Unlike engineering, computer programming has no well established methods of attack. Even when it has, these methods are disregarded too often. How can an engineer's approach be made to computer programming?

Computer programming as an engineering discipline

by Prof. C. A. R. HOARE, M.A.

When the Institution of Civil Engineers was granted its Royal Charter in 1828, Thomas Tredgold defined civil engineering as 'being the art of directing the great sources of power in Nature for the use and convenience of man'. This definition can be readily adapted to computer programming by replacing the 'power in Nature' by the 'power of the electronic digital computer'.

The most important point of Tredgold's definition is the emphasis on the use and convenience of man, and the same applies equally well to computer programming. However, I want to concentrate more on methods than on purpose and emphasise the displinary aspects of the subject.

The word 'discipline' suggests a more-or-less systematic body of knowledge capable of transmission from one generation to the next; it also implies an established method of applying that knowledge to solve a problem. It implies an agreed terminology in which a project can be described and discussed and in which decisions can be taken and communicated. It also implies a standard sequence of steps through which a project progresses and a method of subdividing the labour of a large task so that it can be carried out by large teams of people of varying skills over a long period.

It is in these aspects of discipline that computer programming has hitherto fallen below the standards of longer established branches of engineering. However, in the universities and in industry these shortcomings have been recognised, and solutions are being developed. The basis for these solutions is a clearer understanding of the stages through which any engineering project will progress.

Specification

It is characteristic of engineering that the problems which it undertakes are never clearly defined to begin with. It is the duty of a good engineer to elucidate the problem, not only to himself but to his customer. He must do this successfully right at the beginning of the project. If he fails or makes a mistake at this stage, the true nature of the problem may come to light only on completion of the project. I fear that in computer programming we have perpetrated many such projects.

It is equally important to clarify the objectives—to formulate the criteria by which the success of the project will be judged. Not all these criteria can be quantified but they are no less important for that. Of course, the criteria will conflict with one another, so they should be qualified by some definition of an acceptable level of achievement, possibly in the form of a priority ordering.

The next step is, in the light of the objectives and their relative priorities, to choose, or invent if necessary, a solution technique. A good engineer is already familiar with a wide range of useful techniques and knows the merits and demerits of each of them in various circumstances. The computer programmer usually has to invent a solution technique for each problem he meets. Unfortunately, at present he too often reinvents an inferior version of a known technique, rather than taking the trouble to find out how the problem has been solved previously.

The consequences of the choice of technique should then be worked out in sufficient detail to be able to predict with reasonable confidence the characteristics of the final product, and it should be determined how well the product will meet its defined objectives. If such confidence cannot be established, the previous steps must be repeated, until it is known that the target objectives are at least feasible.

The first phase culminates in a more-or-less precise specification of the product, which has been agreed with a customer or his representative. For many engineers, architects, painters, or sculptors the specification may be a set of plans and drawings. It is one of the unfortunate aspects of computer programming that there is no intuitively acceptable method of summarising the major external characteristics of a computer program by means of a 2-dimensional picture or a 3-dimensional model.

Design

The most general precept for carrying out a complex task is 'divide and conquer'. Each task is split into its major subtasks, and the interfaces between the subtasks are defined with sufficient clarity for each subtask to be undertaken by somebody else. Then, if the subtask is a large one, it can again be split and delegated to a team of designers and implementers. All the steps of the specification phase are repeated on each subtask: elucidation of the problem, clarification of objectives, choice of solution technique, evaluation of cost and effectiveness, and agreement of specification between the designer and the implementer.

When this process is complete, it is usual to re-evaluate the entire specification to ensure that the objectives and schedules can be met. If not, negotiations with the customer are re-entered.

In traditional engineering disciplines the design process has become adequately formalised. Each team member is familiar with the ways in which tasks are broken down into subtasks, the places where the interfaces are reasonably narrow and the manner in which they are defined in detail to eliminate the possibility of subsequent misunderstanding. It is here that computer programming is very weak. The division of a project into subtasks is often ill advised. The interfaces are wide, complicated and inefficient, and,

Tony Hoare is professor of computer science at the Queen's University of Belfast

This article is based on a lecture delivered to the Electronics Division of the IEE at Savoy Place on the 12th April 1972

worst of all, they are defined quite vaguely or even inaccurately. Recent academic research into systematic program-design methods is specifically oriented toward this problem.

