sequence in time with sequence in space. A circuit is expanded into three dimensions and a barrier represents a connection from a lower to a higher level. The effect obtained by a sequence of inputs over time to the barrier circuit is replaced by constant inputs made to successive levels. The sequence of outputs over time corresponds to the outputs at successive levels when the three dimensional circuit reaches stability. The diagram overleaf illustrates the idea.

The complexity of the situation has been greatly diminished. It is possible for the circuit on any level to not be stable internally but still have a well defined meaning. If, as is usual, the circuit, without barrier crossings, is combinational, the meaning of a circuit that involves barriers as memory elements has been explained in a combinational (single assignment, functional) form.

[8]It sounds like an unsolvable problem given that circuits may reach external stability yet be continually changing internally.





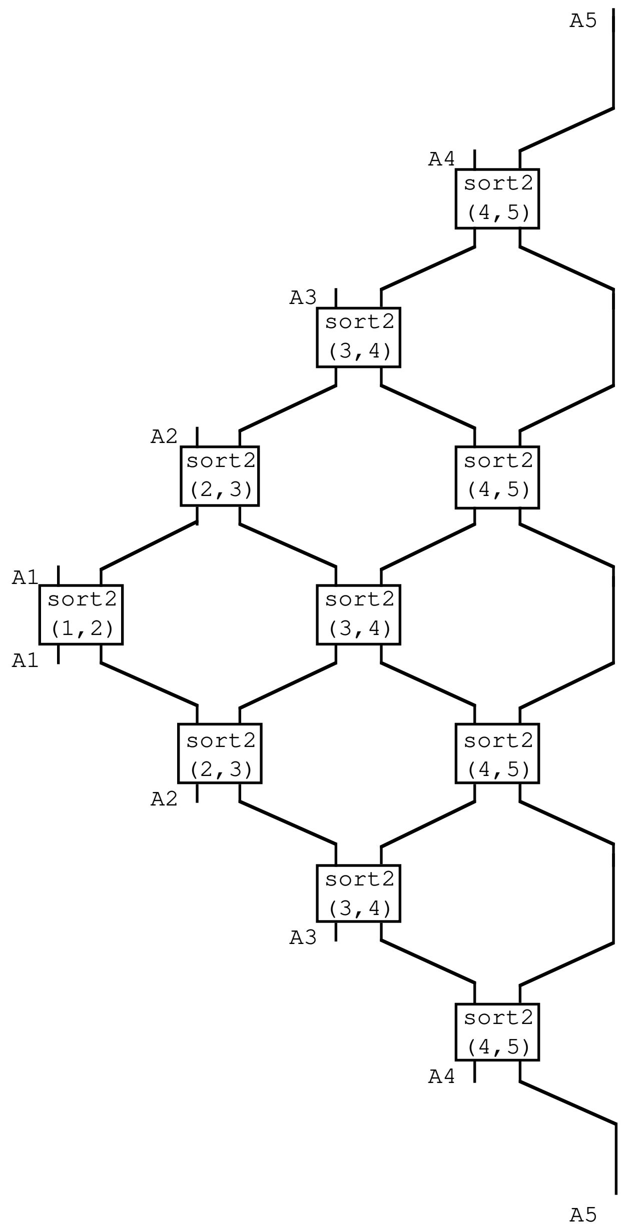
Reuse of resources via Data Barriers

Data barriers can be used to introduce memory and to impose sequentiality. They also provide a means of expressing algorithms that are designed to re-use resources. It is clear that pipelined algorithms can be written with barriers. A more interesting application is to express systolic arrays [KUNG82] in a manner that relates them to the underlying parallel algorithms.

