Advanced techniques for microprocessor systems

Edited by
F.K.Hanna

ADVANCED TECHNIQUES FOR MICROPROCESSOR SYSTEMS

Published by Peter Peregrinus Ltd., Stevenage, UK, and New York

ISBN: 0 906048 31 1

Printed in Great Britain by A. Wheaton & Co. Ltd., Exeter

CONTRIBUTORS

D. ASPINALL, University of Manchester Institute of Science and Technology, UK

A.A. ALLISON, Consultant, Los Altos Hills, California, USA

B.A CARRÉ, University of Southampton, UK

A.J.T. COLIN, University of Strathclyde, UK

A.H. CRIBBENS, British Rail Research, UK

E.L. DAGLESS, University of Manchester Institute of Science and Technology, UK

M. ESCUDER, Hewlett-Packard Ltd., South Queensferry, UK

J. GALLACHER, Praxis Instruments B.V., Leiden, The Netherlands

T.J. GILPIN, Department of Industry, UK

F.K. HANNA, University of Kent, UK

C.A.R. HOARE, University of Oxford, UK

A. HOPPER, University of Cambridge, UK

JANE W. HUGHES, University of Manchester Institute of Science and Technology, UK

A.D. MILNE, Wolfsand Microelectronics Institute, University of Edinburgh, UK

H.J. MITCHELL, CAP MICROSOFT LTD., UK

D.R. NOAKS, University of Birmingham, UK

GILL RINGLAND, MODCOMP, UK

ADVANCED TECHNIQUES FOR MICROPROCESSOR SYSTEMS

Edited by F.K. HANNA

PETER PEREGRINUS LTD.

PREFACE

This book consists of a collection of essays on various aspects of microprocessor systems. The essays are based on themes addressed at an IEE Vacation School at Birmingham University in 1980.

The School was held under the aegis of PGC6, the Microprocessor Applications group of the IEE. It was organised and brought into being by the following individuals:

D. ASPINALL	(UMIST)
T.A. COX	(STL)
D. DACK	(Hewlett Packard Ltd.)
E.L. DAGLESS	(UMIST)
J. GALLACHER	(Praxis Ltd.)
T.D. HILLS	(Peter Peregrinus Ltd.)
D. NOAKES	(Birmingham University)
J.P. STUART	(Warren Springs Laboratory)

whose contribution to this task it is a pleasure to acknowledge.

F.K. Hanna
University of Kent

CONTENTS

I N T R O D U C T I O N

MICROPROCESSOR ENGINEERING

F.K. Hanna

University of Kent, UK

The microprocessor emerged, in the early 1970's, in what perhaps seems in retrospect to have been a surprisingly low key manner. Since then however, and particularly over the last few years, the development of microprocessors (and of programmable electronic devices in general) has proceeded at a vertiginous rate. This in turn has, rather rapidly, brought into being a major new branch of engineering, concerned with the design and production of microprocessor-based systems. It is to aspects of this emergent subject that the essays in this book are addressed.

To a large extent, microprocessor engineering is a synthesis between, on the one hand, digital systems engineering and, on the other, computer science. The subject therefore spans quite a broad extent, and indeed it is interesting to try compiling a checklist of topics with which the well qualified practitioner in this field might be familiar. Such a list includes:-

Set Theory
Boolean algebra
Automata theory
Electronics
I.C. technology
Digital systems design
Asynchronous systems
Design of *testable* systems
Testing LSI chips
Contemporary CPU chips
Contemporary support chips
Contemporary bit-slice chips
Bus structures
Microprogramming
Multiprocessor architectures
Requirements languages
Estimating techniques
Documentation practice
Structured programming techniques
Debugging techniques
Assembly language programming
The PL/- languages
BASIC
PASCAL (and soon ADA)
Interrupt-driven programming
Concurrent programming
Synchronisation techniques
Real-time programming
Macrogenerators
Compiler construction
Portability
Formal validation

and, not forgetting of course, an appreciation of various diverse application areas.

This set of topics (which could certainly be extended) is dauntingly large. Very few practitioners are likely to be expert in all these fields. It is therefore of interest to enquire to what extent useful "subsets" of this list of topics can be identified, and how "self-contained" these specialist subsets could be.

Within the domain of large computer systems, specialisation to a subset of these topics is the accepted norm. Thus we find, for example, "programmers", "digital design engineers", "systems engineers", etc.

