RC.89.117

ETH

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Niklaus Wirth

Modula-2 and Object-Oriented Programming

Drawing Lines, Circles, and Ellipses in a Raster

Flintstone

Eidg. Techn. Hochschule Zürich
Informatikbibliothek
ETH-Zentrum
CH-8092 Zürich

October 1989

117

30.01.12

Adresse des Autors:

Institut für Computersysteme
ETH-Zentrum
CH-8092 Zürich, Switzerland

© 1989 Departement Informatik, ETH Zürich

Modula–2 and Object–Oriented Programming

N. Wirth

1. Introduction

It is a sad fact that our field of Computer Science is overly dominated by fads. They typically appear in times of acute difficulties, are praised as powerful medicine against the major ills, and are carried by high hopes of all in despair. In the area of software and programming, the often cited software crisis, openly confessed in 1968, made *structured programming* popular. It was an expression of the recognition that complex software can only be understood, if it is orderly and structured. The development of huge systems, programmed by armies of "analysts", made it evident that coordination, documentation, and contracts between the participating workers in the form of interface specifications were mandatory. Management became a dominant topic, and all these aspects were somehow covered by the new wave called *software engineering*, implying the claim of a professional approach.

The most recent slogan is *object–oriented programming*. It expresses a different view of systems, focussed on decentralized control, and originated in the area of system programming. Every such trend has its legitimate reasons and goals, and it is appropriate to investigate its usefulness for one's particular objectives. Such an investigation is even necessary in order not to succumb to the negative aspects of a fad, namely to apply it where inappropriate, simply out of fear of being called old–fashioned. It is mandatory to understand the issues and foundations of a new discipline. Otherwise we will not master the discipline, but the discipline will master us.

2. What is "Object–Oriented" ?

The core of the concept is, I believe, decentralized control. The prime example, which is well–suited to explain the idea, is an operating system. A conventional system contains a central routine which accepts input from a keyboard and dispatches control to the routine specified for interpretation of the command. An even simpler example is the "operating system" of a desk calculator, which selects the routine according to the function key pressed. Modern workstations with their multiple window (viewer) capability require a more sophisticated approach. Typically, operations are demanded through a mouse click. The action to be taken depends on the position of the displayed cursor. It is unknown to the system a priori, and depends on the type of viewer in which the cursor happens to be located. Each viewer is considered to carry its own mode of command interpretation, in short, is regarded as an *object with its own behaviour.* This scheme is implemented by executing a search for the descriptor representing the viewer designated by the current cursor position, and then by dispatching control to a routine assigned to that descriptor, a so–called *handler.* Naturally, different (types of) viewers must be able to contain different handlers. Instead of control being centralized in a single dispatcher (in which the identities of the destinations are explicitly specified), control is decentralized in handlers, whose identity and number is not specified in the dispatcher's programtext.

Incidentally, this view of a structure carrying its own routines for interpreting its data coincides with the notion of the abstract data type. The type declaration does not only specify the type and structure of data, but also the applicable operators and function. Variables are said to be instances of the type. In the community of object–oriented programmers, the set of objects with identical data structure and handler is called a *class*, and an object is an instance of a class, just like a variable is an instance of a type.

It is often desirable to be able to derive a new class from a given class in the sense that instances of the new class share the properties, i.e. attributes and operations, with objects of the given class, but feature certain added properties. Thereby they become special members of the original class and form a *subclass.* A typical example is given by subclasses of viewers: text viewers, graphic viewers, picture viewers will share all properties of viewers, and feature additional operators suitable for handling texts, graphics, or pictures.

Some programmers find it attractive to view computer systems like humans. An object-oriented system is then compared with a human society. A symptom of this anthropomorphic view – which I find misleading rather than useful – is the notion that a subclass *inherits* the properties of its superclass. Thus the subject of inheritance has found its entry into the programmers technical jargon. It might be added in passing, that the term *subject-oriented* would have been more consistent with the popular anthropomorphic view than object-oriented. After all, in the conventional sense, it is the subject that displays a behaviour, that receives messages, whereas the object plays the passive role.

It is by no means accidental that the paradigm of object-oriented programming – we bow to convention and adopt the misnomer – originated in the application area of simulation of systems with discrete events. There emerged the need to represent abstractions of agents with properties *and* behaviour. Such abstractions were first expressed in the languages Simula-1 [1] and Simula-67 [2]. The main focus still lay on simulating the collective and concurrent actions of classes of agents using an interpreter with a single processor. Hence, the notion of processes or, more precisely, coroutines remained intimately coupled with Simula. The notion of an object, however, was adopted and plays the central theme in the language Smalltalk [4]; at the same time, the paradigm of simulation and quasi-concurrency was dropped – or at least fell into the background.

3. Which Features make a Language "Object-Oriented" ?

It is worth noting that applications where the object-oriented view makes sense typically involve a large number of data elements, most of them with a transient existence. Therefore, the primary requirement to formulate such applications is the availability of *dynamic data structures*, typically expressed through records related by pointers. The necessary mechanisms are dynamic data allocation and (preferably automatic) retrieval.

Apart from this prerequisite, we can identify two essential requisites:

1. It must be possible to define templates (of objects) consisting of variables *and* procedures. Templates were given the name *class*, and instances of class are said to be *objects*. Procedures defined for a class are called *methods*, and invoking a method is called *sending a message.*

2. It must be possible to derive new classes from existing classes. A derived class *is related* to the class from which it is derived by the fact that it adopts the latter's variables, adopts or replaces its procedures, and possibly adds new variables and procedures. A derived class is compatible with the deriving class in the sense that an instance of the derived class can be substituted for any object of the deriving class.

These compatibility rules imlpy that a procedure of an object may be invoked without reference to its exact identity, because the object may be an instance of many of the derived classes. This explains the term *sending a message* instead of *calling a procedure:* the meaning of the message is known, the interpreting procedure is not. The dispatch of a message can now occur in a place where the actual procedure is unknown, in particular in a module that lies *below* the module in which the procedure is defined. Hence, such calls also are known under the term *upcall.*

type	class
variable	object, instance
procedure	method
call	message
extension	inheritance

Conventional vs. Object-Oriented Terminology

4. Modula-2 for Object-Oriented Programming?

We are now in a position to investigate the suitability of Modula [3] for object-oriented programming. The prerequisite of dynamic data structures is satisfied (although most implementations do not contain automatic storage retrieval).

The first requirement is also satisfied through the existence of procedure types. Objects are representable as records, their methods as procedure-typed fields. Sending a message turns out to be calling a procedure indirectly via such a procedure-typed variable.

The second requirement, however, is not met. It is impossible to derive a type T1 from a type T0 such that T0 remains compatible with T1 (apart from the trivial cases of identity and of subranges).

We could leave the topic at this point. However, it is worth investigating whether an object-oriented style might be used with Modula when, perhaps, certain safety properties of Modula are sacrificed. As was pointed out in [9], a possible approach lies in using Modula's loophole, features from the module SYSTEM. In particular, the type ADDRESS offers a solution: In anticipation of the need for derived types, the declaration of the basic type is provided with an additional field, say *ext*, of type ADDRESS. A derivation is then possible through any record type containing the additional variables and procedures, and by assigning its pointer to the field *ext*.