Implementation

In theory, the implementation of a project should be purely a matter of routine: the application of specified methods on a predetermined time scale towards a known conclusion. But, in practice, something always goes wrong. Machines, deliveries and people break down. New problems come to light. The customer, given half a chance, will change his mind. The engineer must therefore keep control of all activity and take corrective steps as soon as it is apparent that they are needed. It is persistent attention to both principle and detail during implementation which distinguishes the good engineer from the mediocre one and makes the difference between outstanding success and failure or near failure of a project.

Towards the end of implementation comes assembly and commissioning. Again, this should be a matter of routine, with a few minor adjustments. In traditional branches of engineering this is usually the case. But in computer programming, it is notorious that this step is more difficult, expensive, time-consuming and unpredictable than any previous step. It is the prolongation of assembly and commissioning that most often makes it impossible to deliver a large programming product on time, at the stated cost and to specification. But the main reason for this difficulty lies not in this step itself, but in the inadequacy of the earlier phases of specification and design, and it is there that a solution must be sought.

Use

The residual errors remaining in a product will be detected and corrected when the results of the project are put into use. It is inevitable that from experience in use, certain potential improvements will be found which can be made relatively easily without disrupting the general structure of the product. As time progresses, the environment in which the product works will change and will necessitate more substantial adaptation of the design. A well engineered product has reserves of strength and flexibility which make adaptation possible. Finally, of course, a good engineer learns from his experience and resolves to do even better next time—and often succeeds in doing so.

It is in use that the deficiencies of computer programming, as compared with better established engineering disciplines, become most disastrously obvious. First, there are often very many residual errors in a program when it goes into use. The cost of their detection and correction is great enough, but the cost of undetected error is incalculable. Secondly, the quality of the delivered product is often so abysmally low that improvements of a factor of two or more must, and often can, be made almost immediately after delivery. Further improvements have to be made continuously by increasingly severe degrees of reconstruction of the product. Thirdly, in the face of the necessary continuous stream of necessary corrections and improvements, it is not often that the customer will undertake further disruption by attempting to adapt the product to his changing needs, and the hideous task of adapting his adaptations to the subsequent stream of necessary changes. Finally, it is my experience that computer programmers rarely do better next time. Even those who are given a similar project again almost always suffer from overambition and, in the end, do worse.

Electronic system

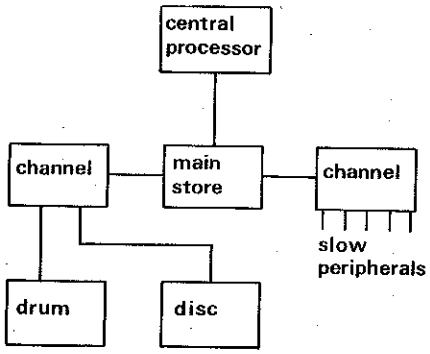
An example of the principle of 'divide and conquer' in engineering is the design of a new computing system (Fig. 1). There are two ways of interpreting this diagram. From the point of view of the external physical structure, the boxes indicate cabinets, and the lines indicate cableforms. But from the point of view of the implementer, the boxes represent subtasks which can be carried out by separate teams, and the lines indicate the interfaces between the subtasks. The close correlation between the physical layout of the product and the logical subdivision of the tasks can, and must, be deliberately turned to advantage by the engineer. Unfortunately it is much more difficult in computer programming to establish and preserve such a correlation.

An important feature of a general-configuration diagram is that each of its tasks can be further subdivided (Fig. 2). The central processor is shown as consisting of control, registers and function unit. Again, this subdivision will probably be reflected in the layout of boards within the central-processor cabinet.