On the other hand, within the domain of microprocessor engineering the situation is less clear cut. One can perceive two, opposing, tendencies. The first, similar to the trend on large machines, is towards a *separation*, or compartmentalisation, of groups of disciplines and the emergence of specialists. The other, however, in the contrary direction, is a unifying influence, tending to create the need for what might be termed the "microprocessor polymath". We will in turn explore the forces influencing each of these two tendencies.

Like any computer, a microprocessor system can be described at a number of fairly distinct *"levels"*, between which, in an ideal case, there will be no interaction. Such a hierarchy of levels might be:-

ELECTRONIC
(transmission lines, rise times, circuit loading, etc.)
- Oscilloscope

LOGIC
(AND-gates, flip-flops, glitches, etc.)
- Logic Analyser

CPU ARCHITECTURE
(instruction sets, addressing structures, programs, etc.)
- Microprocessor Analyser

HIGH LEVEL LANGUAGE
(data types, control types, scoping rules, etc.)
- Interactive Debugger

ALGORITHM
(matrix inversion, z-transforms, hill-climbing)
- Numerical Analysis, etc.

This table shows, for each level of the hierarchy, typical items of concern. It also shows the type of "instrument" that might be used to monitor or debug that level.

In effect, each successive level allows a microprocessor system to be designed and described in a successively more *abstract* way. The higher the degree of craftsmanship with which the lower layers of the system have been implemented, the greater the extent to which an individual will be able to work at any given level *purely in terms of the abstractions available at that level.* For instance, when considering an AND gate at the Logic level, it should make no difference as to whether the underlying electronics is based upon a CMOS or upon a bipolar technology. Likewise, at the Algorithm level, the number

of iterations needed by a hill-climbing algorithm in seeking a particular maximum should be quite independent of whether it is programmed in BASIC or in PASCAL.

It is due to abstraction, to the way that the overall design task can be undertaken at different levels, that computer systems (which, viewed only at the lower levels, would be unimaginably complex) can be designed at all. In a word, abstraction provides both the means and the motivation for specialisation, and enables useful subsets of the list of topics given above to be identified and studied in isolation. Abstraction, and the resulting compartmentalisation of microprocessor expertise into a number of independent areas, is thus very advantageous.

Why then does one find in the domain of microprocessor engineering the claim that to successfully exploit microprocessors, it is necessary to have a broad appreciation across the whole hierarchy of levels of abstraction? Is this just an historic accident, reflecting perhaps the diverse disciplines from which present day microprocessor designers have been drawn, or is it the result of some more fundamental influence?

The answer to this question may be gleaned by considering the role of abstraction in practical situations, and reflecting on the cases where it is *not* a viable approach. We will attempt a classification of these cases. As we do so, we will note that in each instance, in order for the problem to be successfully solved or circumvented, the designer will need expertise at more than just one level in the hierarchy.

Abstractions break down

Abstractions represent idealisations: sometimes the realisation is less than perfect and the abstraction breaks down. When this happens it is necessary to be able to characterise the departure from the theoretical ideal: this will generally involve delving down to the next level below. Then either the underlying fault has to be repaired or the domain of applicability of the abstraction has to be restricted. For instance, at the digital logic level, an integrated circuit AND gate only realises the desired abstraction provided the appropriate loading rules are obeyed, and tolerances on power supplies, ambient temperature etc, are satisfied. At another level, floating-point arithmetic is necessarily (on any finite machine) going to be subject to overflow conditions. At yet another level, compilers for high-level languages may fail, perhaps by not generating code which is compact enough to fit within the memory space of a given microprocessor, or not fast enough to meet real-time constraints imposed by the problem.

Abstractions cannot be proven not to break down

This, whilst related to the previous case, is more subtle. An abstraction may be unusable because, although its realisation is believed by its authors to be perfect in every respect, it is not *proveably* so, from the point of view of a third party. These circumstances often arise with systems where human life would be endangered by failure: good examples are fly-by-wire aircraft controllers and passenger railway signalling systems. It may for instance be very difficult to *prove* (both "formally" and "politically") that a particular microprocessor integrated circuit corresponds to its published "abstraction" (the instruction set of at least one widely used microprocessor does not), or that a given high-level language compiler conforms to its specification.

In cases like this, a practical course is to make use of available abstractions in *designing* the system, but then to *implement* it directly at a lower level.

