An Approach to the Optimization of High-speed Integer Algorithms

Nedeljko Ostojić
University Nikola Tesla, Nikodima Milaša 1, 59300 Knin,
Republic of Serbian Krayina, via Yugoslavia

Abstract

For a certain number of algorithms, efficiency is crucial. Optimization of the number of arithmetical operations is not enough when algorithm an is to be implemented in VLSI. An approach to the optimization of high speed sequential algorithms suitable for implementation in VLSI is presented. The approach is based on the weakest precondition calculus. In this way, the care of correctness of algorithm is taken through the optimization process. The method is illustrated by the optimization of average number of cycles needed per approximation point of integer case circle algorithm. As a result, an efficient incremental circle drawing algorithm suitable for implementation in VLSI is obtained. Although it asks for more arithmetical operations, this algorithm implemented in VLSI is faster than any other known circle generating algorithm. With the same approach other aspects and different types of algorithms could be optimized.

1. INTRODUCTION

The priority of algorithm efficiency is not as high as that of its correctness or maintainability. Furthermore, some techniques often used by programmers in order to make a program more efficient have adverse effect on program correctness (see [1], [3]). On the other hand, in certain applications efficiency is crucial: either to make a system economically useful or because of the particular nature of the application itself (e.g., real time systems). In some applications speed is so crucial that certain algorithms are implemented in VLSI to achieve as high speed as possible.

The task of speed optimization based on hardware characteristics is often left to the implementator. We expose an approach for such a case. The techniques presented belong to the class of "code improvement techniques", which are generally considered undesirable [1] because of danger that they could destroy program reliability. To avoid this, we suggest the techniques based on weakest precondition calculus, therefore the algorithm correctness could be easily kept through the optimization process.

We want to stress that the algorithm design could involve such constraints that an implementator could fail to obtain the best solution. Because of that, we propose that a designer and an implementator work together, at least in the optimization phase.

The fact that certain arithmetical operations, e.g. increment and decrement, are usually faster than the others, has an influence on the algorithm optimization. So, the approach with minimizing the number of arithmetical operations could fail to give the fastest solution even in a case when an algorithm is implemented in programming languages. It is particularly emphasized in cases when an algorithm is implemented in VLSI.

To illustrate our approach in this paper we consider the optimization of a circle generating algorithm.

Algorithms for drawing circles and circular arcs on integer grids belong to the class of fundamental algorithms of raster graphics. They are widely used not only for drawing circular arcs on raster screens, but also for incremental plotters, as well as for brushing different types of curves [8].

Although the problem has been discussed in the literature several times [2, 4, 6..8], the optimization of this class of algorithms seems not to be completed yet. Different environments ask for different types of algorithms, as well as for the optimization of different aspects: number of arithmetical operations, number of cycles, number and complexity of VLSI chips needed, etc.

During exercises with students we have studied possibilities of the correct usage of optimization techniques on algorithms for solving integer equalities. The approach is based on the weakest precondition calculus [3, 5]. Searching for examples

suitable for the illustration of these techniques, we have studied raster scan algorithms. (Students like examples on interactive computer graphics.) The formal approach is very interesting as an explanation of the development of Bresenham's line and circle algorithms, especially as a tool for their optimization. The main reason for this is in the fact that one could think about relations (predicates), and not about variables or values and combinatorial possibilities.

We agree with suggestions in literature [1, 3] that an optimization is to be done as a separate phase. We propose that a designer and an implementator together, as the first step, define the problem translating quality requirements and VLSI technology constraints into algorithm requirements. After that it could be useful that the designer explains the standard solution in terms of weakest precondition calculus. Then the implementator and the designer should try to optimize the algorithm using weakest precondition calculus and bearing in mind requirements from the problem definition step. If possible, different solutions of the problem are to be optimized, and their efficiencies analyzed and compared.