Apart from the need for an additional indirection in accessing the additional fields, the crucial drawback is that the compiler's type checking capability has been crippled. A program using this recipe is potentially as unsafe as assembler code. Type safety should be the last property of a high-level language that we are willing to sacrifice. The low-level facilities in Modula-2 were provided with the intent that they be used sparingly in cases where access to special machine resources are needed, and that they be isolated in small driver-modules. But I caution against their installment as central instruments to be used throughout entire programs.

What we must find from the point of view of language design is neither a fix nor a trick, but a solution which properly integrates the new requirement with the existing properties and fully complies with the concept of type validation through textual inspection (viz. compile-time checking).

5. Extending Modula-2 for Object-Oriented Programming

Our approach to rendering Modula-2 suitable for object-oriented programming lies in extending the language with the proper features and making sure that they fit into the existing framework, allowing a precise and concise definition based on well-understood mathematical concepts. If we concentrate on the fundamental requirements, only a single new facility is actually called for, namely one for introducing derived classes of objects.

In accordance with the basic principle of language design, namely to introduce as few concepts as possible, and recognizing the strong similarity, if not identity, of types and classes, we equate the terms object with instance (of a record type) and class with type. A derived class (subclass) then corresponds to an *extension* of a record type [5]. If a record type is regarded as spanning a coordinate space being the Cartesian product of the subspaces spanned by the record field types, the extension increases the dimensionality of the space. Accordingly, an assignment of an instance of a derived class, i.e. of a variable of an extended type, to an instance of its superclass, i.e. to a variable of the base type, then simply corresponds to a *projection* of the variable's value onto to subspace spanned by the base type.

Example:

```
TYPE  Point2 = RECORD x, y: INTEGER END
      Point3 = RECORD (Point 2) z: INTEGER END
VAR   P2: Point2; p3: Point3;
```

p2 := p3   corresponds to   p2.x := p3.x; p2.y := p3.y

Naturally, the concept of extension is also applied to pointer types. Thereby it becomes possible to construct heterogeneous structures whose relating pointer type is bound to a node type R. The nodes of the structure may then be of different extensions of R, say R1, R2, R3. Clearly, a need for determining the actual (extended) type of a node referenced by a pointer (bound to R) arises. Oberon provides it in the form of a *type test* classified as a Boolean factor with the operator IS.

We emphasize that the principal achievement of this solution in Oberon [6, 7] is that the concepts of types and classes were *united*, and that the coexistence of two distinct notions representing virtually the same concept was avoided.

6. "Syntactic" Issues

With respect to representing methods through procedure–typed record fields, some additional remarks appear appropriate. In object-oriented languages, a class declaration appears like a record declaration with additional procedure declarations (or of headings thereof). This has some consequences and some advantages. In Modula and Oberon, the corresponding procedure–typed field assumes the role of a variable, and hence the actual procedure must be assigned to it explicitly each time an instance of the record is generated. This can be regarded either as a burden (and a source of mistakes) or as an additional degree of freedom (and power). Yet, most typical applications bind the *same* procedure (handler) to *all* instances of a class: the view of methods is *class–centered*, Oberon's view is *instance–centered.*

Example:

```
CLASS Viewers =                          TYPE Viewers =
    BEGIN x, y, w, h: INTEGER;               RECORD x, y, w, h: INTEGER;
     METHOD restore (T: Text)                    restore: PROCEDURE (T: Text)
        BEGIN ...                            END
     END  restore
    END

v := Viewers.New(X, Y, W, H)             NEW(v);
                                         v.x := X; v.y := Y; v.w := W; v.h := H;
                                         v.restore := Restore
```

In an implementation of the class–centered view, each instance will contain a hidden pointer to the same table of procedure references. In an Oberon implementation, each instance contains direct (and duplicated) references to the installed procedures. This is clearly undesirable, if there are many of them.

Another advantage of the class–centered view and of declaring procedure bodies within the class declaration is the possibility to directly refer to the object fields (x, y, w, h) from within the procedure. This led to the following, convenient formulation of sending a message (restore) to an object (v):

```
v.restore(T)
```

This form is entirely consistent with the notation for field designators, viz. v.x. In the above call, v plays a double role as *distinguished parameter.* It both acts as a qualification for the method name (via the class of v), and it represents a parameter of the call, namely the variable v. In Oberon, this abbreviating form is not possible, and the two roles are clearly disjoint:

```
v.restore(v, T)
```

The distinction between class– and instance–centered view with regard to method definitions can be regarded in another way: In the former case, methods are declared as (procedure valued) *constants,* in the latter as (procedure valued) variables. The restrictiveness of the former appears already in conjunction

with subclasses, i.e. type extensions. Typically, a subclass features methods different from those of its superclass. As they are delcared as constants, a new "feature" is called for: that of *overriding* the definition of the superclass. In respective implementations overriding is manifested by the provision of a distinct method table for each subclass. In the instance-centered view of Oberon, no such additional mechanism is necessary, nor has the additional notion of overriding any place, as it occurs through a regular, explicit assignment.

A class-centered derivative of Oberon has recently been devised and implemented [10]. It showed that the additional complexity of the compiler remains within tolerable bounds. The question is rather whether the notational conveniences justify the *conceptual* complications, and it remains open.

Object-oriented languages typically confine objects to be dynamically allocated records referenced via pointers. In Oberon, such a restriction would have to be defined through an explicit, exceptional rule, because static as well as dynamic variables can be of a record type and are therefore extensible. In fact, the availability of the type extension concept turned out to be extremely useful also in the case of static variables passed as (reference) parameters to procedures. The above cited restriction would lead to the use of dynamic allocation when conceptually a variable should be declared as static and local because of its transitory nature. This in turn may have grave consequences on the efficiency of an implementation. Oberon's generality turns out to be a significant benefit, and regarding every variable as an object a mistake.

Conclusions

Apart from convenient syntactic constructs, an object-oriented language features declarations of procedures bound to data structures (records), and offers the possibility to declare structures (extensions) that are derived from other structures and are type-compatible with them.

In Modula-2, it is possible to adopt the object-oriented paradigm by resorting to low-level facilities and sacrificing the most important asset of a high-level language, the guarantee of type consistency. Although Modula allows the use of the paradigm, it does not support it.

The language Oberon extends Modula-2 with the necessary facility: *type extensions*. There exist, however, some differences between Oberon's object-oriented facilities and those of typically object-oriented languages. The principal difference is that in the latter procedures (called methods) appear as constants in the declaration of a record type (called class), whereas in Oberon they appear as variables (record fields). Hence, in the former case methods are guaranteed to be the same for all instances of a class, whereas in Oberon they may differ from instance to instance (and need to be explicitly installed whenever an instance is generated). As a result, no additional language facility is needed in Oberon for providing different procedures for extended types (subclasses), whereas in the typical object-oriented languages the redefinition of methods (installed as constants) requires the additional concept of overriding.