Abstraction is missing

In some cases, suitable abstractions or the tools for their realisation are not available. This may be simply because there is insufficient demand for them, as in the case of assembly languages for microprogramming particular machines. In other cases it may be because development has been bottom-up and ad-hoc: for example, it is only comparatively recently that a well defined concept of a (standard) microprocessor back-plane has emerged.

In most cases however, a lack of suitable abstractions is a reflection of a lack of suitable theoretical foundations. This is particularly to be found at the high-level languages level. Few languages as yet are suitable for programming multi-processor systems, or even for satisfactorily incorporating interrupt handling on a systematic and well structured basis.

Abstractions cannot be Interfaced to

No useful computer system functions in isolation: it must communicate with its environment. This occurs as part of its normal operation (for instance, exchanging information with transducers in a process-control environment) and it may additionally occur during debugging and commissioning by the designer. In either case it is appropriate that this communication takes place in terms of the abstractions in which the system was designed. Unfortunately, however, this is only possible in some cases.

Consider first communication between the designer and the system. As indicated in the table above, this is well catered for at all the lower levels: oscilloscopes for examining waveforms, logic analysers for examining digital logic, microprocessor analysers for examining the execution of low-level programs, etc. At the level of high level programming however it is generally very poorly catered for: hexadecimal dumps are still often the order of the day. The only significant exception is in the facilities provided by the interactive high-level programming languages, for instance APL and POP-2.

When considering communication between a microprocessor system and other parts of its environment, the designer has no choice but to work at a level of abstraction which is common to both. Often the highest such level at which this is possible is digital logic and machine code. Only rarely are higher level mechanisms provided, such as for example CAMAC standardised hardware and software.

Alternative realisations of an Abstraction

An abstraction is only intended to capture the very essence of a functional specification: it states how an (abstract) object is to behave, and *not* how this behaviour is actually to be realised. However, entering into the specification of any system are second-order requirements (often only implicitly stated) like cost, speed, reliability, etc. Each particular realisation of an abstraction will provide a different balance between these secondary factors.

Generally, there is, at all levels in the hierarchy, a wide range of possible realisations of any particular abstraction. At the electronic level, for instance, there are countless ways of realising a one-bit memory cell, whilst equally, at the top of the hierarchy, an operation like matrix inversion may be implemented by many different algorithms.

One of the degrees of freedom a designer has in choosing an implementation which, with continuing advances in VLSI technology, is now of considerable signficance, lies in the choice between an entirely hardware realisation, a microprogrammed one, or an entirely software one, or in a mixture of all three approaches. A case which nicely illustrates this choice is in implementations of floating-point arithmetic operations, where all four approaches are currently viable. An example at another level is that a program for a given high level language can be regarded as running on an abstract, high level language machine. A realisation of this abstract machine may be interpreter or compiler based, or perhaps a mixture. If interpretative, it can be interpreted by a program running at the machine code level, or (as in the PASCAL Microengine approach) directly at a microprogrammed level.

These then are the practical reasons why the microprocessor engineer needs to be able to choose at will the level or levels of abstraction most appropriate for designing and implementing a particular system. Looking at this list of reasons, it is interesting to note that most of them are *not* fundamental reasons as to why design cannot always be undertaken entirely in high level terms. Rather, most simply reflect the present immature state of development of programmable electronic components and software tools: these elements may be unreliable, unvalidated, poorly interfaced, or sometimes just simply unavailable. As time proceeds, we may hope to see many of these limitations fade away. Not all, however, will do so. It is evident that as newer and more complex integrated circuits are developed (associative memories, distributed intelligence memories, mask-configurable VLSI chips, etc) and as higher-level algorithms are "encapsulated" in integrated-circuit form (DDC adaptive controllers, list-processing mechanisms, communications protocol handlers, syntax analysers, etc), and as "unconventional", non Von Neumann architectures are explored (data flow machines, associative pattern recognisers, array processors, etc) so there will continue to be a role for designers able to comprehend and work with abstractions at all levels.

The essays of which this book is composed address aspects of microprocessor engineering at all levels of abstraction, and from many different viewpoints. For convenience they have been grouped under four major headings:

General considerations,
Devices, subsystems and standards,
Distributed and multiprocessor systems,
Theory and techniques in programming,

although of course this can only be a very approximate classification.