The paper is organized as follows. First, in the next section, we illustrate the problem definition step by translating quality requirements and VLSI technology constraints into algorithm requirements. Section 3 gives a development of common solution viewed from the aspect of the weakest precondition calculus. That makes a bases for section 4, where optimization techniques based on the weakest precondition calculus are discussed and implemented. Section 5 discusses optimized solution efficiency and compares it to the other solutions available to the author. Section 6 illustrates possibility to introduce such constraints in design phase which prevent implementator from achieving the best solution. The last section gives the conclusion and suggests the areas of possible implementation of techniques exposed in the letter.

2. PROBLEM DESCRIPTION

We propose that the designer and the implementator, as the first step, define the problem translating quality requirements and VLSI technology constraints into algorithm requirements.

In our illustration, the task is usually defined as the problem of activating the sequence of pixels whose centers are "the nearest" (according to certain metrics) to the circle defined by

$$f(x, y) = x^2 + y^2 - r^2 = 0$$

where r is integer (or sometimes half integer: 2*r is odd). If it goes for circular arc, then starting and ending conditions are defined.

There are other requirements concerning either the quality of the approximation or hardware constraints. Different metrics and kinds of approximations are discussed in [7]. An example of hardware constraints could be find in [8].

For the sake of presenting our approach, we have chosen a circle with the center in origin and integer radius. Pixels lie on grid lines. The algorithm has to select the same points as most widely used Bresenham circle algorithm [2, 4, 6].

The part of algorithm which has to determine the next approximation point may use only integer addition and/or subtraction in the arithmetic logic unit (ALU), as well as repetition and selection mechanisms with logical expressions in the guards. Magnitude comparisons are to be avoided.

Inside one cycle it is possible to execute the following set of actions:

- conditional incrementation or decrementation of some registers before and after an action in the ALU;
- one either addition or subtraction at most in the ALU;
- storage of the ALU output in some register (which could be incremented or decremented only before an action in the ALU);
- switching to the next state according to the values of most significant bits of the ALU and registers.

The most significant bits of all registers and the ALU are available for use in the guards of control statements.

3. A VIEW OF THE STANDARD SOLUTION

When the problem description is finished, it could be useful that the designer explains the standard solution in terms of weakest precondition calculus. Although this step is not necessary, it makes a good preparation for the subsequent optimization phase. Let us consider our illustration.

Exploiting the eight-way symmetry, authors usually consider an octant of circle [4, 6]. Starting from the beginning of the first octant (counting from 0° counterclockwise), the first point to be initialized is *(r, 0)*. After that, there are only two points available: in the first octant these are north *(x, y+1)* and northwest *(x-1, y+1)* points. It is not difficult to show that for integer case the choice could be made up according to the sign of the sum (see [4]):

$f(x-1, y+1) + f(x, y+1)$.

If the value is negative the north point is to be selected, otherwise the northwest point is to be selected.

The standard technique of taking invariant relation outside repetition mechanism [3, 5] is used to avoid the calculation of squares. To achieve that, the invariant relation

$$Pd: \; d = f(x-1, y+1) + f(x, y+1)$$
$$= (x-1)^2 + x^2 + 2*[(y+1)^2 - r^2)]$$

is initialized in the point $(r, 0)$ executing the simple statement "$d := 3-2*r$".

There are two progression statements, depending on whether the next point is to be activated by an axial or a diagonal move. So, the axial move is performed executing:

Sa: "$y := y+1$"

and the diagonal move is performed by concurrently changing both coordinates:

Sd: "$x, y := x-1, y+1$"

Using the weakest precondition calculus [3, 5] we could calculate how progression statements destroy the relation Pd. Consider the axial move:

$wp(Sa, Pd) = wp("y := y+1", Pd)$

$= wp("y := y+1",$
$d = (x-1)^2 + x^2 + 2*[(y+1)^2 - r^2)])$

which, after substituting all occurrences of y by $y+1$ yields

$= d = (x-1)^2 + x^2 + 2*[(y+1)^2 - r^2)]$
$+ 4*(y+1) + 2.$

Now it is easy to recognize the sequence:

$Sacd$: "$y := y+1; d := d+4*y+2$"

which keeps Pd invariant (proof that $Pd \Rightarrow wp(Sacd, Pd)$ is left as an exercise to the students), simultaneously making progress.

