ב״ה

Flexible Algorithms in Education - An Experience Report

Experiences about a course for beginners
developed at Jerusalem College of Technology.

R.B. Yehezkael (Formerly Haskell) December 2011 - טבת תשע״ב

E-mail: rafi@jct.ac.il Home page: http://homedir.jct.ac.il/~rafi

Many thanks to E. Dashtt, E. Gensberger, M. Goldstein.

Introduction

First courses in computer science usually deal with sequential algorithms and programs. This has unfortunate consequences in that the student's mind becomes accustomed to a sequential way of thinking. This makes it hard for him to later understand and utilize parallelism effectively.

Parallel programming on the other hand is very hard and so it is not practical to introduce these concepts at an early stage.

So we have developed flexible algorithms and a course for beginning students of computing science.

What are flexible algorithms?

Background to course development

Simple Flexible Language – SFL, prototype compiler.

First version in Hebrew (JCT) - notationally too complex.

Second version in Hebrew (JCT) - notationally simpler.

Third version rewritten in English and expanded.

Companion programming course.

A total of one lecture hour and one exercise hour were allocated for the course given at JCT. The latest version may need more time allocated.

Educational Approach

Emphasis on flexible algorithms - early awareness of parallelism

Reading - Executing - Understanding

Converting flexible algorithms to hardware block diagrams (a.c.)

Converting flexible algorithms to sequential algorithms (t.r.)

Computational Induction - Deeper understanding

Changes to (and writing) flexible algorithms

Overall description of systems using flexible algorithms (a.c.)

Other Approaches

This is not the first attempt to use a non-sequential approach for beginning students of computer science.

At Imperial College London beginners have been taught the functional language HASKELL and the logic programming language PROLOG, followed by sequential programming in JAVA.

At Massachusetts Institute of Technology, the SCHEME dialect of LISP has been taught in a first course. Now, Python is taught in a first course.

M. Paprzycki and co-workers have published a suggestion to teach sequential programming as a specific case of parallel programming.

Reactions to our approach at Jerusalem College of Technology

Some colleagues thought that such a course should be taught after students had mastered sequential algorithms and programs. Perhaps they were right about this regarding the first version of the course but they were wrong about this regarding the second version of the course. Indeed, after the college instituted a common first semester, our course was deemed not of sufficient interest to all departments and was moved to the second semester, after the course on sequential algorithms and programs. Our course was then found to be too easy, confirming that the course material is suitable for beginners.

Another colleague thought that such a course should be taught before the course on sequential algorithms and programs.

Some student comments

1) This course is superfluous.

2) This course is better than the companion course on sequential algorithms and programs as there is the possibility of parallel execution.

3) In time everyone will know languages such as C++, JAVA. The knowledge of this kind of material regarding parallelism is an advantage in finding work.

4) It is amazing that a hardware block diagram of a serial adder can be derived from a flexible algorithm for addition.

Conclusion

Declarative notation with an algorithmic style.

Notationally simple and multifaceted.

Early awareness of parallelism.

Conversion to sequential algorithms.

Simple hardware block diagrams and overall system description.

Broaden outlook of students - important in a first CS course.

Further Work

Embedded Flexible Programming Language (EFL):

- Scientific computation
- Hardware definition languages

Further develop educational material:

- Integrated course on flexible algorithms and digital logic
- Course for secondary schools
- Advanced course on flexible algorithms

Flexible Algorithms

Stories: Shopping with a list, etc.

Functional form: Parameters for values received.

Parameters for values returned.

(IN, OUT but no INOUT variables.)

Function calls and compositions.

Set of statements
- once only assignment, conditionals.

Examples of errors

Syntax:

```
function x', y' •= bug1(x, y);
{
  x•=y+1;          // x may not be assigned
  y'•=x'+1;        // value of x' not accessible
}
```

Run time:

```
function x' •= bug2(x);
{
 if (x ≤ 3)   x'•=x+1;        // when x is 3
 if (x ≥ 3)   x'•=x+2;        // there is a conflict
}
```

Example without errors

Specific solution for reversing five elements using three functions.

function v' •= reverse(v, low, high);
{
if (low<high)
  {v'_{high} •= v_{low};
  v' •= reverse (v, low+1, high-1);
  v'_{low} •= v_{high};}
else if (low=high)
    {v'_{high} •= v_{high};};
} // end reverse

Should an "else" be used in the above definition?

Execution Methods

1. Parallel

2. Sequential with immediate calls.

3. Sequential with delayed calls.

All methods presented in (multi) set form:

set of statements ⟶ values of results.