As a result, Oberon is conceptually simpler, and Oberon implementations are not burdened with additional class mechanisms. On the other hand, object-oriented languages may offer somewhat more convenient notational facilities and provide security by guaranteeing the constancy of declared methods for all instances of a class, resulting in improved efficiency of upcalls. We consider this as a negligible advantage, since we believe that the use of the object-oriented paradigm should be employed selectively only where appropriate. In the design of an entire operating system [8], we found that almost the whole system was advantageously programmed in the conventional style, the object-oriented style being restricted to the viewer system which provides distributed control. It is wise to use upcalls sparingly.

A much more significant contribution to efficiency is the generalization of the type extension (subclass) concept to static variables, in particular in their use as procedure parameters.

The most significant aspect of Oberon is that it supports object-oriented programming in the framework of a language fully supporting also the conventional style, and that it guarantees full type-consistency

checking. Oberon thus differs from other languages, because it grew from the conviction that language design should strive for simplification through integration of highly similar concepts, rather than for complication through the addition of new facilities highly similar to already existing ones.

References

1. O.-J. Dahl, K. Nygaard. Simula: An Algol-Based Simulation Language. *Comm. ACM 9*, 9, (Sept. 1966), 671-678.

2. G. Birtwistle, et. al. *Simula Begin.* Auerbach 1973.

3. N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1982.

4. A. Goldberg, D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley 1983.

5. N. Wirth. Type Extensions. *ACM Trans. Prog. Lang. and Systems, 10*, 2 (April 1988), 204-214.

6. --. From Modula-2 to Oberon. *Software, Practice and Experience 18*, 7, (July 1988), 661-670.

7. --. The Programming Language Oberon. *Software, Practice and Experience 18*, 7, (July 1988), 671-690.

8. J. Gutknecht, N. Wirth. The Oberon System. *Software, Practice and Experience, 19*, 9, (Sept. 1989), 857-893.

9. G. Blaschek. Implementation of Objects in Modula-2. *Structured Programming, 1*, 3, (June 1989), 147-156.

10. H. Mössenböck, J. Templ, R. Griesemer. Object-Oberon, An Object-Oriented Extension of Oberon. Report 109, Dept. Informatik, ETH Zürich (June 1989).

Paper presented at the First International Conference on Modula-2, Bled, Yugoslavia, Oct. 11-13, 1989.

Drawing Lines, Circles, and Ellipses in a Raster

N. Wirth

Abstract

In a tutorial style, Bresenham's algorithms for drawing straight lines and circles are developed using Dijkstra's notation and discipline. The circle algorithm is then generalized for drawing ellipses.

1. Introduction

Recently, I needed to incorporate a raster drawing algorithm into one of my programs. The Bresenham algorithm is known to be efficient and therefore was the target of my search. Literature quickly revealed descriptions in several sources [1, 2]; all I needed to do was to translate them into my favourite notation. However, I wished – in contrast to the computer – not to interpret the algorithms but to *understand* them. I had to discover that the sources picked were, albeit typical, quite inadequate for this purpose. They reflected the widespread view that programming courses are to teach the use of a (specific) programming language, whereas the algorithms are simply given.

Dijkstra was an early and outspoken critic of this view, and he correctly pointed out that the difficulties of programming are primarily inherent in the subject, namely in constructive reasoning. In order to emphasize this central theme, he compressed the notational issue to a bare minimum by postulating his own notation that is concisely defined within a few formulas [3].

In the following examples we adopt his notation but deviate from his discipline by specifying the task of drawing algorithmically rather than by a result predicate over the drawing plane. In each case of the three curves, the algorithm's principal structure is that of a repetition. In each step a next raster element (pixel) is marked. In one dimension of the drawing plane, the next coordinate value is given by adding 1. The considerations concerning termination are therefore trivial: repetition terminates when a limit value has been reached. The problem is reduced to computing the other coordinate of the next raster point to be marked. Bresenham's central idea is, instead of evaluating the function defining the curve, to compute the coordinate incrementally from the coordinate of the last pixel, using integer arithmetic only. The problem is now reduced to find an auxiliary function h which determines whether the coordinate must be incremented or not. If the slope of the curve is guaranteed to be at most 45°, the increment is either 0 or 1. The statements for computing the auxiliary function in each step are derived by simple application of the axiom of assignment. The function's definition appears as the loop invariant.

2. Lines

Let the straight line L be defined by the equation

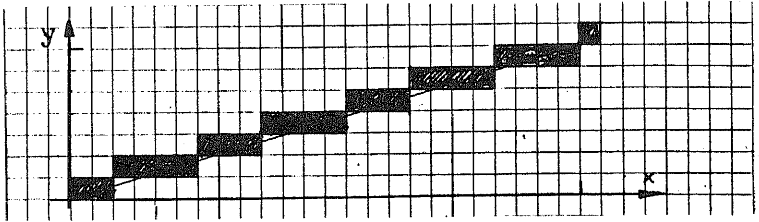
$$L: b \cdot x - ay = 0$$

We now wish to plot the section from the origin to the point P(a, b) and accept, without loss of generality, the condition $0 \leq b \leq a$. Evidently, this can be done by stepwise incrementing x, each time computing and rounding y, and marking the square with the resulting coordinates.

```
x := 0;
DO x < a →
  y := b•x DIV a; Mark(x, y); x := x+1
OD
```

The occurrence of a multiplication and a division in each step calls for a more efficient solution, particularly one that avoids the use of fractions (floating-point numbers). It rests upon the idea to determine, through evaluation of a simple integer valued function, whether or not the ordinate y has to be incremented or not. Given a point x, y, the exact value Y of the ordinate of the next point follows from the equation for L:

$$Y = (x+1) * b/a$$



The integer value y nearest to Y must satisfy the condition

$$Y - 1/2 \leq y < Y + 1/2$$

If the first part of the inequality is not satisfied, an increase of y is necessary. An increment by 1 then implies that both parts become satisfied:

$$(x+1) * b/a - 1/2 \leq y$$
$$bx + b - a/2 \leq ay$$
$$bx - ay + b - a/2 \leq 0$$

We introduce an auxiliary variable $h = bx - ay + b - a/2$; this equality is a loop invariant. Each time x or y are incremented, h has to be adjusted according to its definition. By direct application of the axiom of assignment, we determine the precondition of x := x+1:

{h = b(x+1) + ay + b – a/2} x := x+1 {h = bx – ay + b – a/2}

and now wish to find an expression u such that

{h = bx – ay + b – a/2} h := u {h = b(x+1) + ay + b – a/2}

By applying the axiom of assignment once again, we find u = h+b. A similar application yields the necessary adjustment of h in the case of incrementing both x and y, the term k being constant.

h := h+b; {h = b(x+1) + k} x := x+1 {h = b+k}
h := h+b–a; {h = b(x+1)–a(y+1)+k} y := y+1; x := x+1 {h = bx–ay+k}

The resulting program is known as Bresenham's algorithm. In the expression defining the initial value of h, a/2 has been replaced by a DIV 2. Hence, h may be (at most) 1/2 too small. This is acceptable, because for all integers h, and for all c such that $0 \leq c < 1$, $h > 0 \equiv h > c$.

```
x := 0; y := 0; h := b – a DIV 2;
DO {P}
 x < a → Mark(x, y);
 IF h ≤ 0 → h := h + b
  | h > 0 → h := h + (b–a); y := y+1
 FI ;
 x := x+1
OD
```

3. Circles

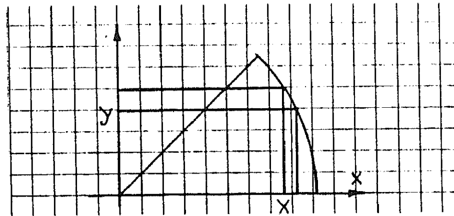
Our task is to plot a circle C with its center at the origin and with radius r. It is defined by the equation

$$C: x^2 + y^2 = r^2$$

The obvious method of computing x and y values for various angles by using trigonometric functions must be rejected as too inefficient. Like in the case of a straight line, we wish to operate on integers only, and to find a simple function which determines whether or not the ordinate has to be incremented. For this purpose, we concentrate on plotting the circle in the first octant of the plane only (0 – 45°). The remaining 7 octants can easily be covered by symmetry arguments and need no further computation.