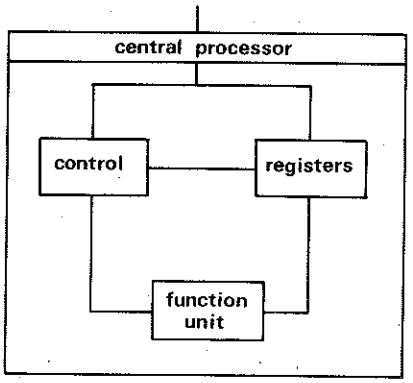
Such diagrams may appear to be so vague as to be useless. But, in fact, the important lines are not those shown but the lines that are not there. For example, the main information given by Fig. 2 is that there is no direct connection between the outside world and the function unit. The absence of a connection is a major simplification of the logical design and implementation process and will probably reduce the cost of construction and increase the reliability and ease of maintenance of the product. If the customer demands an order-code facility which would require the insertion of a new direct connection of this sort, the wise engineer will go to great lengths to dissuade him.

The benefits of splitting a task into subtasks, corresponding to splitting a program into modules, have also been sought by computer-program designers, who also often illustrate the connections between modules by means of a diagram. But, when conscientiously drawn, such diagrams often have the appearance of Fig. 3. This is because program designers have failed to realise that the absence of an interface line is far more important than its presence. Indeed, in the general case, the number of interfaces goes up as the

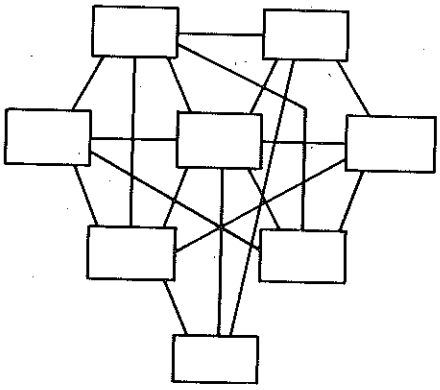
'Computer programmers rarely do better next time'



1 Subdivision of tasks into subtasks in a computing system. The lines between the boxes represent interfaces



2 Further subdivision of central processor in Fig. 1



3 In modular construction the number of interfaces increases rapidly

square of the number of modules, so the program designer's project will rapidly become too complicated to control unless he deliberately accepts the discipline of a sparser structure of interfaces.

Another way

There is another entirely different way in which the task of the design of electronic equipment is split into stages—a way which cannot be illustrated by diagrams consisting of boxes and lines. This method consists in a complete alteration of the conceptual level and framework within which the design is carried out; namely, the transition between the logical design and the physical design of the equipment.

After the whole system has been, so it seems, specified and designed in complete detail by the logic designers, the design process starts again at a different level of discourse. A whole new class of design decisions must be made on the physical construction, cabinets, racks, boards, components, sockets, wiring etc. The logic must be split between boards, the backwiring must be specified, the components must be laid out on the boards and the tracking between them must be drawn.

The division of the design task between the logical and physical aspects is one of the major intellectual tools in the control of complex engineering projects. Of course, the division is not absolute. The logic designers must know in principle that their designs satisfy the constraints imposed by physical layout, and the layout designers sometimes introduce significant changes to the design, for example, to secure uniformity of boards. But the important point is that the mass of detail involved in the physical layout may be ignored at the logic-design phase, and the logical characteristics of the design may be ignored during the stage of physical layout.

This important distinction between the logical and physical characteristics of an engineering design can be carried over to the design and implementation of computer programs and forms the basis for the solution of the major problems encountered when the program is put to use.

Program design

The application of the methods of the design of electronic systems to the development of computer programs can be illustrated by finding all the prime numbers less than 1 000 000. For efficiency, division is to be avoided. A very efficient method of finding prime numbers due to the Greek mathematician Eratosthenes is therefore used. It is based on the principle of a sieve, in which the numbers from 2 to 1 000 000 are put in order. All the nonprimes are gradually sifted out until only prime numbers are left (Fig. 4).

Note that in striking out the multiples of p, we may start at p^2, since all lower multiples of p will already have been removed.

