Computer architecture and instruction set design*

by P. C. ANAGNOSTOPOULOS, M. J. MICHEL, G. H. SOCKUT, G. M. STABLER, and A. van DAM

Brown University
Providence, Rhode Island

INTRODUCTION

A group of computer scientists and mathematicians at Brown University has been engaged in the study of computer graphics for the past eight years. During the course of these studies a variety of topics has been investigated, in particular, during the last few years, the use of microprogramming for implementing graphics systems.[20,21] In early 1971, Professor Andries van Dam and his associates submitted a threefold research proposal to the National Science Foundation. The problems to be investigated were:

(1). Inter-Connected Processing (ICP-ing) between a central computer and an associated satellite processor, with the goal of a dynamically alterable solution to the "division of labor" problem; program modules would be dynamically linked in either machine as a function of availability and cost of resources and response time;
(2) Programming aids at the source language level for the automatic generation of data structure manipulation subroutines and symbolic debugging of data structure oriented applications programs;
(3) The development and use of the Language for Systems Development (LSD),[22] a high-level systems programming language, for generating the applications *and* systems software for both the central computer and the satellite in such systems:

An interactive graphics system is an excellent paradigm for such investigations since graphics applications.

(1) are typically very large in terms of memory space required;
(2) maintain large data bases, many with intricate (list-processing oriented) data structures;
(3) have processing requirements that change dynamically, varying from very heavy (e.g., structural analyses of a bridge) to very light (e.g., inputting a command); and
(4) require real-time response.

* This work is sponsored in part by the National Science Foundation, grant GJ-28401X, the Office of Naval Research, contract N000-14-67-A-0191-0023, and the Brown University Division of Applied Mathematics.

The Brown University Graphics System (BUGS)[18] was designed as the vehicle for performing this research. Principally, the configuration consists of an IBM S/360-67 running the CP-67/CMS time-sharing system,[10] used by the entire Brown University community, and a satellite display station, as illustrated in Figure 1. This reasonably powerful satellite configuration provides such facilities as program editing and compilation, debugging tools, and most importantly, application processing power and data storage. However, because of the two rather distinct demands placed upon the local processor, that of display generation and general computing, and because these two capabilities could run in parallel, it was further determined that the inclusion of two separate processors in the graphics station would be in order. In particular, the first of these processors would be of a general-purpose nature, while the second would be designed specifically for maintenance and regeneration of the display. Figure 2 illustrates the division of these processing capabilities. Unfortunately, the configuration shown in Figure 2 was far removed in scope from any commercially available equipment, and the purchasing of a general-purpose computer from one manufacturer, a graphics processor from another, and perhaps even a display from a third would prove not only unworkable in terms of compatability, interfacing, and programming, but also unadaptable to the implementation desired. It became apparent that it would be necessary to design the satellite system from the ground up. This could be accomplished by building the hardware at Brown; however, the lack of engineering manpower ruled out this possibility. The one other method that could be employed would be to purchase a pair of user-microprogrammable host computers; a few such computers were available at the time. Microprogrammable computers provide the system designer with the hardware upon which he can base a novel system, presenting him with the opportunity, but also the problem, of writing software from the ground up, and with actually designing and implementing his own target architecture and instruction set.

The problem of computing system architecture has been of major importance since the dawn of computers in the late 1940's. The computer user, however, has had little or nothing to do with this problem; scientists and engineers at the manufacturing companies (or universities) have done all the design in seclusion. Once designed,

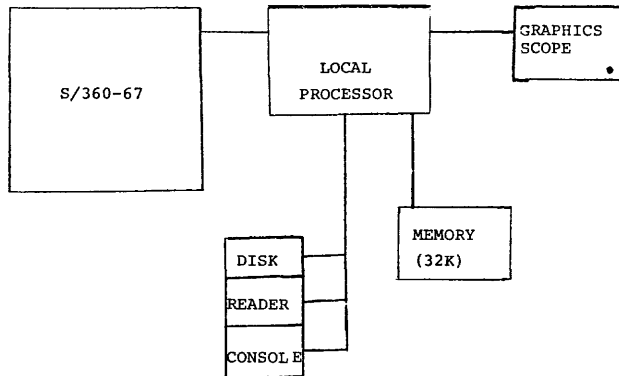

Figure 1

it was then up to salesmen to sell the machine to the unsuspecting public, which accepted it on faith or out of necessity.