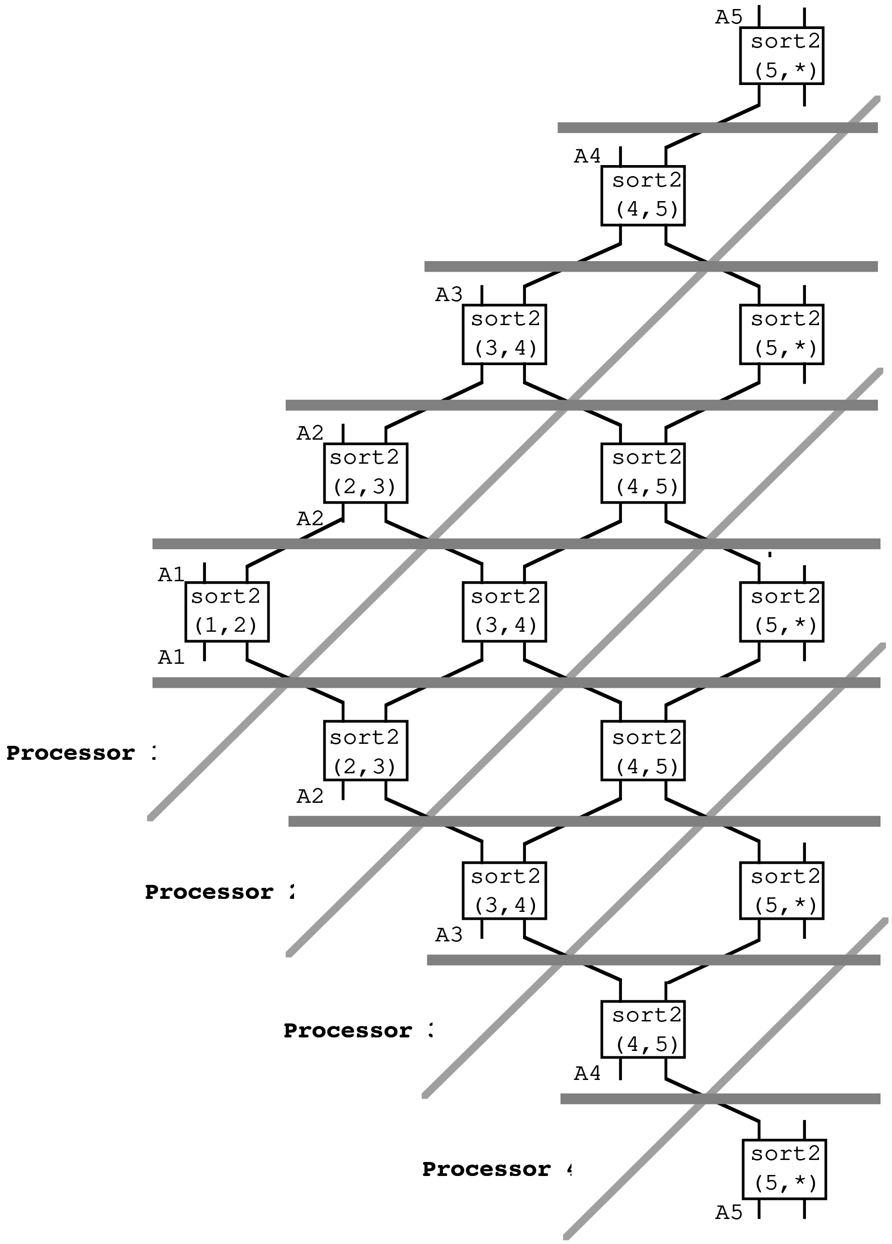
Lets take as an example the old faithful bubble sort. For a length-5 array we could express this serially as follows, where sort2 is a procedure that compares and conditionally swaps its two inputs :

```
for j from 4 downto 1 do
     for i from j to 4 do
          A[i],A[i+1] := sort2(A[i],A[i+1])
```

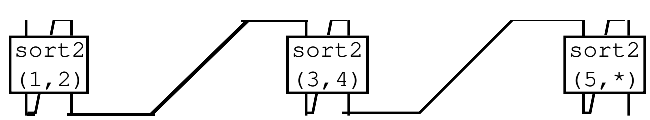
Regarding this as a serial expression of a parallel algorithm it represents the following circuit:



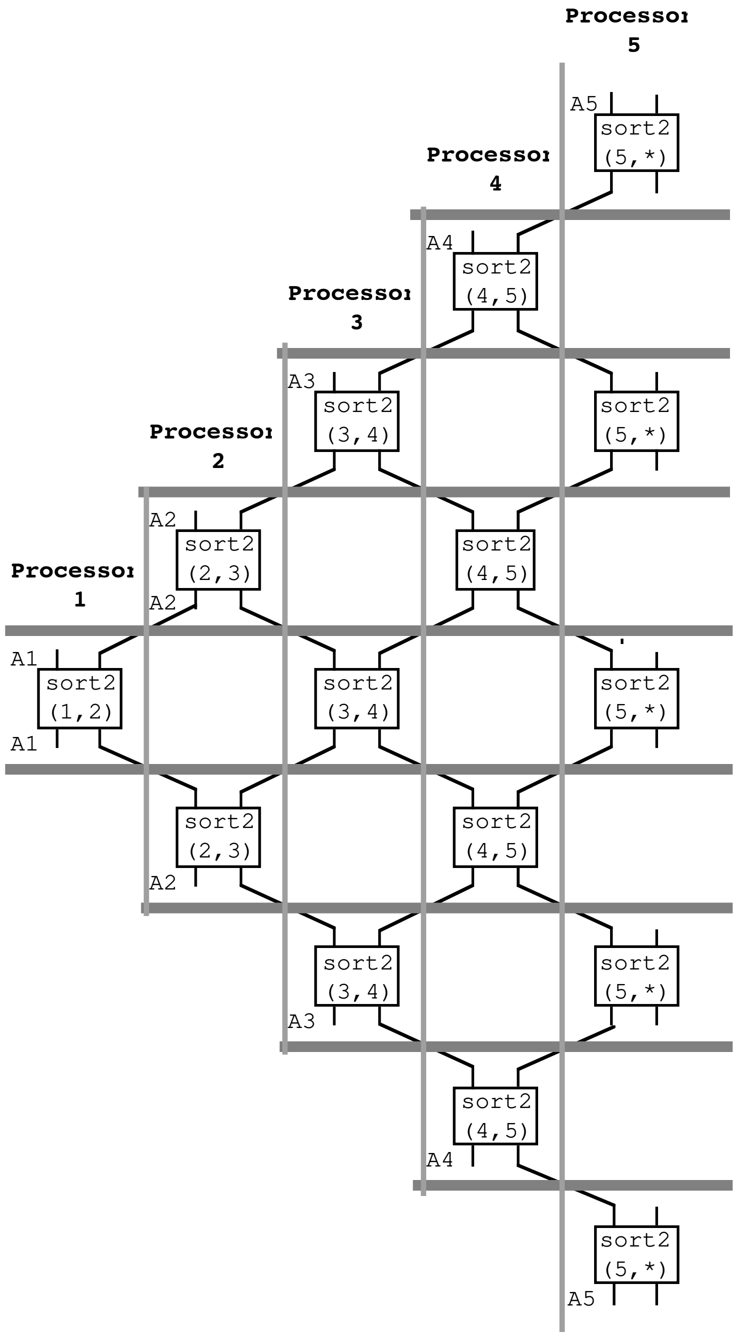
Without going into details, one can immediately adapt such a circuit by inserting barriers so that each region has many independent operations. Once the barriers are in place it is possible to assign the work to processors to create a systolic algorithm. In the following diagram the data barriers are the horizontal lines. With the assignment of work to processors as shown by the dashed lines this represents an algorithm where the data is presented to the first processor serially and the results appear in parallel at the outputs of the processors:



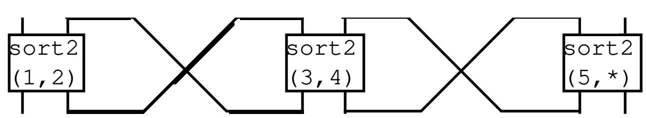
This would require the connections between the processors to be made as follows:



The same algorithm can be assigned to processors in a different fashion:



This requires the following interconnection:



It is perhaps surprising that this is the parallel odd-even sort. The fact that it would be equivalent to a bubble sort is not immediately obvious.