I

GENERAL CONSIDERATIONS

This first group of essays places the design of microprocessor-based systems in its full context. The essays are especially valuable in that they serve to make explicit some of the intuition, background experience and working assumptions upon which the expert manager, engineer or programmer in this field relies.

The essays need little introduction. The first one traces the evolution of a microprocessor based product through from its inception (as an abstract human need), through its realisation, to its eventual end, and notes some of the many pitfalls on its route.

The second essay, on software engineering, identifies those aspects of experience gained (the hard way) during three decades of programming mainframe computers, which are equally relevant in programming microcomputers. Especial attention is devoted to methods of realistically estimating the manpower resources needed for this latter task, and of adequately documenting the resultant product.

Testing and maintenance is the main theme covered in the last essay of this group. Experience is showing that maintenance of microprocessor based systems poses exceptional problems and that the need for maintenability must be taken into account from the earliest stages of the design process.

PRODUCT LIFE CYCLE

D. Aspinall

University of Manchester Institute of Science and Technology, U.K.

INTRODUCTION

The dictionary definition of life cycle is : "Series of forms of an organism between successive occurrences of a given form". For the purpose of this lecture the organism is the product based upon microelectronics technology. The various forms of the product include :

- Human Need
- Knowledge and skills of microelectronics engineering
- Embryonic specification of product
- Detailed specification of product
- Abstract design information
- Detailed design information
- Manufactured Product
- Inspected Product
- Commissioned Product
- Tested, Packaged, Documented Product
- Used Product
- Faulty or Damaged Product
- Modified Product (New Human Need)

The abstract concept of a human need is fertilised by contact with the skills of a microelectronics engineer to produce the embryonic specification of a product. The human needs are all pervasive since the technology may be applied to most situations in which human beings communicate information in the form of memories, ideas or skills to one another or through electro-mechanical artefacts during the manufacture of new things.

Many embryos will be stillborn, due to an early realisation of the economic factors or that the technology is as yet unable to satisfy the need in full.

Live embryos will pass to a formal detailed specification of the product in a form which states the exact need to be satisfied and also permits the microelectronics engineer to commence the phase which leads to the abstract design information form. In this form the engineer will represent the information processing algorithms to realise the need. These will be represented in a language which is more suited to a description of the human requirements rather than the detailed implementation requirements.

Before the product is in the detailed design form, to enable manufacture, the engineer must make some important decisions in choosing the implementation strategy. We will presume that the engineer has decided to implement the algorithms by the interworking of a microprocessor with the peripheral equipment of the product, together with suitable programs to be developed to run on the microprocessor-memory pair.

Three separate design activities follow this decision :

1. Design the hardware of the product
2. Design the system's software
3. Design the product software

The system's software is the vital link between the programmer of the product software and the hardware upon which this will run. The system software includes the development facilities to enable the verification of the product software.

Hardware design information for manufacture will pass to the pre-production manufacturing activity which will manufacture, inspect and verify the hardware.

Eventually the software will pass through design, manufacture, inspection and verification until there is sufficient confidence to bring the software and the hardware together to complete the commissioning of the product.

At this stage in the cycle it is possible to initiate a repetition of the manufacturing activities to produce replicas of the commissioned form which are, in turn, the products to be shipped to customers. The form which they take prior to shipment may be described as the Tested, Packaged, Documented Product. Thus there need to be separate activities to establish the appropriate product test procedures and to establish the packaging which the market expects and last, but by no means least, to prepare extensive, succinct documentation to market the product and to support its use.

The used product form may well be the final form of the product cycle. However, it is likely that the product will occasionally become damaged or faulty or indeed may be modified to suit new human needs. If these new needs are significant then the whole cycle may begin again.

This life cycle is typical in many respects to that of most products of human ingenuity. The uniqueness arises from the all pervasive nature of the application of microelectronics-based products and also from the range of options available for the translation from abstract to detailed design. Furthermore, the components of the product, hardware and software, must separately pass through several forms before they come together as a product for shipment. For a healthy product it is essential to obtain an accurate specification and to exercise due care and attention to the choice of options and the management of the interworking of hardware and software.

Specification of the Product

The specification of the product is the crucial act of conception which must be correct otherwise the product may need to be aborted or may ultimately turn out to be

unwanted. There must be a nice judgement of the human need. This may arise from an inspired hunch which anticipates a need which is not obvious but which will become urgent when the means to satisfy it exists. All inventors hope to be so lucky as to have such a hunch. Most will have to settle for a proper market assessment based on well established needs.