We can repeat the analysis for the diagonal move:

$wp(Sd, Pd) = wp("x, y := x-1, y+1", Pd)$

$= wp("x, y := x-1, y+1",$
$d = (x-1)^2 + x^2 + 2*[(y+1)^2 - r^2)])$

which, after substituting all occurrences of x by $x-1$ and all occurrences of y by $y+1$ yields

$= d = (x-1)^2 + x^2 + 2*[(y+1)^2 - r^2)]$
$- 4*[(x-1) - (y+1)] + 2.$

It could be proved that the sequence:

$Sdcd$: "$x, y := x-1, y+1; d := d-4*(x-y)+2$"

keeps Pd invariant, simultaneously making progress.

Therefore, the solution in a Pascal-like notation could take form:

```
procedure circle (r : integer);
  { assumes center in origin }
  var x, y, d : integer;
begin { circle }
  x := r;
  y := 0;
  d := 3 - 2*r;
  while x < y do begin
    activate_circle_points (x, y);
    y := y+1;
    if d < 0 then d := d + 4*y + 2
          else begin
            x := x-1;
            d := d - 4*(x-y) + 2
          end
  end { while };
  if x=y then activate_circle_points (x, y)
end { circle };
```

The procedure *activate_circle_points* (x, y) could make use of the eight-way symmetry to activate not only the first octant points but the full circle. There are also variations with the arbitrary center, but the program body is similar to the form given above.

At this point, the implementator and the designer together should consider drawbacks of solution from the implementator point of view. Such consideration for our illustration follows.

Although a simple one, the algorithm body asks for multiplication and four arithmetical operations per axial move or six arithmetical operations per diagonal move. Only one or two operations are increments/decrements. Besides, magnitude comparison is needed in the guard of the iteration mechanism.

After these considerations, the implementator and the designer are ready to start the optimization phase. The rest of this section is intended for the readers who are interested in the particular problem given in illustration.

Multiplication has been eliminated and the number of arithmetical operations has been reduced to five (after introducing new local variables for optimization) per diagonal move in [2]. It has been shown in [6] that, if magnitude comparison is allowed, the number of arithmetical operations could be reduced to two per axial and four per diagonal move without the use of local variables.

The common characteristic of the mentioned solutions is that they need at least two aditions/subtractions per diagonal step, without increments/decrements by one of some variables. Some of them also ask for magnitude comparison in the guard of the iteration mechanism.

4. AN APPROACH TO THE OPTIMIZATION

Bearing in mind the requirements given in the problem description, we shall try to optimize the algorithm in the sense that the optimized solution uses only allowed operations, and that the number of cycles per approximation point is as small as possible. In our example, we have to eliminate the multiplication and the magnitude comparison. Also, the number of aditions/subtractions is to be reduced.

Although there are nice demonstrations of using program transformation techniques in scan conversion algorithms, e.g. [9], they have not been implemented on circle-drawing algorithms yet. On the other hand, programers are discouraged to use so called "code improvement techniques", which are generally considered undesirable [1] because of danger that they could destroy program reliability. To avoid this, we suggest the approach based on weakest precondition calculus, therefore the algorithm correctness could be easily kept through the optimization process.

The common orderly technique is to recognize certain relations which stay invariant during the execution of the repetition mechanism, and to take them outside the iteration body. In this way, some operations are either eliminated or replaced by cheaper ones. We shall expose separately three variants of implementing this technique, because the each of them could be applied on a number of integer (including almost all variants of circle generating) algorithms.

4.1 Elimination of multiplication

Let us consider again the mechanisms *Sacd* and *Sdcd*. Although *4* is the constant which is the power of *2*, and multiplying by it is simple shifting, we have to do that shifting in each compensation statement. The idea is to introduce a new variable, say *dd*, and a new invariant relation, in our case $d = 4*dd$. Because the result of comparing the expression $4*dd$ to *0* is the same as that of comparing *dd* to *0*, it would be enough to use only one of the variables *d, dd*. In this way we could eliminate multiplying by *4*, if it is possible to initialize and keep the invariance of just introduced relation.