Over the last ten years things have begun to change. People have realized that their applications, be they business data processing, process control, or bio-medical research, are distinct and have peculiar computational requirements. The advent of the reasonably cheap minicomputer has allowed users to program their own monitor systems and software packages, oriented toward their specific needs. Regardless, the target architecture of these machines was still fixed and unchangeable, and could not be tailored to a user's specific needs in order to increase effectiveness. However, this latter problem is now being alleviated by the introduction of user-microprogrammable host computers. The purpose of such computers is to allow the user himself to design an appropriate target architecture and instruction set for this particular application, implement this architecture, and perhaps change it after he has learned more about what he needs. A good overview of microprogramming in general is found in Reference 16. Microprogramming trade-offs for user applications are discussed in Reference 5.

It is at this point that a clear distinction between a target architecture and a target instruction set must be made. The architecture defines the basic relationships

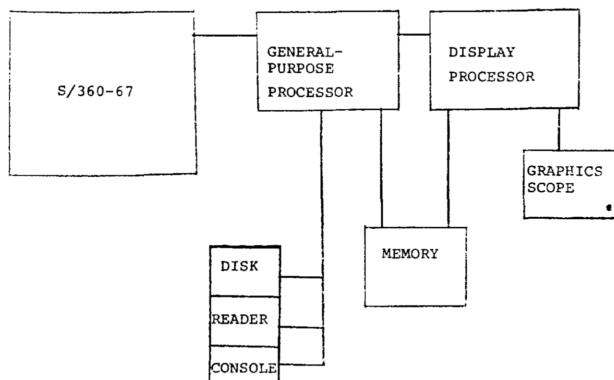

Figure 2

between the various components of the machine, e.g., storage, registers, control, arithmetic units, etc. On the other hand, the instruction set is simply the array of discrete operations which may be utilized by the programmer. A specific example is the comparison of the Burroughs family of stack machines[3] and the IBM S/360 family.[9] The target architectures are entirely different, whereas the instruction sets are similar.

The purpose of this paper is to discuss the problems of machine architecture and instruction set design in general, while referring to the specific BUGS implementation. Based on this discussion, a set of ideas and suggestions is presented to form an initial guide for future implementers of microprogrammed machine architectures.

CHOOSING A MICROPROGRAMMABLE HOST COMPUTER

As stated in the Introduction, much of the rigidity of conventional computers can be overcome if the user is willing to microprogram his own target architecture and instruction set. Although it has been said too many times already, it remains necessary to point out that the aura of complexity surrounding microprogramming is purely a product of scientific mysticism. Microprogramming is not much more than fairly conventional programming at a different level, perhaps requiring greater attention to efficiency;[12] anyone who has coded a simulator or interpreter has already programmed at that level. Microprogramming therefore, being programming at a lower level, transforms the problem of rigidity of the target level architecture into the lower-level problem of host architecture rigidity. After all, how can one design a 24-bit target architecture if he knows it will be implemented on a 16-bit host? And one might as well give up if a decimal machine is desired without decimal hardware in the host. Such conflicting features are not impossible to implement, but they will be extremely inefficient and difficult to microprogram.

At the time the microprogrammable hosts for BUGS[1] were chosen, there were none available that were sufficiently adaptable to allow a wide choice of target architectures. In other words, the rigidity of the host architecture limited the range of target architectures almost entirely to the standard Von Neumann variety. Most users would not consider this limitation a hindrance; they are used to standard architectures and would be at a loss to design an alternate one. However, it is becoming more and more apparent that the barriers to increasing computational effectiveness today are a factor not so much of the crudity of the instruction set as of the unyielding nature of unadaptable hardware. Even the simplest instruction set can simulate a Turing machine and hence compute any function, but the ease with which these functions can be performed depends on the overall blend of machine facilities. Burroughs has begun an attempt at solving the rigidity problem with the introduction of the

B1700 variable-micrologic processor,[23] which takes a first step toward eliminating certain inherently structured components. However, the B1700 cannot as yet be considered an inexpensive user-microprogrammable computer.