Starting from the point P(r, 0), we increment y in unit steps. In each step the exact abscissa X for the next point is derived from the circle equation as

$$X = \mathrm{sqrt}(r^2 - (y+1)^2)$$



The abszissa of the next raster point to be marked must satisfy the inequalities

$$X - 1/2 \le x < X + 1/2$$

Replacing X in the second part yields

$$x - 1/2 < \mathrm{sqrt}(r^2 - (y+1)^2)$$
$$x^2 - x + 1/4 < r^2 - y^2 - 2y - 1$$
$$x^2 + y^2 + 2y - x - r^2 + 5/4 < 0$$

In each successive step, y is incremented and x is, if necessary, decremented such that the condition $h < 0$ is reestablished, where

$$h = x^2 + y^2 + 2y - x - r^2 + 1$$

The necessary adjustments of h upon changing x or y directly follow from application of the axiom of assignment:

$$\{h = y^2 + 2y + k\}\ h := h + 2y + 3\ \{h = y^2+4y+3+k\}\ \ y := y+1\ \ \{h = y^2 + 2y + k\}$$
$$\{h = x^2 - x + k\}\ h := h - 2x + 2\ \{h = x^2-3x+2+k\}\ x := x-1\ \ \{h = x^2 - x + k\}$$

From this follows the very efficient algorithm due to Bresenham.

```
x := r; y := 0; h := 1–r;
DO y < x → Mark(x, y);
  IF h < 0 → h := h + 2y + 3
  | h ≥ 0 → h := h + 2(y–x) + 5; x := x–1
  FI ;
  y := y+1
OD
```

4. Ellipses

Similarly to the circle algorithm, we wish to design an algorithm for plotting ellipses by proceeding in steps to find raster points to be marked. We concentrate on the first quadrant; the other three quadrants can be covered by symmetry arguments and require no additional computation.

Let the ellipse be defined by the following equation. Again without loss of generality, we assume $0 < a \leq b$.

$$E: (x/a)^2 + (y/b)^2 = 1$$

We start with the point P(0, b) and proceed by incrementing x in each step, and decrementing y if necessary. The exact ordinate of the next point follows from the defining equation:

$$Y = b * \text{sqrt}(1 - ((x+1)/a)^2)$$

The raster point coordinate must satisfy

$$y - 1/2 < b * \text{sqrt}(1 - ((x+1)/a)^2)$$
$$y^2 - y + 1/4 < b^2 - b^2*(x+1)^2/a^2$$
$$a^2y^2 - a^2y + a^2/4 < a^2b^2 - b^2x^2 - 2b^2x - b^2$$
$$b^2x^2 + 2b^2x + a^2y^2 - a^2y + a^2/4 - a^2b^2 + b^2 < 0$$

The necessary and sufficient condition for decrementing y is therefore $h \geq 0$ with the auxiliary variable h being defined as

$$h = b^2x^2 + 2b^2x + a^2y^2 - a^2y + a^2/4 - a^2b^2 + b^2$$

As in the case of the circle, the termination condition is met as soon as y might have to be decreased by more than 1 after an increase of x by 1, i.e. when the tangent to the curve is greater than 45°. Unlike in the case of the circle, however, this condition is not obviously given by x = y. We reject the obvious solution of computing the ordinate for which the curve's derivative is –1, because this computation alone would involve at least the square root function. Instead we compute a function g, similar to h, incrementally. Its origin stems from the inequality

$$y - 3/2 < b * \text{sqrt}(1 - ((x+1)/a)^2)$$

implying that the ordinate of the next point be at least 3/2 units below the current raster point. Therefore, a decrease of y by 2 would be necessary for an increase of x by 1 only. A similar development as for h yields the function g as

$$g = b^2x^2 + 2b^2x + a^2y^2 - 3a^2y + 9/4*a^2 - a^2b^2 + b^2$$

and x can be incremented as long as $g < 0$. The first quadrant of the ellipse is then completed by the same process, starting at the point P(a, 0), of incrementing y and conditionally decrementing x. The auxiliary function h is here obtained from the previous case of h by systematically substituting x, y, a, b for y, x, b, a. The derivation of the incrementing values for h and g follow from application of the axiom of assignment and completes the design considerations for the following algorithm:

$$h := h + b^2(2x + 3) \;\; \{h = b^2x^2+2b^2x+b^2 + 2b^2x+2b^2 + k\} \;\; x := x+1 \;\; \{h = b^2x^2 + 2b^2x + k\}$$
$$h := h - 2a^2(y-1) \;\; \{h = a^2y^2-2a^2y+a^2 - (a^2y-a^2) + k\} \;\; y := y-1 \;\; \{h = a^2y^2 - a^2y + k\}$$
$$g := g - 2a^2(y-2) \;\; \{g = a^2y^2-2a^2y+a^2 - 3(a^2y-a^2) + k\} \;\; y := y-1 \;\; \{g = a^2y^2 - 3a^2y + k\}$$

```
x := 0; y := b;
h := (a² DIV 4) – ba² + b²; g := (9/4)a² – 3ba² + b²;
DO g < 0 → Mark(x, y);
```

```
        IF h < 0 → d := (2x + 3)b²; g := g+d
         | h ≥ 0 → d := (2x + 3)b² – 2(y–1)a²; g := g + d + 2a²; y := y–1
        FI ;
        h := h+d; x := x+1
    OD ;
    x := a; y1 := y; y := 0;
    h := (b² DIV 4) – ab² +2a²;
    DO y ≤  y1 → Mark(x, y);
        IF h < 0 → h := h + (2y + 3)a²
         | h ≥ 0 → h := h + (2y + 3)a² – 2(x–1)b²; x := x–1
        FI ;
        y := y+1
    OD
```

We close this essay with the remark that values of h may become quite large and that therefore overflow may occur when the algorithm is interpreted by computers with insufficient word size. Unfortunately, most computer systems do not indicate integer overflow! Using 32-bit arithmetic, ellipses with values of a and b up to 1000 can be drawn without failure.