Example of execution methods

Suppose that v=(1,2,3,4,5) and we wish to execute:

r' •= reverse (v,0,4);

Parallel execution

set of statements	r'
{ r' •= reverse (v, 0, 4); }	(_, _, _, _, _)
{ r'_4 •= v_0; r' •= reverse (v, 1, 3); r'_0 •= v_4; }	(_, _, _, _, _)
{ r'_3 •= v_1; r' •= reverse (v, 2, 2); r'_1 •= v_3; }	(5, _, _, _, 1)
{ r'_2 •= v_2; }	(5, 4, _, 2, 1)
{ }	(5, 4, 3, 2, 1)

Sequential execution left to right with immediate execution of the function call at left

set of statements	r'
{ r' •= reverse (v, 0, 4); }	(_, _, _, _, _)
{ r'_4 •= v_0; r' •= reverse (v, 1, 3); r'_0 •= v_4; }	(_, _, _, _, _)
{ r' •= reverse (v, 1, 3); r'_0 •= v_4; }	(_, _, _, _, 1)
{ r'_3 •= v_1; r' •= reverse (v, 2, 2); r'_1 •= v_3; r'_0 •= v_4; }	(_, _, _, _, 1)
{ r' •= reverse (v, 2, 2); r'_1 •= v_3; r'_0 •= v_4; }	(_, _, _, 2, 1)
{ r'_2 •= v_2; r'_1 •= v_3; r'_0 •= v_4; }	(_, _, _, 2, 1)
{ r'_1 •= v_3; r'_0 •= v_4; }	(_, _, 3, 2, 1)
{ r'_0 •= v_4; }	(_, 4, 3, 2, 1)
{ }	(5, 4, 3, 2, 1)

Sequential execution left to right with delayed execution of the function call at left

set of statements	r'
{ r' •= reverse (v, 0, 4); }	(_, _, _, _, _)
{ r'_4 •= v_0; r' •= reverse (v, 1, 3); r'_0 •= v_4; }	(_, _, _, _, _)
{ r' •= reverse (v, 1, 3); r'_0 •= v_4; }	(_, _, _, _, 1)
{ r' •= reverse (v, 1, 3); }	(5, _, _, _, 1)
{ r'_3 •= v_1; r' •= reverse (v, 2, 2); r'_1 •= v_3; }	(5, _, _, _, 1)
{ r' •= reverse (v, 2, 2); r'_1 •= v_3; }	(5, _, _, 2, 1)
{ r' •= reverse (v, 2, 2); }	(5, 4, _, 2, 1)
{ r'_2 •= v_2;}	(5, 4, _, 2, 1)
{ }	(5, 4, 3, 2, 1)

Parameter Passing Styles

Function: *v' •= reverse (v, low+1, high-1)*

Assignment style
(Changes only): *reverse (low•=low+1; high•=high-1)*

IMPORTANT: The values of low and high are not changed by the statements low•=low+1, high•=high-1. There are separate variables for each call or activation of a function.

Conversion to a sequential algorithm

```
function v' •= reverse(v, low, high);
{
if (low<high)
   {v'high •= vlow;
     reverse (low•=low+1; high•=high-1);  // Note style change
     v'low •= vhigh;}
else if (low=high)
          {v'high •= v high;};
} // end reverse
```

```
reverse: {if low<high
           {              // Need temporary variable(s).
            v_high:=v_low;
            low:=low+1; high:=high-1;
            goto reverse;
            v_low:=v_high    // Statement unreachable.
           }
          else    // Unnecessary and inefficient.
           if low=high
              v_high:=v_high
         }

// result is given in v itself
```

reverse: { if low<high
{
$new_vlow := v_{high}$; $new_vhigh := v_{low}$;
$v_{low} := new_vlow$; $v_{high} := new_vhigh$;
low:=low+1; high:=high-1;
goto reverse;
}
}

```
while low<high
{
  new_vlow:=v_high;    new_vhigh:=v_low;
  v_low:=new_vlow;    v_high:=new_vhigh;
  low:=low+1;    high:=high-1;
}
```

Possibility of doing things in parallel in the above.

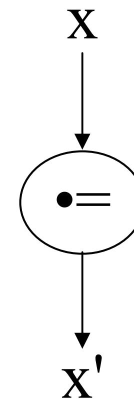
Solution using one temporary variable.

while low<high
{
  temp:=v_{high};
  v_{high}:=v_{low};
  v_{low}:=temp;   // temp holds the previous value of v_{high}
  low:=low+1;   high:=high-1;
}

Fewer possibilities for doing things in parallel.

Hardware block diagrams

Circuit for performing a copy operation x' •= x.