All in all, there were four hosts from which to choose, including the Interdata Model 4[11] Microdata 800[13] Digital Scientific META 4,[6] and the Nanodata QM-1.[15] It is immediately apparent that the machines vary widely in architecture. Our consideration was narrowed down very quickly by the fact that the Interdata and Microdata machines have two major deficiencies. The first is the 8-bit microregisters, which would prove horribly inefficient for implementing the 16- or 32-bit arithmetic required for even basic numerical computing. Two or four registers would be required per operand, and multiple-precision arithmetic would have to be performed. The second deficiency is the unavailability of on-site user microprogramming (let alone writeable control storage), making experimentation and redesign virtually impossible. For these two reasons the choice was narrowed down to the META 4 and the QM-1.

The QM-1 had two major features in its favor. The first was the abundance of microregisters and large amount of storage, while the second is that of writeable control storage. However, it appeared that the machine would not be available for many months, whereas the META 4 was immediately deliverable. Furthermore, the impressive control cycle speed of the META 4 was enticing. On these grounds it was decided to purchase two META 4's, the first of which would be the general-purpose processor called the "META 4A"[2] and the second the graphics processor called the "META 4B."[19]

DESIGNING A TARGET ARCHITECTURE

Designing a target architecture should not have to be a major research effort. Unfortunately, however, there is a plethora of considerations which will greatly affect the ultimate usefulness of any design, and yet there are no available guidelines to help cope with them. If these considerations are not dealt with and ultimately synthesized in a reasonable manner, the architecture may fail to be of any use whatsoever.

At first glance, the most important consideration may appear to be the application for which the architecture is intended. Although every application requires certain basic computational capabilities, the strength of a specific architecture lies in its ability to simplify the problems at hand. For example, process control requires a fast and flexible interrupt handling mechanism, whereas information retrieval necessitates powerful data structure manipulation operations. So it might well be concluded that an optimal architecture contains basic arithmetic, logical, and decision-making tools plus facilities oriented toward the ultimate application(s).

The above assumption should be examined at a lower level. Suppose there is a machine with an instruction (call it SEARCH) which scans a linked list for an entry with a specific key. Such an instruction is immensely useful for operating systems with queue-searching requirements, for information retrieval, or for computer graphics. Consider now the level of programming available for the machine. If programmers are coding in assembly language, the instruction can probably be utilized; the determination of when it can be used is up to the programmer. However, if a higher-level language is available, it may be impossible for even a very sophisticated compiler to determine when such an instruction can be generated without an explicit SEARCH primitive, because the fact that the programmer is performing a queue search is hidden in a four or five statement loop. A vast amount of research concerning the relationship between compilers and instruction sets has yet to be conducted.

The SEARCH difficulty is only indicative of a basic contradictory aim in the current design of computers. In most cases, the designers are thinking in terms of assembly language programming, and hence produce an instruction set with an abundance of special-purpose operations that can be used only by a resourceful assembly language programmer. As soon as the compiler designer begins considering the type of code his compiler is to generate, however, these instructions prove useless and perhaps cumbersome.