References

1. N. Cossitt. Line Drawing with the NS32CG16 and Drawing Circles with the NS32CG16. AN–522 and AN–523. National Semiconductor Corp. (1988)

2. J. D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics.* Addison–Wesley, 1982.

3. E. W. Dijkstra. Guarded Commands, Nondeterminacy, and Formal Derivation of Programs. *Comm. ACM 18*, 8, (Aug. 1975), 453–457.

Flintstone

N. Wirth

1. Historical Background

Flintstone is – in this context – the name of a microprogrammable microprocessor. It consists, like the common flintstone, of silicon or – to be more precise – of silicon oxyde. This at least since July 1989. Before that date, it existed on paper only, its architecture having been defined by the author in the summer of 1983. At that time, or rather even earlier, the idea existed to design and build a more powerful, more highly integrated version of the Lilith computer [1]. The idea was soon dropped for economic reasons, and the purpose of Flintstone was rather seen as a fast and flexible coprocessor for raster operations on the one hand, and as a processor optimally suited to display the essentials of a processor architecture and to exercise the craft of microprogramming.

Plans to implement Flintstone were picked up by M. Morf, but they didn't come to fruition and the project seemed buried. In 1985, W. Fichtner, head of the Institute for Integrated Systems at ETH, rediscovered Flintstone in a search for projects suitable for gaining design experience for integrated circuits, the design appearing to be neither too simple, nor too complicated for the available tools and technology. Flintstone's layout was designed by Th. von Eicken as a student project, in fact twice because during the design phase the fabrication process and its parameters had been altered. The layout had been completed in the fall of 1987 for Faselec's Sacmos fabrication process with a 2u gate width.

In the summer of 1989, a Flintstone wafer became available and samples were tested by H. Bonnenberg and N. Felber [2]. They reach a clock rate of about 8 MHz. The interest in Flintstone primarily focusses on its use as a vehicle for microprogramming a RISC–type architecture in a laboratory environment. To make available a suitable description of the processor is the primary reason for writing this report. Another is the opportunity to acknowledge the many contributions that have moved Flintstone from paper to silicon, and to thank all the participants for their expertise and endurance.

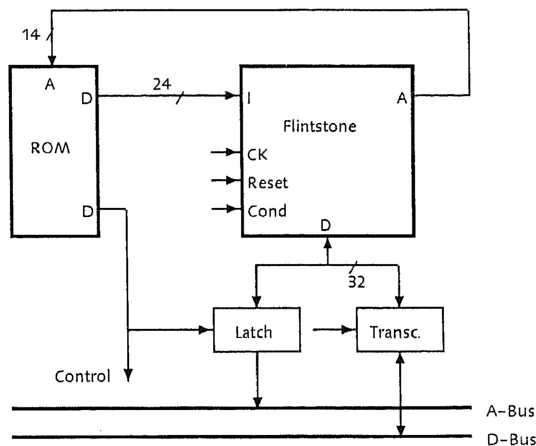


Fig. 1. Typical Use of Flintstone

2. Flintstone's Architecture

Because of the planned goal of using Flintstone as processor of a second version of the Lilith computer, and because Lilith's central processing unit was built around AMD's eminently successful bit-slice chip Am2901, Flintstone's architecture is modelled after the 2901 with a few modifications. The chip combines eight 2901's, yielding a data path width of 32 bits, and a sequence controller, modelled after the Am2910 with its address path widened to 14 bits [3]. In order to obtain optimal flexibility, the program code was to be stored in an off-chip memory, presumably ROM. The micro-instruction width is 24 bits. A typical application is sketched in Fig. 1.

In the subsequent description of the internal architecture, we first concentrate on the arithmetic-logical unit (ALU) and its data paths. This is followed by an explanation of the support of multiplication and division, which differs from that of the Am2901. The last part deals with Flintstone's control unit and its instruction address paths. The control signals for ALU and CU together determine the micro-instruction format.

3. The Arithmetic-Logic Unit

The data processing unit consists of 16 registers, the arithmetic-logic unit (ALU) and a funnel shifter. The data path is shown in Fig. 2. The functions available on the ALU are addition and logical AND, OR, and XOR. The funnel allows selecting any 32-bit section from a 64-bit double word. In each clock cycle, data flow from the two input sources either through the ALU or the funnel to their destination, either to a register or via the data port to the external bus. The input sources are selected by the R- and S-multiplexors. The T-multiplexor selects the output of either the ALU or the funnel. The registers form a two-port RAM; two registers can be selected concurrently and are specified by the A- and B-addresses; the latter also specifies the destination register which is to receive the RAM-input.

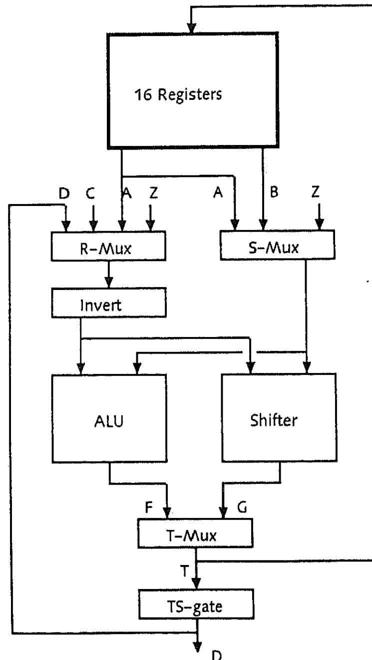


Fig. 2. Basic Data Paths

The selected data path and the ALU function are determined by the current instruction. During each clock cycle, exactly one instruction is interpreted. The following instruction fields are provided for controlling the data flow:

a	4	A–register address
b	4	B–register address
r		R–mux control
s		S–mux control
t		T–mux control
w	1	Register input (write) enable
f		ALU function
m		Shift count
c		ALU carry input
i		inverter control

The shift amount of the funnel can be selected from either the corresponding instruction field or – if it depends on the result of previous operations – from the 5 low–order bits of the Q register.

Since either the output of the ALU or the output of the funnel is discarded, the instruction fields for the ALU–function and for the shift amount overlap (are the same).

4. Multiplication

The Flintstone architecture contains a few additional facilities to make an efficient multiplication program possible. A multiplication of two 32–bit numbers consists of 32 instructions, so–called multiplication steps. If the operands are unsigned numbers, the product z is given, given the multiplicand y and the multiplier x:

$$z = 2^{31} x_{31} y + 2^{30} x_{30} y + 2^{29} x_{29} y + \dots + 2 x_1 y + x_0 y$$

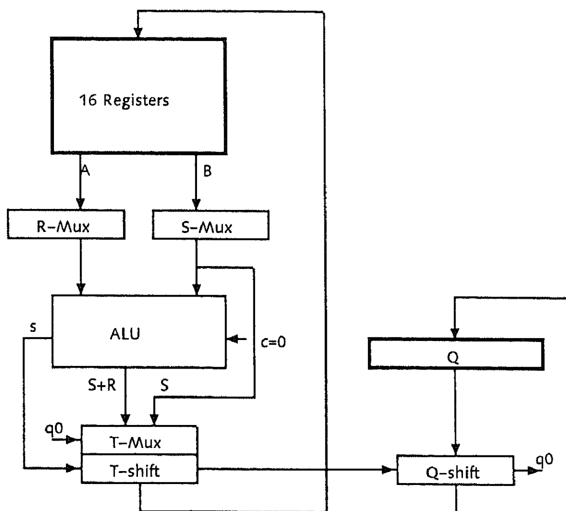


Fig. 3. Data Paths for Multiply Step

Each *multiply step* consists in the addition of a term to the partial product and the shifting of multiplier and partial product by one bit position. Initially, y is stored in a register (say R1, and remains constant), x is stored in the Q-register (specifically provided for this purpose), and forms together with another general register, say R0, a double register that ultimately contains the product (64-bit). This double register (R_0Q) can be shifted as a whole by one bit position. the multiply step is given by

```
IF x0 = 1 THEN z := z+y END;
z,x := z,x DIV 2
```

and the relevant data path is shown in Fig. 3. Before step i ($0 \le i \le 32$), the multiplier is represented by $Q_{31-i} \ldots Q_0$, and the partial product by $R0_{31} \ldots R0_0, Q_{31} \ldots Q_{32-i}$.