So, let us try to eliminate the multiplication by *4* introducing a new variable and a new invariant relation:

Pdd': $d = 4*dd$.

Now we have to answer the question: "how the progression mechanisms *Sa* and *Sd* followed by the compensation statements for restoring *Pd*, impact the relation *Pdd'*?" Using the weakest precondition calculus we obtain

$wp(Sacd, Pdd') =$

$= wp("y := y+1; d := d+4*y+2", Pdd')$

$= wp("y := y+1; d := d+4*y+2", d = 4*dd)$

which, after substituting *d* by $d+4*y+2$ yields

$= wp("y := y+1", d+4*y+2 = 4*dd)$

and, after substituting *y* by $y+1$:

$= d+4*(y+1)+2 = 4*dd.$

So, the result of executing the sequence: "$y := y+1; d := d+4*y+2$" is the enlargement of the left side in *Pdd'* by $4*(y+1)+2$. To restore the truth of *Pdd'* the right side has to be enlarged by the same amount. Obviously, the compensation statement for axial move should be:

"$dd := dd + (y+1) + 0.5$".

After the similar analysis of

$wp(Sdcd, Pdd') =$

$= wp("x, y := x-1, y+1; d := d-4*(x-y)+2", Pdd')$

$= d-4*((x-1)-(y+1)) + 2 = 4*dd,$

we recognize the compensation statement for the diagonal move:

"$dd := dd - (x-1) + (y+1) + 0.5$".

Although we have avoided multiplication, there is another problem: both compensation mechanisms ask for the addition of noninteger value *0.5*. Because of that, we should redefine the last invariant relation into:

Pdd: $(d = 4*dd + res)$ **and** $(0 \le res < 4)$.

It is important to note that the restriction of the domain of *res* in the second term of the conjunction in the definition of *Pdd* ensures

$(dd < 0 \Rightarrow d < 0)$ **and** $(dd \ge 0 \Rightarrow d \ge 0)$

i.e. that *d* always has the same sign as *dd*, if zero is associated with the positive sign.

After considering *wp(Sacd, Pdd)* and *wp(Sdcd, Pdd)*, it is not difficult to recognize appropriate compensation mechanisms:

Sac: **if** *res* ≥ *2* **then begin**
dd := (dd+1) + (y+1);
res := res - 2
end
else begin
dd := dd + (y+1);
res := res + 2
end

Sdc: **if** *res* ≥ *2* **then begin**
dd := (dd+1) - (x-1) + (y+1);
res := res - 2
end
else begin
dd := dd - (x-1) + (y+1);
res := res + 2
end

So, we know how to keep *Pdd* invariant when it has been initialized properly. How to initialize it?

To initialize *Pd* in state *(x=r, y=0)* we have to initialize *d* to *3 - 2*r*. It is not difficult to prove that the selection mechanism

if *odd (r)* **then begin**
dd := (1-r)/2;
res := 1
end
else begin
dd := -r/2;
res := 3
end

would initialize the relation *Pdd* in state *(x=r, y=0)*. (It is obvious that the execution of this mechanism establishes *4*dd + res = 3 - 2*r*. The weakest precondition calculus could be used to prove this formally.)

4.2 Reduction of aditions/subtractions introducing boolean variables

It could be noted that we need the variable *res* only to determine the value of the boolean expression *(res ≥ 2)* in the guards of *Sac* and *Sdc*. We could eliminate this variable and all arithmetical operations with it, if we introduce a new boolean variable and a new invariant relation:

Ppar: par ⇔ (res ≥ 2).

This relation is affected only by the changes of the variable *res*. There are two statements which destroy relation *Ppar*: if *par* is true this is "*res := res-2*", and if *par* is false this is "*res := res+2*". In both cases *Ppar* could be restored by the same mechanism: "*par := **not** par*".

The initialization of relation *Ppar* is quite simple: *par* is to be initialized to *true* if *r* is even, and to *false* if *r* is odd.