The basic structure of the algorithm is shown in Fig. 5. As in the block diagram of the computer configuration, this flow diagram can be interpreted in two ways. In one view, the boxes represent parts of the computer program, and the lines represent flow of control. Alternatively, the boxes represent subtasks, and the lines represent interfaces between the subtasks. As in the diagram of the electronic system, each of the subtasks can be further analysed in greater detail. For example, the removal of multiples of p from the sieve can be described by the diagram of Fig. 6.

start with all numbers from 2 to 1 000 000
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 . . .

remove all multiples of 2
2 3 5 7 9 11 13 15 17 19 21 23 25 . . .

remove all multiples of 3
2 3 5 7 11 13 17 19 23 25 . . .

remove all multiples of 5
2 3 5 7 11 13 17 19 23 . . .

etc.

remove all multiples of 997
2 3 5 7 11 13 17 19 23 . . .

end

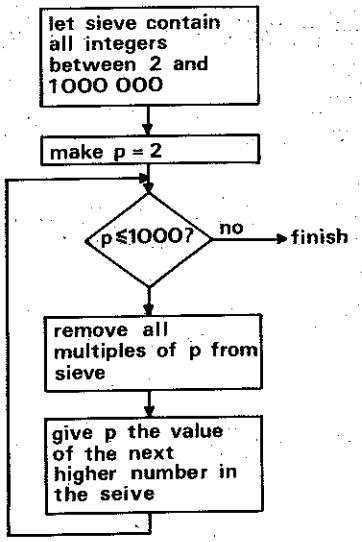
4 'Sifting' out all the prime numbers up to 1 000 000 by the method of Eratosthenes

In this way, the logical behaviour of the algorithm can be specified in complete detail in terms of addition of integers and removal of integers from a set. We have reached a stage analogous to the completion of the logic design of a computer. But, just as the logic diagram cannot actually carry out its functions until it has been realised as a physical assembly of circuits, boards, and wires, so our algorithm cannot actually be executed on a computer until further decisions on the physical representation of the data have been taken and implemented by program.

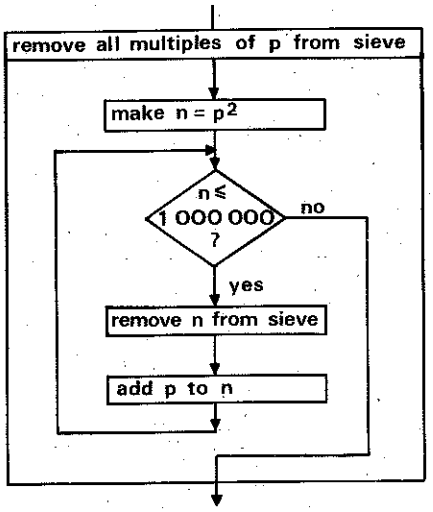
For example, Fig. 7 illustrates a decision to represent the sieve as an array of one million consecutive bits, where the nth bit is one if n is in the sieve and zero if it is not. In a computer, the million bits will stretch over many thousands of words of the computer store. Assuming that the wordlength is 10 bit, 10^5 consecutive words will be needed.

This large number of words causes another problem. To remove an arbitrary n from the sieve, it is necessary to set the nth bit of the array to zero. This requires a knowledge of which word contains this bit and where it is in the word. These can readily be computed by dividing n by the wordlength to give the right word number and taking the remainder as the bit number within that word. Unfortunately, the process involves a division, which on a computer with wordlength unequal to a power of two can be time-consuming—and is against the rules of our game anyway.

The solution to this problem is to choose an appropriate nonstandard representation for the integers involved. To do



5 Basic structure of the algorithm for Eratosthenes's method of finding prime numbers



6 Further subdivision of a box in Fig. 5

this, the integer p is represented by two words, the first of which (known as the $p$word) gives a word number of the word of the sieve in which the pth bit is found, and the second of which gives a bit number of the right bit within that word. To find the value p, it is necessary to multiply the $p$word by the wordlength and add the $p$bit. We represent n in a similar fashion.

$$p =_{\text{df}} p\text{word} \times \text{wordlength} + p\text{bit}$$

$$n =_{\text{df}} n\text{word} \times \text{wordlength} + n\text{bit}$$

This representation may be regarded as a 'mixed radix' number, rather like pounds and ounces or feet and inches.