Multiplication of signed numbers differs only in its last step, where the multiplicand is subtracted rather than added.

5. Division

The Q-register and its data path and shift-mux are also used in the process of division. The Q-register actually derives its name from the fact that it holds the quotient. Division (for unsigned numbers) is also performed by 32 identical instructions, called *divide steps.* Initially, the double register formed by a general purpose register (say R0) and the Q-register holds the dividend (R0 = 0), and another register (say R1) holds the divisor (and remains constant). Each divide step is given by

```
r,q := 2*(r,q);
IF r ≥ y THEN r := r-y END
```

where r is the remainder (initially the dividend x), q is the quotient (initially 0), and y the divisor. Before step i ($0 \le i \le 32$), the remainder is represented by $R0_{31} \ldots R0_0, Q_{31} \ldots Q_i$, and the quotient by $Q_{i-1} \ldots Q_0$. The data path relevant in a divide step is shown in Fig. 4.

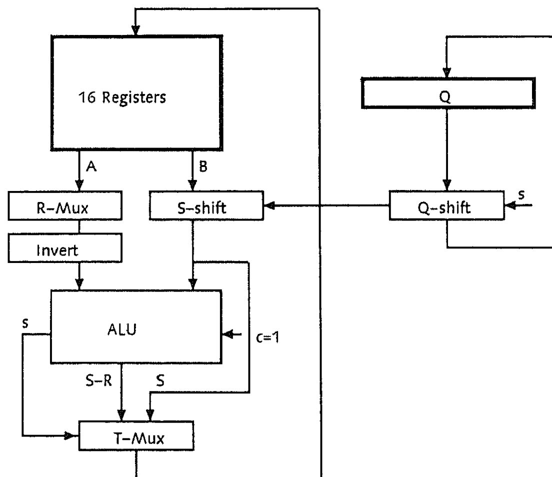


Fig. 4. Data Paths for Divide Step

6. The Control Unit

The control unit contains the register I holding the instruction to be interpreted, the register PC holding the address of the instruction to be fetched, and the (6-bit) register CC holding conditions resulting from ALU operations. Interpretation of an instruction and fetching the subsequent instruction proceed concurrently.

The source of the address of the next instruction is selected by the A-multiplexor. It is typically the current instruction's address plus one, except in the case of sequence control instructions *(jumps)*. The control unit therefore also contains an incrementor. Control instructions are distinguished from other instructions by a 1-bit field of the instruction format (k). In their case, the source of the A-mux may be the next-address field of the control instruction itself, and the selection between the incremented address PC+1 and the jump address IR.adr depends on a condition. It is selected by the *condition mask*, a field of the control instruction, from several conditions available in the *condition code* register CC. These conditions are results of the previous instruction (except x):

z	ALU output zero
n	ALU output negative
v	ALU overflow
e	ALU extend

The condition for taking the next instruction address from the current instruction (jump) is met, if at least one condition of the CC register is set which is not masked by its corresponding mask bit of the control instruction. The sense of the decision (to jump or not to jump) is inverted, if the i bit of the instruction is set.

A control (jump) instruction may additionally store the current value of PC is a stack of addresses, and act as a *jump to subroutine*. This stack may be selected by the A-mux. Its selection represents a *return from subroutine* instruction. The address-stack is only four elements deep, and no overflow indication is provided. The data path of the control unit is shown in Fig. 5.