In this way we have eliminated multiplication. The price is paid through the introduction of the boolean variable *par*, the conditional incrementation of the variable *dd* and the negation of the variable *par* in each step. This is acceptable, because all these operations could be done in the same cycle with one addition/subtraction.

4.3 Reduction of aditions/subtractions introducing integer variables

There is still need to compare the magnitudes of variables *x, y*, as well as to make two aditions/subtractions in the case of the diagonal move (in the statement "*d := d-x+y*"). One could try to eliminate both of these imperfections by introducing a new integer variable defined by the following predicate:

Pdxy': dxy = x-y.

If *Sa* destroys *Pdxy'*, it could be restored by executing "*dxy := dxy-1*". If *Sd* destroys *Pdxy'*, it should be restored by executing "*dxy := dxy-2*". Again, it seems we could not afford only the increments/decrements of input variables for the ALU, because *dxy* is to be decremented either once by two, or twice by one.

The idea is to redefine *Pdxy'* in such a way that the value of *dxy* would be appropriate for the ALU input just after the first decrementation of *dxy*. Obviously, *Pdxy'* is to be redefined into

Pdxy: dxy = x-y-1.

In this case, *dxy* is prepared for the input in the ALU after a simple decrement by one (statement "*dxy := dxy-1*"). After the value of *dxy* is forwarded to the ALU, it could be decremented once again in order to keep the truth of *Pdxy*.

The initialization of *Pdxy* could be done by initializing *dxy* by *x-1* when *y* is *0*.

4.4 Optimized solution

Keeping *Pdd* invariant ensures that both variables *d, dd* have the same sign in all states ever encountered in the course of computation (according to the definition of *Pdd* and assuming that zero is associated with the positive sign). So, there is no need to use both of them. For the reason of simplicity, we shall borrow the name from the variable *d* and use it instead of *dd*. Having on mind that *(1-r)/2* in case when *r* is odd has the same result as *-r* **div** *2*, we could summarize our algorithm in the following form:

```
procedure circle (r : integer);
  { assumes center in origin }
  var x, y, d, dxy   : integer;
      par            : boolean;
begin { circle }
  x := r;
  y := 0;
  par := even(r);
  d := -r div 2; { Pd has been initialized }
  dxy := x-1; { Pdxy has been initialized }
  while dxy ≤ 0 do begin
    activate_circle_points (x, y);
    if d<0 then begin
        "increment conditionally d; increment y;
         decrement dxy";
        d := d + y;
        par := not par
      end
      else begin
        "increment conditionally d; increment y;
         decrement x, dxy";
        d := d - dxy;
        par := not par; "decrement dxy"
      end
  end { while };
  "increment dxy";
  if dxy = 0 then activate_circle_points (x, y)
end { circle };
```

We have reduced the set of operations in iteration step to one adition/subtraction of two integer variables, and several increments/decrements. All these could be done in the same cycle, according to the requirements given in section 2.

5. OPTIMIZED SOLUTION EFFICIENCY

If there are more than one algorithm solving the same problem, they are to be compared. If we have only one, it is good practice to try to make another one. See section 6 for illustration. Anyway, optimized solution efficiency is to be analyzed.

In our example, each iteration step of the optimized solution takes exactly one cycle and produces exactly one approximation point. Obviously, the average number of cycles per approximation point is one.

Other algorithms available to the author ask for more than that. A solution with considerably less arithmetical operations [6] in an implementation according with the requirements for VLSI given in section 2, would ask for one cycle per axial step and for two cycles per diagonal step, even if magnitude comparison was allowed. The average for that case could be estimated like this: the total number of approximation points in each octant is approximately $r \times \cos(\pi/4) = r/\sqrt{2}$. There are $r \times (\sqrt{2}-1)$ approximation points which are computed with axial moves, and $r \times (1 - 1/\sqrt{2})$ approximation points which are computed with diagonal moves. That gives the average of

$$\frac{r \times (\sqrt{2}-1) \; + \; 2 \times r \times (1 - 1/\sqrt{2})}{r/\sqrt{2}}$$

$$= \sqrt{2} \quad \text{cycles per point.}$$

The circle generating part from circle-brush algorithm given in [8] asks for even $3 \times \sqrt{2}$ cycles per point. Although it has stronger requirements on algorithm properties, it should be reconsidered in the light of results obtained here.