6. Conclusion

We came into this study knowing that algorithms could often be very useful in describing circuits [DORA74}. We are now also convinced that circuits can be very useful for describing highly parallel algorithms. We have pushed the relationship of circuits as algorithms further than in the past by seeing how any circuit may be regarded as an algorithm and vice versa. It is gratifying that we can take the relationship so far, even if it does require us to accept potentially infinite circuits. It is also reassuring that the well-behaved subset of circuits that are combinational seem to be the same set of algorithms that have been scoped by other models.

It is also interesting that we seem to have a natural meaning for an algorithmic analogue of the latch, which is the data barrier. This has been defined simply and is clearly useful in a number of circumstances. However, we have yet to explore the extent to which the data barrier can be applied. It would be nice to be able to use it to express simply algorithms (such as the dining philosophers) that are quite easy to express by more conventional means, but we have not made much progress in this direction yet.

References

[ALMA89] Almasi, G.S. & A. Gottlieb. Highly Parallel Computing. Benjamin/Cummings, 1989.

[AMBL90] Ambler, A.L. & M.M. Burnett. Visual Forms of Iteration that Preserve Single Assignment, Journel of Visual Languages and Computing, No. 1, pp 159–181, 1990

[BATC68] Batcher, K.E. Sorting Networks and their applications. AFIPS Spring Joint Computing Conference, No. 32, pp 307–314, 1968

[BELL92] Bell, G. Untracomputer - a teraflop before its time. CACM August 1992, Vol. 33, No. 8, pp 27-47.

[CHAN88] Chandy, K. M. Parallel program design: a foundation. Addison Wesley. 1988.

[DAVI82] Davis, A.L. & R.M. Keller, Data Flow Program Graphs, IEEE Computer, Vol. 15, No. 2, pp 26–41, February 1982

[DIJK89] Dijkstra, E. On the Cruelty of really teaching computer science. CACM December 1989, Vol. 30, No. 12, pp. 1398-1404.

[DORA74] Doran, R. W. & Thomas, L. A. Recursive algorithms in combinational circuit design. Massey University Computer Unit. Report #13, 1973.

[FELD92] Feldman, Y. & Shapiro,*Spatial machines: A more realistic approach to parallel computation*. CACM October 1992, Vol 35, No. 10, pp 61-73.

[HOAR78] Hoare, C.A.R. Communicating Sequential Processes. Communications of the ACM. Vol 21, No. 8, pp 666–677, August 1978.

[HUDA89] Hudak, P. Conception, Evolution, and Application of Functional Programming Languages. ACM Computing Surveys, Vol. 21, No. 3, pp 359–411, September 1989

[HUFF71] Huffman, D. A. Combinational circuits with feedback. In Recent developments in switching theory. A. Mukhopadhyay (Ed.). Addison Wesley, New York, 1971.

[KUNG82] Kung, H.T. Why Systolic Architectures? IEEE Computer, Vol. 15, No. 1, pp 37–46, January 1982

[SABO88] The paralation model: architecture independent programming. MIT Press. 1988.

[SHU88] Shu, N.C. Visual Programming. Van Nostrand Reinhold Company, 1988

[ULLM84] Ullman, J.D. Computational Aspects of VLSI, Computer Science Press, 1984.

[THOR92] Thornley, J. A Collection of Compositional Ada Example Programs, 1992

[VALI90] Valiant, L.G. A Bridging Model for Parallel Computation, Communications of the ACM, Vol. 33, No. 8, pp 103–111, August 1990.