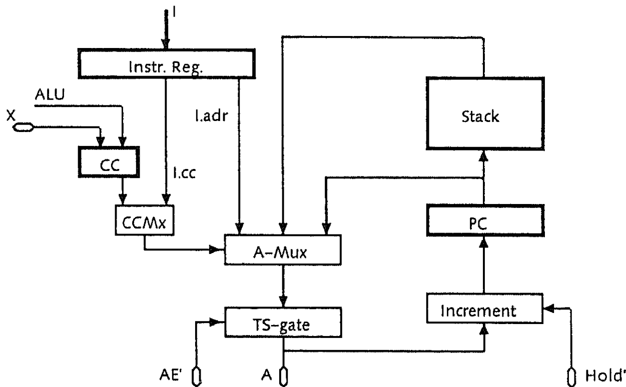


Fig. 5. Control Unit

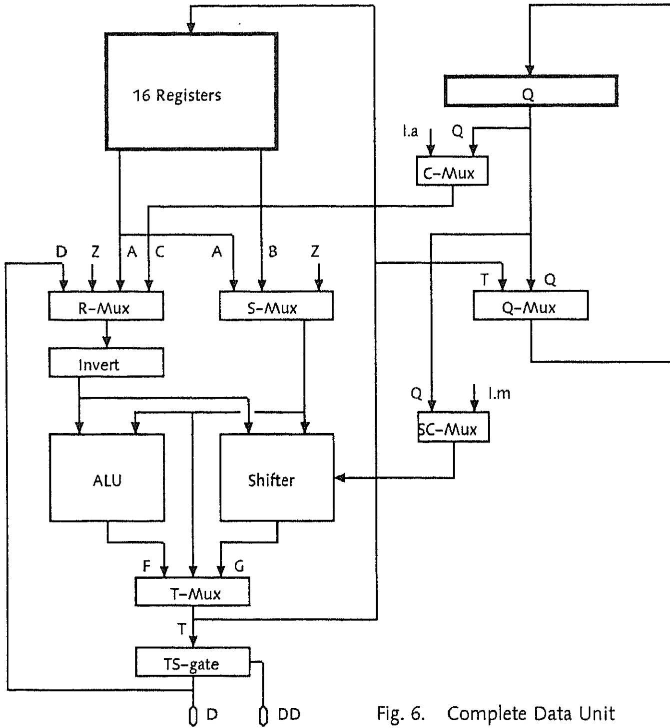


Fig. 6. Complete Data Unit

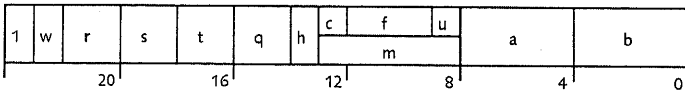


Fig. 7. Format of Data Instructions

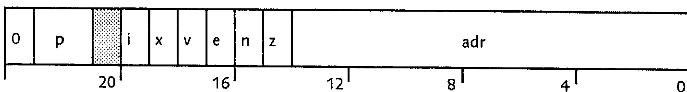


Fig. 8. Format of Control Instructions

7. Instruction Format

7.1. ALU/Shift – Operations

field	unit	value	result	comment
k		1		enable ALU/shift clock
w	registers	0		write enable
		1		write disable
r	R–mux	0	Q_0=0: 0, Q_0=1: A	multiply step
		1	A	
		2	C	constant or Q
		3	D	data input, DD = 1
s	S–mux	0	Z	zero
		1	B	
		2	B shifted left	divide step, $S_0 = Q_{31}$&I.q_1
		3	A	
t	T–mux	0	F	
		1	e=0: S, e=1: F	divide step
		2	F shifted right	multiply step, T_{31} = e
		3	G	
q	Q–mux	0	Q	
		1	T	
		3	Q shifted left	divide step, Q_0 = e
		3	Q shifted right	multiply step, $Q_{31} = F_0$
h	SC–mux	0	I.m	shift count
		1	Q	Q_0–Q_4
	C–mux	0	Q	
		1	I.a	zero–extended constant
c	ALU	0/1		carry input
u	ALU	0	R	
		1	~R	invert R input
F	ALU	0	R + S + I.c	unsigned
		1	R + S + CC.v	
		2	R + S + I.c	signed
		3	R + S + CC.v	
		4	R OR S	
		5	R AND S	
		6	R XOR S	
		7	FFO(R)	first one bit
m	Shifter	0 .. 31		shift count, if I.h = 0
a	registers	0 .. 15		register A address
b	registers	0 .. 15		register B and write address

Define shifter input X as $X_i = S_i$ and $X_{i+32} = R_i$ for i = 0 ... 31. Then $G_i = X_{i+32-m}$

7.2. Control Instructions

field	unit	value	result	comment
k		0		disable ALU/shift clock
p	A–mux and stack	0		undefined
		1	stack, pop	return subroutine
		2	I.adr	conditional jump
		3	I.adr, push	cond jump subroutine
i	CC–mux	0	jump if cond = 1	
		1	jump if cond = 0	invert condition
x	CC–mux			external
v	CC–mux			carry/overflow
n	CC–mux			negative
e	CC–mux			extend
z	CC–mux			zero
adr		0 .. 2^{14}–1		jump address

cd = (CC.x & I.x) OR (CC.v & I.v) OR (CC.n & I.n) OR (CC.f & I.f) OR (CC.z & I.z)

Fields x ... z are called the condition mask field.
Upon reset, the output of the A–mux is 0.
If I.k = 1, the adr–mux selects the PC register as input.

8. External Signals and Pin Assignments

D0–D31	I/O	data
A0–A13	O	address
I0–I23	I	instruction
X	I	ext. condition
Hold'	I	PC increment carry input
AE'	I	address output enable
DD	O	data direction
Reset	I	
CLK	I	
VCC (VDD)		
GND (VSS)		

Pin	Pos	Pin	Pos	Pin	Pos	Pin	Pos	Pin	Pos	Pin	Pos
D0	N4	D8	L1	D16	G2	D24	C2	A0	N12	A8	J12
D1	M4	D9	K2	D17	F1	D25	B1	A1	M11	A9	J13
D2	N3	D10	K1	D18	F2	D26	A1	A2	M12	A10	H12
D3	M3	D11	J2	D19	E1	D27	B3	A3	M13	A11	H13
D4	N2	D12	J1	D20	E2	D28	A2	A4	L12	A12	G12
D5	M2	D13	H2	D21	D1	D29	A3	A5	L13	A13	G11
D6	L2	D14	H1	D22	D2	D30	B4	A6	K12	AE'	M9
D7	M1	D15	G1	D23	C1	D31	A4	A7	K13	Hold	N11
I0	A7	I8	A10	I16	B13	PHI1	A5				
I1	C7	I9	B10	I17	C13	PHI2	B6				
I2	B7	I10	A11	I18	D12	Reset	N9				
I3	A8	I11	B11	I19	D13	XC	N7				
I4	B8	I12	A12	I20	E12	DD	M5				
I5	C8	I13	B12	I21	E13	VDD	G3				
I6	A9	I14	A13	I22	F12	VSS	G13				
I7	B9	I15	C12	I23	F13	VSS	A6, M10				