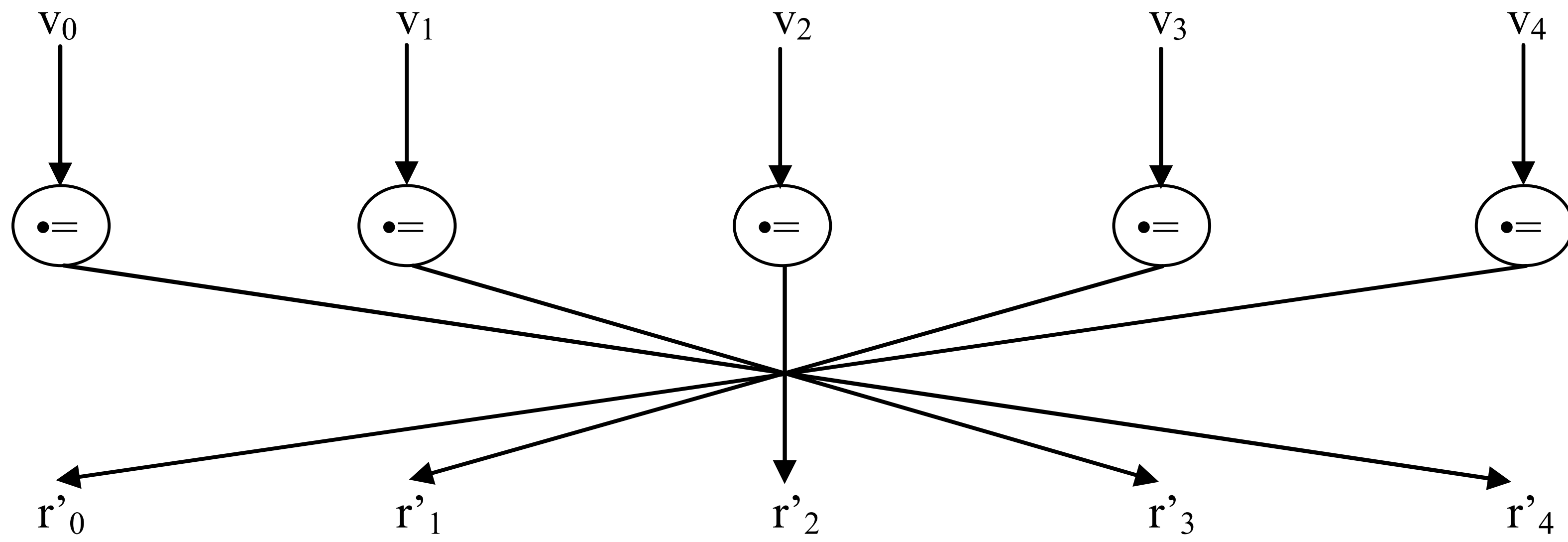
Develop a block diagram of a circuit for reversing first five elements of v putting the result in r'.

{ r' •= reverse (v, 0, 4); }

≡ { r'_4 •= v_0; r' •= reverse (v, 1, 3); r'_0 •= v_4; }

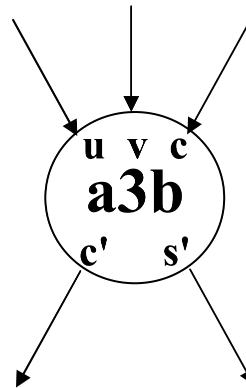
≡ { r'_4 •= v_0; r'_3 •= v_1; reverse (v, 2, 2); r'_1 •= v_3; r'_0 •= v_4; }

≡ { r'_4 •= v_0; r'_3 •= v_1; r'_2 •= v_2; r'_1 •= v_3; r'_0 •= v_4; }


v0
v1
v2
v3
v4
•=
•=
•=
•=
•=
r'0
r'1
r'2
r'3
r'4

Another hardware block diagrams

Assume there is hardware operation "a3b" for adding three digits (or bits) giving their carry and sum respectively (i.e. a full adder).



Flexible algorithm for adding two vectors of digits u, v with a (previous) carry digit c, and giving results c' the resulting carry and s' the sum of the vectors of digits u, v. Here n is the lowest digit position, where the high order digit has position zero.

```
function c', s' •= add (u, v, c, n);
{
 if (n>0)
 add( c, s'n•= a3b(un, vn, c); n•=n-1);
 else c', s'0•=a3b(u0, v0, c);
} // add
```

Example of parallel execution of c',s' •= add(123, 987, 6, 2).

set of statements	c' s'
u v c n	
{ c',s' •= add(123, 987, 6, 2) }	_, _, _, _
{ c',s' •= add(123, 987, 1, 1) }	_, _, _, 6
{ c',s' •= add(123, 987, 1, 0) }	_, _, 1, 6
{ c', s'_0•=a3b(1, 9, 1) }	_, _, 1, 6
{ }	1, 1, 1, 6

Develop a block diagram of a circuit for a 4 digit serial adder.

$\{c',s'\bullet\!=add(u,v,c,3);\}$

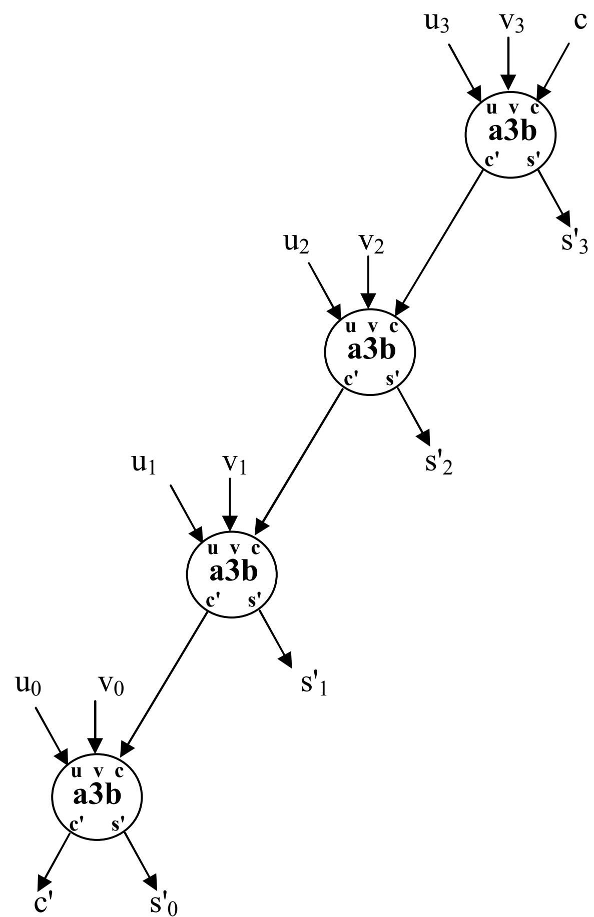
$\equiv\{add(c,s'_3\bullet\!=a3b(u_3,v_3,c);n\bullet\!=2);\}$