If it did not go for VLSI implementation, these results may happen to be far away from the best. In optimization techniques we introduce certain relations and keep them invariant. So, the price of optimization is paid through introducing new variables and establishing and keeping their definition relations invariant. Sometimes, this price is too high. A careful analysis is to be done concerning the number of machine cycles which are to be spent in different situations. The techniques exposed here could be used for such analysis.

Sometimes, the definite answer depends on the characteristics of available hardware. Certain solutions, theoretically faster, have proved to be slower on a specific hardware [8]. So, we have to understand the method and techniques of optimization, as well as techniques of estimating "what is better", to treat successfully a particular problem in a given context.

6. INFLUENCE OF THE DESIGN PHASE

In this section, we want to stress out the importance of cooperation between the designer and the implementator. There is a danger that the output from the design phase could involve such constraints that optimization could fail to achieve the best solution.

To illustrate this, suppose that the algorithm designer has done the job reasoning as follows (and not as it is described in section 3).

Starting from the beginning of the first octant (counting from 0° counterclockwise), the first point to be initialized is *(r, 0)*. After that, there are only two points available: in the first octant these are north *(x, y+1)* and northwest *(x-1, y+1)* points. In both cases we have to increase the value of the variable *y* by *1*. So, we could done this step unconditionally, and then

consider if there is need to decrease the value of variable x. It is not difficult to show that for integer case the choice could be made up according to the sign of the sum:

$f(x-1, y) + f(x, y)$.

If the value is negative current point is to be selected, otherwise the west point is to be selected.

Using the same optimization techniques as in section 4 we want to avoid the calculation of squares. To achieve that, the invariant relation

$Pd: d = f(x-1, y) + f(x, y)$
$= (x-1)^2 + x^2 + 2*[y^2 - r^2)]$

is initialized in the point $(r, 0)$ executing the simple statement "$d := 1-r$".

There are two progression statements, depending on whether we advance along y axes:

Sa: "$y := y+1$"

or x axes:

Sd: "$x := x-1$"

Using the weakest precondition calculus [3, 5] we could calculate how progression statements destroy the relation Pd. Consider the mechanism Sa:

$wp(Sa, Pd) = wp("y := y+1", Pd)$
$= wp("y := y+1", d = (x-1)^2 + x^2 + 2*[y^2 - r^2)])$
$= d = (x-1)^2 + x^2 + 2*[y^2 - r^2)] + 4*y + 2.$

Now it is easy to recognize the sequence:

$Sacd$: "$d := d+4*y+2; y := y+1$"

which keeps Pd invariant simultaneously making progress.

Consider the mechanism Sd:

$wp(Sd, Pd) = wp("x := x-1", Pd)$
$= wp("x := x-1", d = (x-1)^2 + x^2 + 2*[y^2 - r^2)])$
$= d = (x-1)^2 + x^2 + 2*[y^2 - r^2)] - 4*(x-1).$

It could be proved that sequence:

$Sdcd$: "$x := x-1; d := d-4*x$"

keeps Pd invariant simultaneously making progress.

Therefore, the solution in a Pascal-like notation could take form:

```
procedure circle (r : integer);
  { assumes center in origin }
  var x, y, d : integer;
begin { circle }
  x := r;
  y := 0;
  d := 1 - r;
  while x < y do begin
    activate_circle_points (x, y);
    d := d + 4*y + 2;
    y := y+1;
    if d ≥ 0 then begin
          x := x-1;
          d := d - 4*x
        end
  end { while };
  if x=y then activate_circle_points (x, y)
end { circle };
```

If the implementator had started from this solution, it is likely he would repeat the steps from section 4.

Trying to eliminate the multiplication by *4* the implementator would introduce a new variable and a new invariant relation:

Pdd': $d = 4*dd$.