Fig. 9. Pin Assignment in 100–PGA Package

9. Algorithmic Definition of Flintstone

```
TYPE BIT  = CARDINAL;  (* 0 .. 1 *)
   ADDR = CARDINAL;  (* 0 .. 2↑14–1 *)
   WORD = ARRAY [0 .. 31] OF BIT;

      Condition =
           RECORD
              x, v, e, n, z: BIT
           END ;

      Instruction =
           RECORD
              k: BIT;
              p: [0 .. 7];
              i, x, v, e, n, z: BIT;
              adr: ADDR;
              w: BIT;
              r, s, t, q: [0 .. 7];
              h, c: BIT;
              f: [0 .. 7];
              u: BIT;
              m: [0 .. 31];
              a, b: [0 .. 15]
           END ;
```

```
(*input signals*)
CONST Hold = 1;    (*PC increment carry input*)
VAR I0, I1: CARDINAL;  (*instruction*)
    XC:   BIT;    (*external condition bit*)

(*the following input signals are not represented in the simulator:
    AE:   BIT;    instr. address output enable, active low
    RESET: BIT;   forces address output to zero*)

(*output signal*)
VAR DD:   BIT;    (*Data Direction for D. D = 0: output*)

(*input/output signals*)
VAR D: WORD;
    A: ADDR;

(*registers*)
VAR R:  ARRAY [0 .. 15] OF WORD;   (*register RAM*)
    Q:  WORD;                      (*multiplier/quotient register*)
    St0, St1, St2, St3: ADDR;      (*return address stack*)
    PC: ADDR;                      (*program counter*)
    CC: Condition;                 (*condition code register*)
    I:  Instruction;               (*instruction register*)

(*components external to the processor*)
VAR Mem0, Mem1: ARRAY [0 .. MemSize-1] OF CARDINAL;

PROCEDURE ROR(x, n: CARDINAL): CARDINAL;
  (*rotate x right by n bits*)
END ROR;

(*---------------------------------------------------*)

PROCEDURE Step;
  VAR k, count: CARDINAL;
  (*signal variables*)
    a, b, r, r1, s, f, g, q, t: WORD;
    cond: BOOLEAN;
    cc: Condition;

  PROCEDURE Add(carry: BIT; signed: BOOLEAN);
    VAR k, sum, lastcarry: CARDINAL;
  BEGIN
    FOR k := 0 TO 31 DO
      sum := r1[k] + s[k] + carry;
      f[k] := sum MOD 2; lastcarry := carry; carry := sum DIV 2
    END ;
    IF signed THEN
      cc.v := (carry + lastcarry) MOD 2;
      IF I.s = 2 THEN cc.e := (carry + b[31]) MOD 2
      ELSE cc.e := (carry + I.u) MOD 2
      END
    ELSE
      cc.v := carry;
      IF I.s = 2 THEN cc.e := (carry + r[31]) MOD 2
      ELSE cc.e := (carry + r[31] + s[31]) MOD 2
      END
    END
  END Add;
```

```
   PROCEDURE Set(VAR w: WORD; n: CARDINAL);
     VAR i: CARDINAL;
   BEGIN (*convert n to binary representation*)
     IF n < 32 THEN cc.v := 0 ELSE cc.v := 1; n := 0 END ;
     FOR i := 0 TO 4 DO
       w[i] := n MOD 2; n := n DIV 2
     END ;
     FOR i := 5 TO 31 DO w[i] := 0 END
   END Set;

BEGIN (*interpretation step; latch instruction in I*)
     I1 := ROR(I1, 8); I.b := I1 MOD 16; I.adr := I1 MOD 256;
     I1 := ROR(I1, 4); I.a := I1 MOD 16; I.u := I0 MOD 2; I.m := I0 MOD 32;
     I.adr := I0 MOD 64 + I.adr;
     I0 := ROR(I0, 1); I.f := I0 MOD 8;
     I0 := ROR(I0, 3); I.c := I0 MOD 2;
     I0 := ROR(I0, 1); I.h := I0 MOD 2;
     I0 := ROR(I0, 1); I.z := I0 MOD 2; I.q := I0 MOD 4;
     I0 := ROR(I0, 1); I.n := I0 MOD 2;
     I0 := ROR(I0, 1); I.e := I0 MOD 2; I.t := I0 MOD 4;
     I0 := ROR(I0, 1); I.v := I0 MOD 2;
     I0 := ROR(I0, 1); I.s := I0 MOD 4; I.x := I0 MOD 2;
     I0 := ROR(I0, 1); I.i := I0 MOD 2;
     I0 := ROR(I0, 1); I.r := I0 MOD 4;
     I0 := ROR(I0, 1); I.p := I0 MOD 4;
     I0 := ROR(I0, 1); I.w := I0 MOD 2;
     I0 := ROR(I0, 1); I.k := I0 MOD 2;

  (*register output*)
     a := R[I.a]; b := R[I.b];

  (*R/C-Mux*)
     CASE I.r OF
       0: (*mul step*)
          IF Q[0] = 0 THEN
            FOR k := 0 TO 31 DO r[k] := 0 END
          ELSE r := a
          END
     | 1: r := a
     | 2: IF I.h = 0 THEN r := Q ELSE Set(r, I.a) END
     | 3: r := D
    END ;

  (*S-Mux*)
     CASE I.s OF
       0: FOR k := 0 TO 31 DO s[k] := 0 END
     | 1: s := b
     | 2: (*div step*)
          s[0] := q[31];
          FOR k := 1 TO 31 DO s[k] := b[k-1] END
     | 3: s := a
     END ;

  (*Inverter*)
     IF I.u = 0 THEN r1 := r
     ELSE  (*invert r*)
       FOR k := 0 TO 31 DO r1[k] := 1 - r[k] END
```

```
    END ;

  (*ALU*)
    CASE I.f OF
       0: Add(I.c,  FALSE)
    | 1: Add(CC.v, FALSE)
    | 2: Add(I.c,  TRUE)
    | 3: Add(CC.v, TRUE)
    | 4: (* AND *) cc.v := CC.v; cc.n := CC.n;
           FOR k := 0 TO 31 DO
              IF r1[k] = 0 THEN f[k] := 0 ELSE f[k] := s[k] END
           END
    | 5: (* OR *) cc.v := CC.v; cc.n := CC.n;
           FOR k := 0 TO 31 DO
              IF r1[k] = 1 THEN f[k] := 1 ELSE f[k] := s[k] END
           END
    | 6: (* XOR *) cc.v := CC.v; cc.n := CC.n;
           FOR k := 0 TO 31 DO
              f[k] := (r1[k] + s[k]) MOD 2
           END
    | 7: (* First One Bit *) k := 0;
           WHILE (k < 32) & (r1[k] = 0) DO k := k+1 END ;
           Set(f, k);
           IF k = 32 THEN cc.v := 1 ELSE cc.v := 0 END
    END ;

  (*Shifter*)
    IF I.h = 0 THEN count := I.m
    ELSE count := 0;  (*Q[0..4]*)
       FOR k := 4 TO 0 BY -1 DO count := 2*count + Q[k] END
    END ;
    k := 0;
    WHILE k < count DO g[k] := s[k+32-count]; k := k+1 END ;
    WHILE k < 32 DO g[k] := r[k-count]; k := k+1 END ;

    (*T-Mux*)
    CASE I.t OF
       0: t := f
    | 1: (*div step*)
           IF cc.e = 0 THEN t := s ELSE t := f END
    | 2: (*mul step*)
           FOR k := 0 TO 30 DO t[k] := f[k+1] END ;
           t[31] := cc.n
    | 3: t := g
    END ;

  (*Q-Mux*)
    CASE I.q OF
       0: q := Q
    | 1: q := t
    | 2: (*div step*)
           q[0] := cc.e;
           FOR k := 1 TO 31 DO q[k] := Q[k-1] END
    | 3: (*mul step*)
           FOR k := 0 TO 30 DO q[k] := Q[k+1] END ;
           q[31] := f[0]
    END ;
```

ETH-Zentrum
CH-8092 Zürich

```
(*CC-register*) k := 0;
  WHILE (k < 32) & (t[k] = 0) DO k := k+1 END ;
  IF k < 32 THEN cc.z := 0 ELSE cc.z := 1 END ;
  cc.n := t[31]; cc.x := XC;

(*A-Mux*)
  cond :=
     (I.x = 1) & (CC.x = 1) OR (*external*)
     (I.v = 1) & (CC.v = 1) OR (*carry/overflow*)
     (I.n = 1) & (CC.n = 1) OR (*negative*)
     (I.e = 1) & (CC.e = 1) OR (*ext*)
     (I.z = 1) & (CC.z = 1); (*zero*)
  IF I.i = 1 THEN cond := NOT cond END ;
  IF (I.k = 1) OR NOT cond THEN A := PC
  ELSIF I.p DIV 2 = 1 THEN A := I.adr
  ELSIF I.p = 1 THEN A := St0
  END ;

(*Adr-stack*)
  IF I.k = 0 THEN
    IF I.p = 2 THEN (*pop*)
      St0 := St1; St1 := St2; St2 := St3
    ELSIF I.p = 3 THEN (*push*)
      St3 := St2; St2 := St1; St1 := St0; St0 := PC
    END
  END ;

(*PC-register*)
  PC := A + Hold;

(*output*)
  IF I.k = 1 THEN
    IF I.w = 0 THEN R[I.b] := t END ;
    Q := q; CC := cc;
    IF I.r = 3 THEN DD := 1 ELSE DD := 0; D := t END
  END

END Step;
```

References

1. N. Wirth. The Personal Computer Lilith. *Proc. 5'th Int'l Conf. on Software Engineering,* San Diego, March 1981.

2. H. Bonnenberg. Data Sheet for the Microcontroller Flinstone. Inst. für integrierte Systeme, ETH. Zürich, Oct. 1989.

3. The Am2900 Family Data Book. Advanced Micro Devices, Inc., 1976.