The representation has been chosen in order to access efficiently an arbitrarily numbered bit of the sieve. Unfortunately, the choice has the result that the simple operation of integer addition becomes more complicated and can no longer be carried out by a single machine instruction. Fig. 8 shows the steps that are required to add p to n when both numbers are held in the mixed-radix representation. Fig. 9 shows the steps required for removing n from the sieve.

Similar fragments of program must be written to implement the other operations required by the algorithm. Each fragment carries out on the physical representation of the data the transformations which are analogous to the required logical operations on the abstract data. At all times, the original logic diagram (flowchart) is used as a framework and a guide, dictating not only what has to be done but also how the various parts of the program fit together and what the interfaces between them must be.

In the design of logic assemblies, this 2-stage method has become quite standardised; the stages are generally delegated to different teams, and the language in which they communicate has been formalised. In computer programming, there is no reason why the process should not extend over more than two stages. However, the formalisation and standardisation of the multistage design process has not yet been achieved.

Problems

One of the major deficiencies in computer-programming practice is that when parts of a program written by different programmers are assembled they do not work together properly. Also, commissioning is expensive, time-consuming and unpredictable and leaves many residual errors to plague the user after the program is delivered.

These problems are characteristic of any new branch of engineering. It is not so long ago that computer hardware suffered from similar troubles, although it is now quite usual for a new computer design to start working within a few days of first switch-on. Similar troubles have afflicted bridgebuilders, aircraft designers and architects, often with far more expensive and tragic results. But at least early engineers had the excuse that their troubles were due to unknown physical forces such as unsuspected wind behaviour, filaments growing in core stores or metal fatigue. But computer programmers have no such excuse.

Their raw material, the computer hardware, is more structurally simple, reliable, controllable and predictable than the material used by any other branch of engineering. There is no struggle with Nature, no groping into the unknown, no unpredictabilities other than those which arise from oversights. Computer programmers should be able to do better.

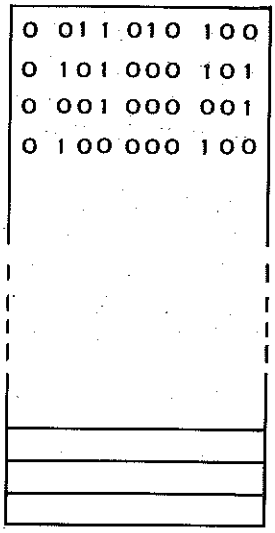
First, specifications must be better and far more attention must be paid to the quality of the final product rather than merely to its functional specification. The specifications should take into account the characteristics of known algorithms and known solution techniques and must correspond more closely to the customer's requirement and less to what he says he requires. In other words, the specification should arise from a true negotiation between programmer and customer, as it does for other better respected engineering disciplines.

Secondly, the programmer's knowledge of algorithms and techniques must be improved. Quite often in programming the difference between a good algorithm and a bad one is measured in many orders of magnitude. For example, if the programmer were unaware of the sieve of Eratosthenes and attempted to find primes by the familiar method of division, his program would have been grossly inefficient, no matter how carefully he had carried out all the other steps in the design and implementation.

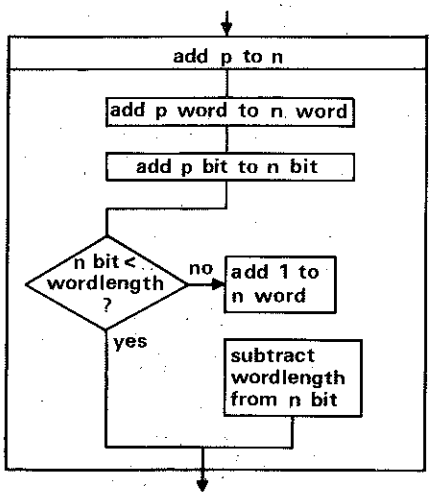
The third lesson that the programmer can learn from the engineer is more careful division of his task into subtasks, corresponding to the physical structure of his product, and more careful definition of the interfaces between his subtasks. It is here that recent academic and industrial research appears to be showing the way. Note that the interfaces between modules of a program are represented by the lines of a block diagram which indicate flow of control. It is the information which passes, as it were, along these lines that must be specified more rigorously in order to ensure that the final assembly of separately implemented parts is to be successful.