$\equiv\{add(c,s'_2\bullet\!=a3b(u_2,v_2;c,s'_3\bullet\!=a3b(u_3,v_3,c));n\bullet\!=1);\}$

$\equiv\{add(c,s'_1\bullet\!=a3b(u_1,v_1;c,s'_2\bullet\!=a3b(u_2,v_2;c,s'_3\bullet\!=a3b(u_3,v_3,c)));n\bullet\!=0);\}$

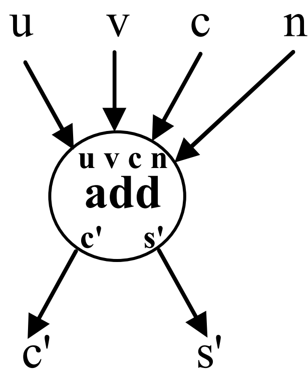
$\equiv\{c',s'_0\bullet\!=a3b(u_0,v_0;c,s'_1\bullet\!=a3b(u_1,v_1;c,s'_2\bullet\!=a3b(u_2,v_2;c,s'_3\bullet\!=a3b(u_3,v_3,c))));\}$

Represent last line diagramatically to obtain block diagram of a serial adder


u3
v3
c
u v c
a3b
c' s'
s'3
u2
v2
u v c
a3b
c' s'
s'2
u1
v1
u v c
a3b
c' s'
s'1
u0
v0
u v c
a3b
c' s'
c'
s'0

Diagrammmatic development

A function call c', s'•=add(u, v, c, n) is represented by:



This diagram is equivalent to one of the following depending whether or not n>0.

Diagram for n>0

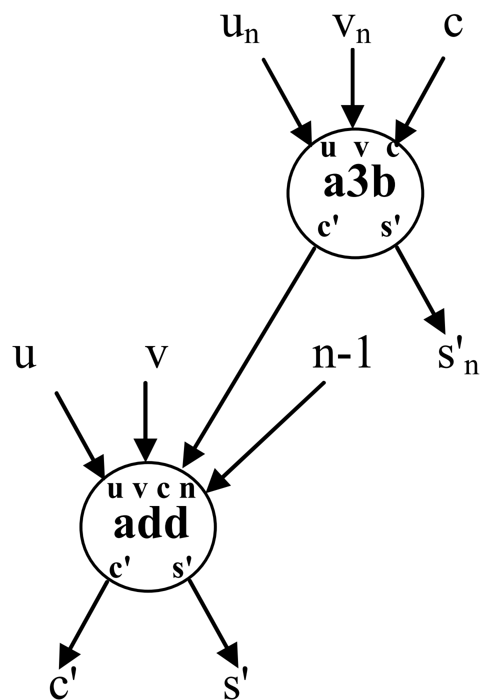
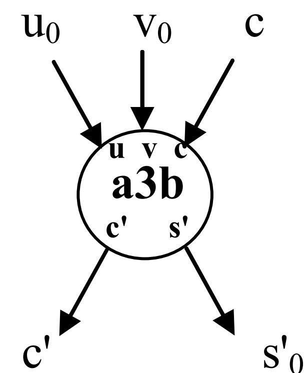
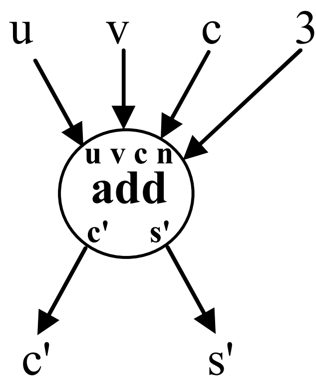