After the same treatment as in section 4, using the same techniques, implementator should come to the following algorithm (a step before eliminating the magnitude comparison):

```
procedure circle (r : integer);
  { assumes center in origin }
  var x, y, d : integer;
begin { circle }
  x := r;
  y := 0;
  par := even(r);
  d := -r div 2; { Pd has been initialized }
  while x < y do begin
    activate_circle_points (x, y);
    if par then begin
          y := y + 1;
          d := d + y
        end
        else begin
          d := d + y;
          y := y + 1
        end;
    par := not par;
    if d ≥ 0 then begin
          x := x - 1;
          d := d - x
        end
  end { while };
  if x=y then activate_circle_points (x, y)
end { circle };
```

Although this solution has considerably less arithmetical operations (see [6]), in VLSI implementation it would be slower than the solution from section 4. That is so because it asks for two cycles when making the diagonal move. Elimination of the magnitude comparison make things more complicated. The transformation of this algorithm

into the form which asks for fewer cycles per approximation point is rather complicated task, even in this simple case.

Because of that, we propose that the designer and the implementator work together during the optimization phase.

7. CONCLUSION

An approach to the optimization of high speed integer algorithms suitable for VLSI implementation is considered. The techniques exposed belong to the class of "code improvement techniques", which are generally considered undesirable [1] because of the danger that they could destroy program reliability. To avoid this, we suggest techniques based on weakest precondition calculus, so that program correctness is not destroyed during the optimization process.

The approach is illustrated on the integer case circle generating algorithm. As a result, an efficient incremental circle drawing algorithm suitable for VLSI implementation is obtained. The output is the same as in the Bresenham's algorithm.

Each iteration step asks for exactly one either addition or subtraction in the ALU, and all other operations are only increments/decrements of four register variables (plus negation of one boolean variable). This enables the average of one cycle per approximation point. The algorithms from the literature available to the author ask for at least $\sqrt{2}$ cycles per approximation point in average.

This result is obtained using the optimization techniques which are variations of getting invariant relation outside repetitive construct. The approach is based on the weakest precondition calculus, which ensures an easy way to consider different possibilities taking into the consideration relations (predicates), and not variables or values. The author expects that the exposed method of optimization could be applied on the other contemporary integer algorithms as well.

The importance of cooperation between a designer and an implementator is emphasized. It has been illustrated how the algorithm design could involve such constraints which prevent the implementator from obtaining the best solution. Because of that, we propose that the designer and the implementator work together, at least during the optimization phase.

ACKNOWLEDGMENTS

I am indebted to Rade Tanjga, without whose support this work would never be carried out, to Predrag Rapajić for help about references, to Aleksa Zejak for his invaluable help and advices, and to my students for motivation.

REFERENCES

[1] Armenise P.: *A Structured Approach to Program Optimization,* IEEE Transactions on Software Engineering, Vol. 15, No. 2 (Feb. 1989), pp 101-108.

[2] Bresenham J. E.: *Algorithm for Circular Arc Generation,* Springer-Verlag, 1985., p 213. (Not available to the author, quoted according to [6].)

[3] Dijkstra E. W.: *A Discipline of Programming,* Prentice-Hall. Englewood Cliffs, N.J., 1976.

[4] Foley J. D., Van Dam, A.: *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, Reading, Mass., 1982, reprint with corrections, July 1984.

[5] Gries D.: *The Science of Programming,* Berlin-Heidelberg-New York, Springer 1981.

[6] Kuzmin Y. P.: *An Efficient Circle-Drawing Algorithm,* Computer Graphics Forum, Vol. 9, No. 4 (Dec. 1990), pp 333-336.

[7] McIlroy M. D.: *Best approximate Circles on Integer Grids,* ACM Transactions on Graphics, Vol. 2, No. 4 (Oct. 1983), pp 237-263.

[8] Posch K.C., Fellner W. D.: *The Circle-Brush Algorithm,* ACM Transactions on Graphics, Vol. 8, No. 1 (Jan 1989), pp 1-24.

[9] Sproull R. F..: *Using Program Transformations to Derive Line-Drawing Algorithms,* ACM Transactions on Graphics, Vol. 2, No. 4 (Oct. 1983), pp 259-273.