Unfortunately, the information which passes between the subtasks is not just a collection of 50 or 100 signals; it is the entire state of the computer store and external files at the time that control passes from one part of the program to another. The state of the whole machine at the moment of transition must therefore be defined as precisely as possible. In a few cases, this may be done by specifying the contents of every relevant location of store, together with the status and information content of all peripherals. But this would be most unusual. Normally, control flows across the interface on many occasions during the running of the program, and, on each occasion, the values of many of the variables are different. So the state of the machine must be specified in a general fashion by formulating certain assertions which

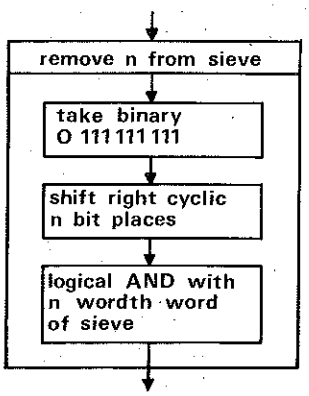
'Computer hardware is more structurally simple, reliable, controllable and predictable than the material used by any other branch of engineering'



7 Physical representation of prime-number 'sieve' by an array of 1000000 computer words



8 Adding two mixed-radix numbers



9 Removing n from the prime-number 'sieve'

describe properties of the machine store and relationships between the contents of the locations. Naur[1] called these assertions 'generalised snapshots'. A flow chart with assertions attached to the lines is known as an annotated flow chart (Fig. 10).

This method can be applied to the more detailed breakdown of the subtasks. Indeed, a subtask may be wholly specified by giving the assertions which are to hold before entry to that piece of program and after exit from it (Fig. 11).

The text of these flow charts is expressed in ordinary English, together with a few standard mathematical and programming notations to describe the actions of each part of the program and the assertions which annotate it. In practice, it is preferable to use standard mathematical and logical notations, both for the assertions and for the program. This practice aids brevity and also avoids the potential ambiguities of ordinary English.

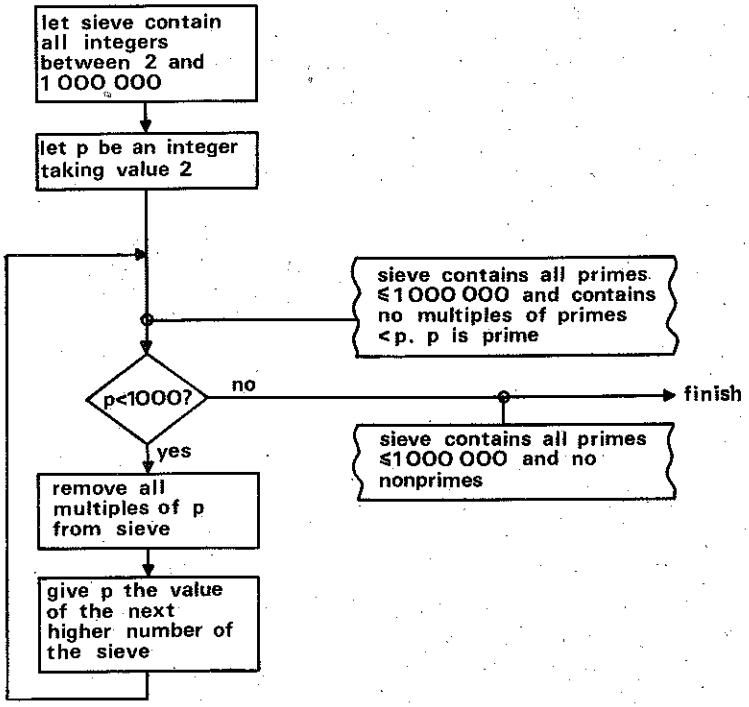
Two questions now arise:

- How do we know that the assertions on each flow line are sufficiently precise to define exactly what we require each part of the program to do for it to play its proper role in the functioning of the whole?