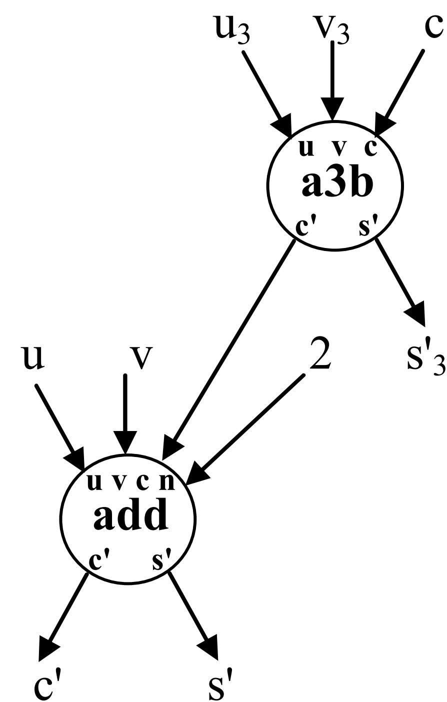
Diagram for n=0



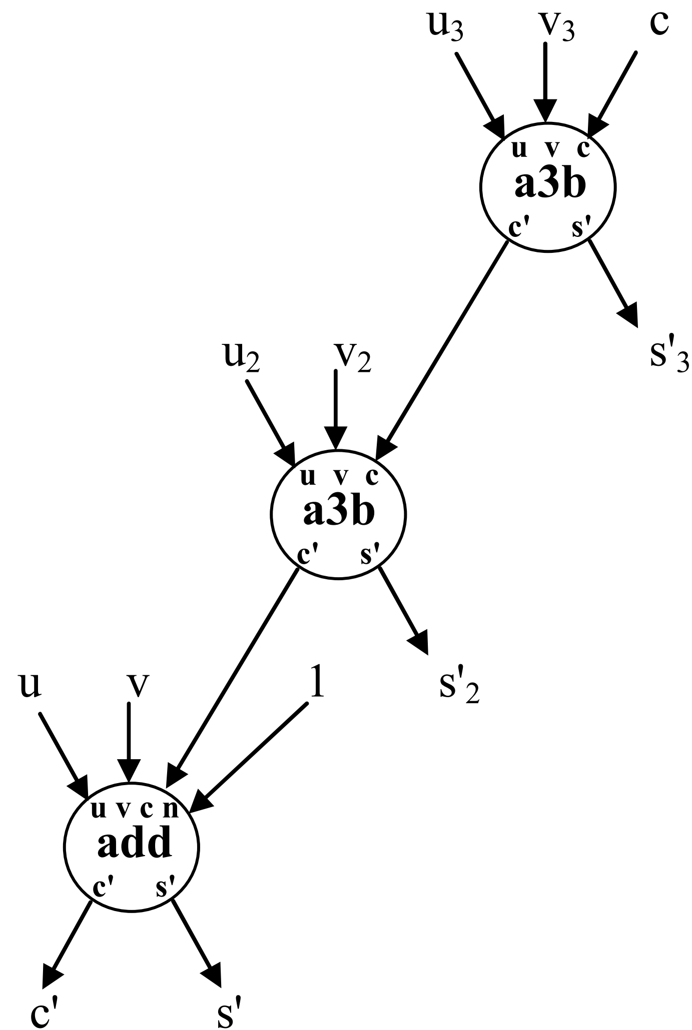
The set {c', s'•=add(u, v, c, 3);}
is represented by:



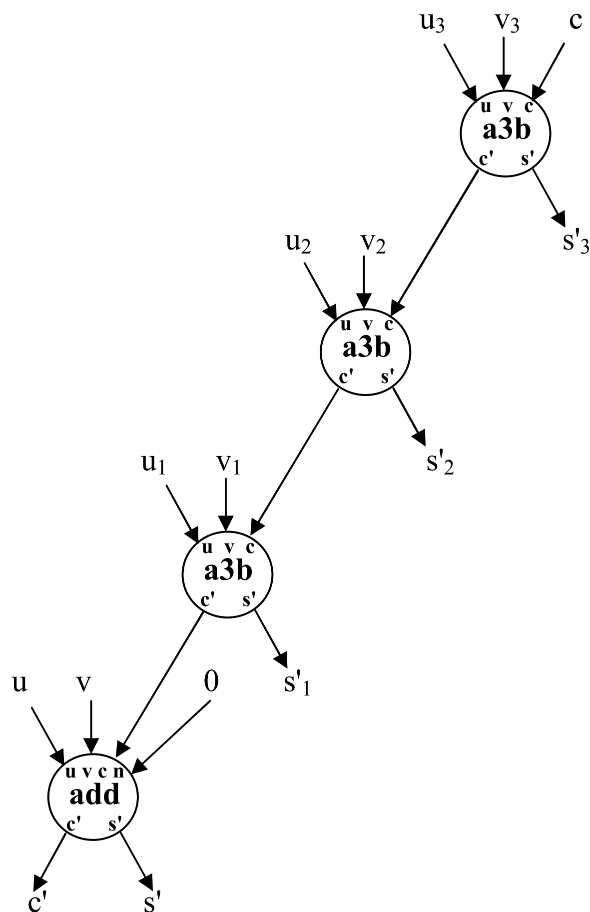
Equivalent to:



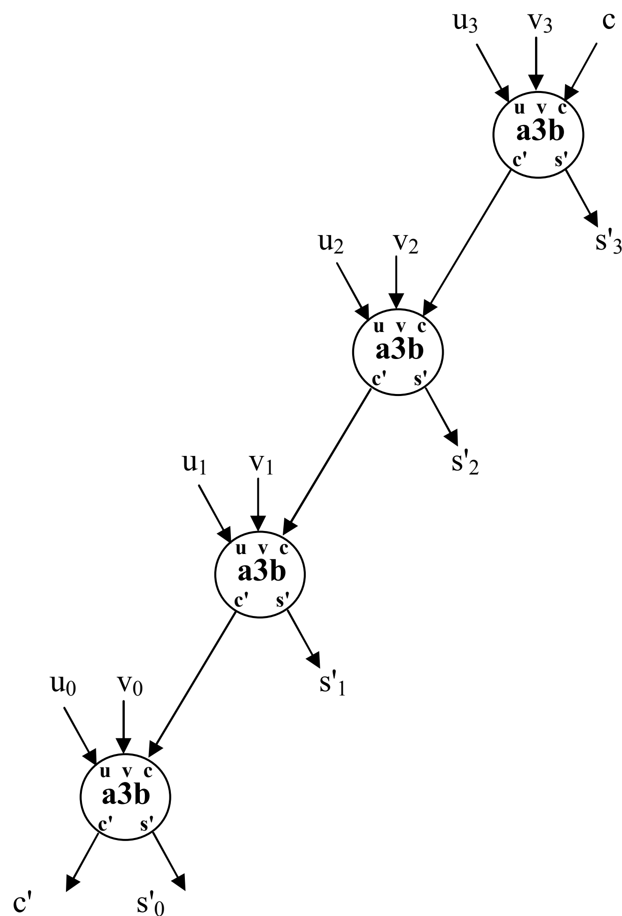
Equivalent to:



Equivalent to:



Finally giving:



Obtained diagrammatically, a block diagram of a serial adder

Computational induction

We found this proof technique easiest to use and we point out its limitation that it does not prove termination of execution. Some measure of confidence that execution terminates is obtained by executing the algorithm on example data (by hand).

We also mention that in the general case, "proof of termination" is a provably unsolvable problem.

Unlabeled "let block"

```
function r' •= f(x, y, z);
{
// statements of f
… x+y-z…
…
… x+y-z…… x+y-z…
…
}
```

```
function r' •= new_f(x, y, z);
{
r'•=block(x, y, z, x+y-z);    // equivalent to   r'•=block(xyz•=x+y-z);
}

function r' •= block(x, y, z, xyz);
{
// statements of f with xyz in place of x+y-z
… xyz…
…
… xyz…… xyz…
…
}
```

```
function r' •= new_f(x, y, z);
{let xyz •= x+y-z;     // Give local or internal variable xyz its value.

// statements of f with xyz in place of x+y-z
… xyz…
…
… xyz…… xyz…
…
}
```

Labeled "let block"

```
function v' •= squares (v);
{ loop(i•=0); };    // end squares

function v'•=loop(v, i);
{ if (i< length of v)
    { loop(i•=i+1); v'_i•=v_i×v_i; };
};    // end loop
```

```
function v' •= new_squares (v);
{ let i•=0;        // initial value of local variable i.
  loop:            // this is the label
  if (i< length of v)
     {
       loop(i•=i+1);    // This activates or calls "loop" again.
       v'_i•=v_i×v_i;
     };
}; // new_squares
```

"Forall loop"

forall i•=m, n;{ statements }

Viewed equivalent to either of the following:

<table>
<tr>
<td>

```
{
 let i•=m;
 forall:
 if i ≤ n
    { forall(i•=i+1); statements };
}
```

// Can also count down
// from n to m.
</td>
<td>

```
{let i•=m; statements }
{let i•=m+1; statements }
  ...
  ...
{let i•=n-1; statements }
{let i•=n; statements }
```

// The let blocks may be
// written in any order.
</td>
</tr>
</table>

Adding n numbers - n a power of 2

For example, when n=8 we would like to calculate as follows.

$s' \bullet= (((a_0 + a_1) + (a_2 + a_3)) + ((a_4 + a_5) + (a_6 + a_7)))$

This enables parallel computation.

a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7

a_0 a_1 a_2 a_3

a_0 a_1

s'

First present specific solution for this case using three functions.

```
function s'•=addn(v,n)
{
if (n=1)
      {s' •= v0}
else addn ( n•=n/2;
            v0•=v0+v1;
            v1•=v2+v3
            .
            .
            vn/2-1 •= vn-2 + vn-1  );
}
```

May use a "let block" as follows:

```
function s'•=addn(v,n)
{
if (n=1)
     {s' •= v0}
else addn   (     n•=n/2;
                  { let i•=2; loop: v_{i/2-1} •= v_{i-2} + v_{i-1}; loop(i•=i+2); }
            );
}
```

May use a "forall loop" as follows.

```
function s’•=addn(v,n)
{
if (n=1)
     {s’ •= v0}
else addn   (    n•=n/2;
                 forall i•=1, n/2; { vi-1 •= v2i-2 + v2i-1; }
            );
}
```

$$s' \bullet= v_0 \qquad v_{i-1} \bullet= v_{2i-2} + v_{2i-1}$$

Sorting n numbers - n a power of 2

```
function v'•=m2 (v,size,place);
// SPECIFICATION: Merge two sorted runs of v of length "size"
// from position "place" onwards, giving one sorted run of length
// 2×size in v' from position "place" onwards.
```

{ASSUME THIS IS GIVEN.}

```
function v'•=mergesort(v)
{let size•=1;
 loop:
 if (size=length of v)
   {v'•=v}
  else
     loop ( size•=size*2;
              v•=m2(v,size,0);
              v•=m2(v,size,2*size);
              .
              .
              v•=m2(v,size,length of v – 2*size);
             );
}
```

function s’•=addn(v,n)

{

if (n=1)

  {s’ •= v_0}

else addn (n•=n/2;

  v_0•=v_0+v_1;

  v_1•=v_2+v_3

  .