Careful analysis of the need will lead to a specification which acts as the formal instruction to the design authority. The way in which the specification is set out is important since this vital document must be clear and unambiguous to both the human with the need and also to the human design authority. The needy must be satisfied that the end-product will be satisfactory, but must also be able to understand and appreciate its limitations. The language used in normal human communication can be misused and result in ambiguous documents. The same word or phrase can convey a different meaning to different individuals. Because of this it is necessary to be extremely careful in the preparation of the specification document. It may be necessary to define a special language for such documents.

The analysis must be carried out with the knowledge of the capabilities of the available technology. There is no point in proceeding to the preparation of a specification unless there is confidence in the ability of the technology to satisfy the need. Before the final target specification is drawn up it is advisable to conduct a preliminary design exercise to establish both the technical and economic viability of the project.

The target specification should be final and should not be modified without a thorough analysis involving both the needy and the designer. It should be carved in stone and signed in the blood of the needy and of the design authority.

Unfortunately, it is not possible to insist upon this ritual. Quite often the innovation of the product will itself modify or expand the need and will, in use, expose omissions in the specification. In these cases the target specification must allow for the introduction of new features as the use of the product demonstrates the need. It must be realised that such modifications to the specification will increase the cost of the product and such costs must be assessed before authority is granted to incorporate the modifications into the specification.

A technology which includes programmable components, such as uncommitted logic arrays and micro-computers, gives the impression that it is very tractable and that changes may be incorporated with ease. Such an impression leads to the attitude that the specification need not be complete since any deficiencies will be tactically patched over by programming during the design activity and even during commissioning and use. This attitude makes it difficult to predict the time to complete the design and commissioning of a product. It is not possible to estimate the cost of these activities and it is often the case that the resultant lengthening of the design and commissioning phases cause the project to run out of money, or if it survives this hazard, to arrive in the market place after the market has been cornered by a competitor with an inferior, lower cost product which adequately satisfies the need.

Creative engineers may find the discipline of working to a tight specification to be inhibiting and irksome. They must learn to bite the bullet and accept that it is more important to create a product within time scales and budgets which place it in the market place at the right time and at the right price than to create fancy innovations. By all means be innovative, but please only use the innovations which enable the achievement of the immediate objective and file, for future development and use, theoretical and exciting innovations which may have a profound effect upon the ability to specify the next product. Do not forget that you will only be in a position to enjoy the next project if your last product is successful and your firm is still trading.

The correct attitudes during the preparation and use of the specification are essential for the health of the product.

Abstract Design

The design process should proceed by step-wise refinement or so-called top-down design. Practical engineers, with experience and knowledge of the characteristics and limitations of the components to be interconnected to fabricate the product, may find this procedure to be cumbersome. They would much prefer to synthesise the design bottom-up. For products below a certain level of complexity it is possible to be successful by following a bottom-up procedure. However, there are limitations and dangers in the bottom-up procedure for complex products.

A product may be visualised as a set of separate elements which are interconnected. Each element is itself made up of interconnected components. By following the bottom-up approach each element is designed to achieve its function before its interconnection to other elements is designed to the same level of detail. The design of all the element interconnections may result in changes to the internal design of the elements. If this occurs it becomes difficult to manage and verify the correct working of the total product. The top-down approach forces the design of the interconnection between elements before the design of their internal functions. This is a much safer strategy in complex systems with many elements and many more interconnections.

The success of a product depends upon the quality of the design and implementation of the interconnections between standard bought-in components whether these be micro-electronic functional units or programme statements. The opposition can buy the same components. It is up to the designer to interconnect them better than the opposition. Therefore the design strategy should ensure proper consideration of the interconnections before the detailed consideration of the working of the elements to be interconnected.

The early stages of a top-down approach permit the design to be created in an implementation independent manner and allow theoretical analysis of the design to verify its operation. Each of these attributes is important.

In an explosive technology, which is constantly offering new components for the implementation of products, it is important to defer the choice of components until the last possible moment and also be in a position to exploit new components in later production runs of the product. The top-down approach permits these new components to influence the lower levels of the design without necessarily affecting the upper levels.