- How do we know that each part meets its stated specification?

The traditional answer to these questions is that we cannot know until after the complete program has been implemented, assembled, and tested. But this answer is seriously wrong, since we cannot know even then. As Dijkstra has pointed out, 'program testing can only reveal the presence of bugs, never their absence'.

' "Program testing can only reveal the presence of bugs, never their absence" '
—E. W. Dijkstra

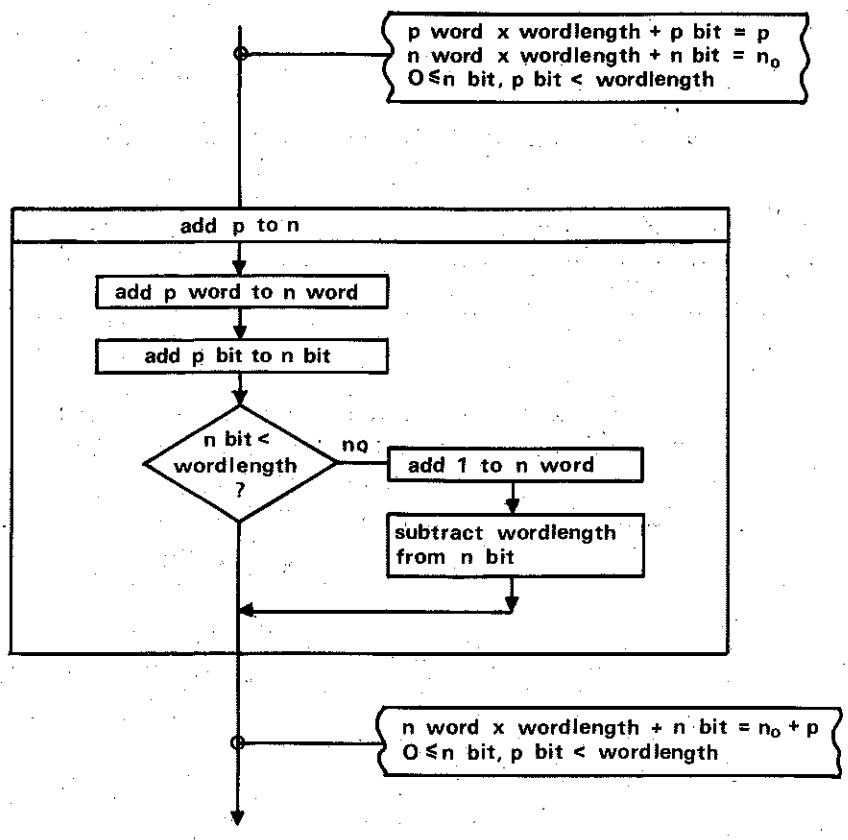


10 Annotated flowchart for specifying the interfaces in a general fashion

Answer

The correct answer is both startling and obvious. We know that our program will work because we can prove it does. Methods of reasoning which lead to valid program proofs are at present the subject of research. But there is no doubt that they are in principle feasible, although their practical application is at present unacceptably laborious.

The development of more practical proving techniques presents an exciting prospect. It is now a matter of common knowledge (and indeed common suffering) that computer programs are the most unreliably designed component of any system or environment in which they operate. But there is no reason in principle why a properly proven program should not be the most reliable of all engineering products. I look forward to the day when it is an accepted practice that whenever a program gives the wrong result we call the computer maintenance engineer—and he comes.



11 Annotated diagram for breaking down a task into its subtasks

References

1 NAUR, P.: 'Proof of algorithms by general snapshots', *Bit*, 1966, **6**, pp. 310–316

2 NAUR, P.: 'Programming by action clusters', *ibid.*, 1969, **9**, pp. 250–258

3 DAHL, O-J., DIJKSTRA, E. W., and HOARE, C. A. R.: 'Structured programming' (Academic Press, 1972)