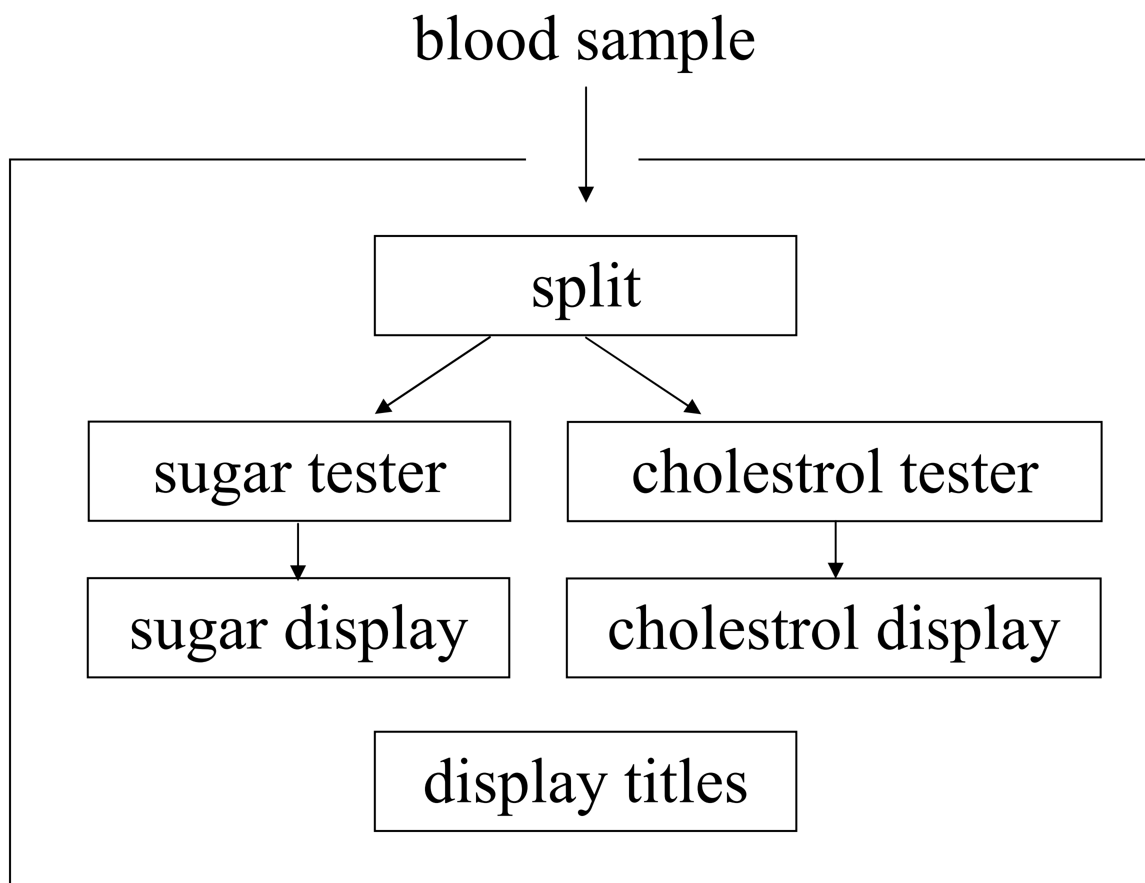
  .

  $v_{n/2-1}$ •= v_{n-2} + v_{n-1});

}

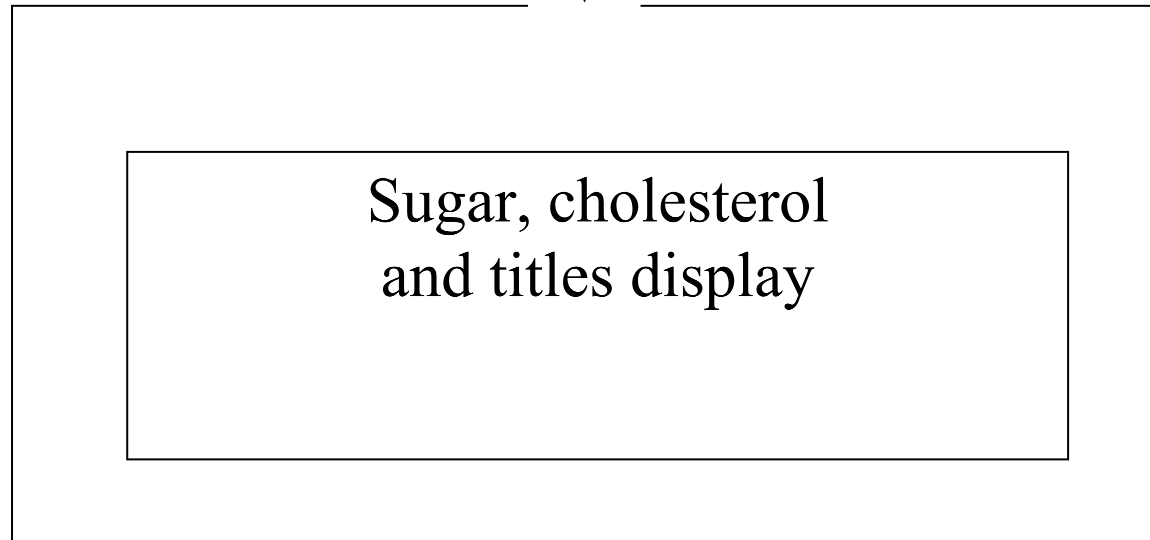
Overall System Description - Simplified Blood test machine

Machine can test Sugar, Cholesterol in a blood sample.