Verification of the design at all stages is important for obvious reasons. The problem is heightened in digital microelectronics because of the uncompromising nature of the boolean variable. There is no such thing as tolerance. The variable is either True or False. If the design produces a product in which one of the boolean variables is at the wrong value in one of the states of the product then the product is a heap of junk! If the product is in a consumer durable and several thousand have been shipped before it is realised that a variable can have the wrong value, then there is trouble. There will be a need to recall and replace all products shipped.

Thus it is essential to carry out exhaustive verification of the design. Theoretical verification of the total design is best carried out at the higher levels. Stepwise refinement makes it possible to verify the individual elements and their interconnections.

Detailed Design

The abstract design will be represented in a high level or pseudo-language which is implementation independent. Certain assumptions will have been made about the characteristics and performance of certain implementation components, particularly those which translate information between the external, often analogue, regime and the internal digital regime. Now is the time to decide the implementation strategy for the digital algorithm. In general the whole spectrum of microelectronics components is available. In practice it will be obvious whether to choose a microprocessor, rather than one of the lower level options. The question as to which processor may be difficult to answer. Before choosing a processor which is new it is necessary to establish the investment which must be made in the development environment and in the acquisition of the detailed knowledge of its characteristics and performance.

Design of the programme in a high level language may make the move to a new processor less painful but there is still much scope for development of such languages and their translation software. At the present state of knowledge it is advisable to stay with a microprocessor with which you are familiar rather than to move to one which seems to offer marginally better facilities. The move to a new microprocessor should be taken as a major policy decision. The objective in the choice of microprocessor should be to find one which will satisfy the requirements of a given class of products which embraces the range of products to be manufactured. Having made the choice, acquire all the knowledge and tools to enable efficient use of the microprocessor in the range of products to be developed. This attitude takes away some of the fun but the times are hard! Play time is over!

Having chosen the microprocessor the detailed design must progress along three distinct paths : Hardware, System Software and Product, or Application Software. The objective of the Hardware/System Software design includes the realisation of a computing machine which will run the applications or product software. In reaching this objective the designers do not need to know the intricate detail of the minutia of the applications programme unless there are critical sections which can benefit from hardware assistance. Such sections should be identified at the outset before the design of the computing machine begins. The computing machine designers will have quite enough to worry about in providing a special purpose machine which contains sufficient generality to enable the running of the programmes. When this computing machine is being used by the applications programmer to develop the product software, the prime requirement is for a reliable machine which behaves as the programmer expects. Thus there must be separate test and verification procedures which confirm the performance of the computing machine before the applications programmer attempts to use it to develop the product software. Separate test and verification procedures will then be necessary to prove and characterise the total product. All designers need to bear the test procedures in mind and make due provision. The tests may not appear explicitly in the target specification but they are implicit in all specifications.

The design information passes through many stages before the product is available for test. The electronics industry has a well established infrastructure and disciplines to take a design through to a reliable physical product. The drawing office, workshop, quality assurance inspectorate, goods inward test, product test departments, all have a well established role with clear responsibilities. A similar infrastructure is needed to translate a software design into a reliable product. The exact form of this infrastructure is still to be found and there seems to be some reluctance to admit that it is necessary since, after all, the programmes are so tractable and so easy to replicate.

The development of large software projects on main frame computers has demonstrated the need for such an infrastructure and the associated disciplines, though programmes are tractable they tend to be complex and apparently simple modifications can have serious, unforeseen consequences if they are introduced in an undisciplined manner. The microprocessor engineer can learn from the experiences of those main frame programmers who went before.

Commissioned Product

Eventually the commissioned product form will exist. The computing machine will be providing all the facilities required and the product software will have transformed it into a product which meets the target specification. One must now take stock of the situation. Many tests should be performed to fully characterise the product and to check and check again its characteristics against the target specification. Meanwhile, the hardware design information will be tidied to take account of all the modifications introduced during commissioning and to make the design data more amenable to the manufacture of a saleable product.

The documentation to enable proper use and maintenance of the hardware must be finalised for publication. The software must similarly be tidied and the documentation for the user prepared. The after sales support strategy needs to be finalised and the appropriate actions taken. Before product manufacture can begin it will be necessary to decide upon the final product test procedures and take the appropriate action. Tests of the commissioned machine should continue.

The commissioned product form is not the end of the life cycle but the end of the beginning.