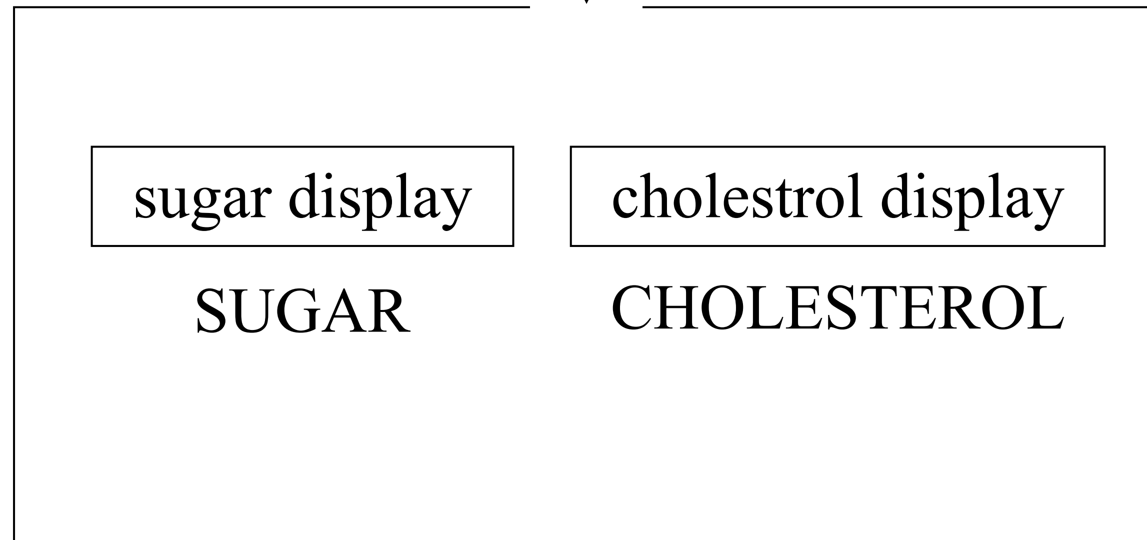
Design 1

blood sample



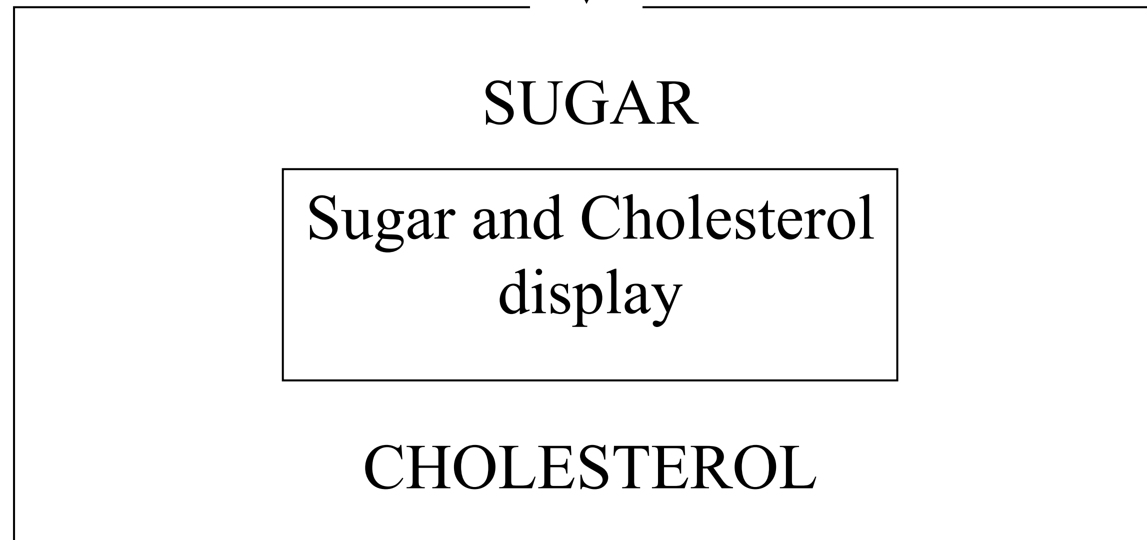
Design 2

blood sample



Design 3

blood sample



Description as a flexible algorithm

```
function    title1', sugar', title2', cholesterol' •=
bloodtester(blood_sample);
{ let sample1, sample2 •= split(blood_sample);
  title1’•="SUGAR";
  sugar' •= sugartester(sample1);
  title2’•="CHOLESTEROL";
  cholesterol' •= cholesteroltester(sample2); }
}; // end bloodtester
```

How to contact me

E-mail rafi@jct.ac.il

How to get the notes

Home page http://homedir.jct.ac.il/~rafi

Flexible Algorithms - An Introduction
(Latest English version)

מבוא לאלגוריתמים גמישים
(Older Hebrew version)