Conclusions

The life cycle of the product from conception to birth of the commissioned product is that which occupies the creative microelectronics engineer. The infant product needs to be nursed through its early ills and trained to serve a proper role in society before it is shipped as a tested, packaged, documented product. This period, immediately after birth, is as important as the gestation period and demands skills, knowledge and creativity of a high calibre. Though it may not seem to be as exciting as the creation of a product at the forefront of technology, it offers scope for rich fulfillment and must not be under-rated as a vital and rewarding activity. Establishment of a healthy environment for this early period of the product's life will provide a solid base to support it after it has left the nest and embarked upon its career in the outside world. This support may repair its damaged parts or may destroy it and replace it with a replica depending upon the economics of the situation. Modifications may be carried out, but again it may be more cost effective to destroy and replace with an improved product.

The end can come in many ways but it is hoped that it comes after the product has successfully satisfied the needs of both its producers and its users.

SOFTWARE ENGINEERING

Gill Ringland

Software Manager, Inmos Limited, Whitefriars, Lewins Mead, Bristol 1. 1

Abstract

The paper takes the software engineering concepts developed on large projects and shows the extent of their applicability to microprocessor implementations. The topics of estimating and documentation are discussed in some detail.

INTRODUCTION

Computers have been programmed now for over 30 years: to old hands, one of the dismaying features of the microcomputer revolution is that many of the same mistakes that were made first in implementing mainframe systems, then in implementing systems on minicomputers, are being made with microcomputer systems. Does this matter - there are many successful installations running computer systems which perform an adequate job day after day - why can't microcomputer systems be implemented the same way?

The most important reason is that computer systems are used to a large extent by people who have been trained, who have access to expert advice and are in a 'work' environment. The computer is known to be a complex beast with needs for peculiar commands at intervals (eg, // DD *). Microcomputer systems, on the other hand, will be used to run car engines, in telephones and by small businesses like estate agents. They will be used in situations where, not only are no experts handy should there be a malfunction, but the immediate consequences to individuals of component failure are much higher. Microcomputer systems will be far more noticeable if (when) they fail than computer systems. So, programmers and system designers must be prepared to provide proofs of the quality of their work in the same way that major organisations insist on quality assurance for their software. But programmers and designers of micro based systems must go further. They must also consider the whole system design and implementation process with a view to minimising problems to the end user.

1 Current address : European Software Manager
MODCOMP
Molly Millars Lane
Wokingham
Berkshire

It is also true that, in the last few years particularly, there have been significant advances in understanding problems of software development and how to avoid them. The industry has realised that "clever" software is not clever. It has looked at where software effort goes in implementing computer systems and found that development is less that half of the activity: a quarter is devoted to adding features left out of the original design, and ten per cent to fixing bugs, after the system has gone live. Since there are clearly many practical problems in fixing software after, for instance, it has been installed in five million new phones, again the emphasis must be on avoiding bugs in microcomputer systems, in validating the specifications of the system and in constructing the software so that it is robust.

To understand how to actually go about it, consider the stages of a project:

- formulation of requirements and specifications;
- system design;
- build (for software this is programming verification and validation or code and test).

Specification

There are three main approaches to specification. The traditional approach is informal: the system is specified in natural language, often following a standard layout, but with all the ambiguities and misunderstandings present in informal communication. The second can be called 'formatted specification' [Boehm (1)] and consists of a programming language-like description which can be checked for consistency and completeness using a computer system. The available tools have been reviewed by Ramamoorthy (2) but the overhead of running most of these tools is high and they are, therefore, more appropriate for large projects. Work on formal specification languages has reached the stage where they have been used on non-trivial applications such as operating systems. The languages are mathematical and allow for formal proof of the system and verification of its implementation. They have been used so far on military systems and in research labs so far: it looks as though their use will extend more generally in the next few years. Bibliographies are to be found in Wegner (3).

In the discussion on documentation and estimating that follows the use of informal specifications is assumed.

Design

System design is the step which requires the conceptual leap from what is needed (to get across the river), to how (bridge or dugout canoe). Advances in software design techniques have been in two areas: enabling a designer to express the program flow succintly, and to express and manipulate data structures. Techniques for both have been reviewed in Freeman and Wasserman (4). In the subsequent discussion on documentation and estimating, it will be assumed that an adequate system design methodology is used - PDL [Caine and Gordon (5)] and Mascot