So where does that leave us? It is at this point that the user must decide how much expertise he has available, how much time he is willing to devote, and how much money he has to spend. If he decides to bend over backwards, then he can purchase something like a B1700, spend time experimenting with the architecture design, and probably produce a fine target machine. Indeed, such a processor is a particularly convenient vehicle for tackling the hardware-firmware-software problem,[14] i.e., the problem of how to distribute the processing function between hardware, microprogram, and software for optimal performance. Many users might complain that they have not the money nor desire to spend excessive time in the design of a system. In this case a somewhat narrower approach to a conventional architecture is in order, with perhaps only a few basic improvements. If the user chooses to go the B1700 route, then the conflict between assembly and high-level languages can be eliminated by maintaining multiple emulators, one for each language. However, most users will probably choose the more conventional direction, in which case the problem still remains, and has a few solutions. If only assembly language will be used, then there is no reason not to include SEARCH. The trend today, though, is toward higher-level languages, particularly with programmers becoming more enlightened to the successes of the structured programming approach.[3,7] The compiler designer can simply ignore the SEARCH instruction, or he can add a SEARCH statement to the language. If he chooses to ignore it, then its inclusion in the instruction set is questionable and must be reconsidered. In this manner, each

special-purpose feature of an architecture must be evaluated separately to determine its ultimate usefulness.

Another major consideration is that of the ultimate speed of the target machine. Most users might immediately say that "the faster it goes, the better I like it." Many applications indeed require such speed, particularly real-time systems. Unfortunately, because of the aforementioned rigidity of host processors, it is impossible for a user to reduce emulation time by putting often-used functions into the hardware. In the work at Brown it has been found that, because of this fact, the speed of the individual functions in a target machine varies directly with their complexity. A good example is that of memory addressing schemes. Simple absolute addressing is extremely fast, while base register-displacement addressing is much slower. Add on an index register and an indirect flag and memory referencing will slow to the speed of cold molasses. The question that must be asked about each of these functions is: How much speed am I willing to trade off for useful complexity? You may come to the conclusion that speed is all important. Then again, programming and compilation ease may be the biggest factor, in which case a more powerful instruction set is desired. It must be kept in mind that the execution speed of a simple, fast architecture and a complex slower one may be equalized by the fact that the slowness of the latter is made up for by its more powerful instructions, i.e., the faster machine requires more instructions to perform the same function. This implies an advantage to choosing the latter architecture, since programs coded on it should be generally smaller than those coded on the other architecture (see the IBM 1130/META 4A benchmark in a later section for a specific example).

One of the most probable misconceptions in evaluating speed requirements is that of determining how much computing per unit time is actually going to be done. If the application is input/output bound, or if it processes human requests, a somewhat slow CPU may go completely unnoticed. Another possibility is that of a multiprocessing system—upon consideration it may be determined that one processor can be slow and more complex even if the other needs to be fast.

An unfortunate problem with most microprogrammable processors today is the very limited amount of control storage which can be included (one to four thousand words in most cases). Once a basic target instruction set is microprogrammed, there may be little space left for application-oriented or experimental facilities. As in all programming, the space/time tradeoff is thus present, requiring the speed and space considerations to be evaluated in parallel.

The above considerations lead directly to a related consideration, that of tuning the target architecture to fit the host computer. For reasons of speeding up the target machine and simplifying the microprogramming task, certain functions that the host machine does poorly should be avoided. One example is a host computer without bit testing facilities; this suggests that a TEST BIT target instruction would be unwieldy. Another example is that of the word size of the target machine—it should optimally be the same as the host machine word size, and at worst a multiple thereof. The speed consideration is by far the most frustrating of all. It may require leaving out many features of the target machine that would otherwise be desirable, simply because they are uselessly slow or impossible to implement. It has also been shown that many functions may run almost as fast in the software as in the firmware,[5] in which case, for the purpose of saving control storage or for making the function more easily changeable, they should not be microprogrammed.

A final consideration is that of the human programming factor. There may well arise a situation in which there are a handful of expert programmers trained on a specific machine, but it is decided for one reason or another to replace the machine with a microprogrammable processor. Certainly, if this new processor were to support a target architecture similar or identical to the original machine, the programmers would be doing useful work much sooner than if they had to be retrained. Furthermore, any existing software packages could be converted in much less time, a factor that may well prove to be the saving or the death of the conversion. Perhaps the new processor is to run as a satellite to a bigger computer. In this case, programmers may be writing assembly language code for both machines. If they had identical instruction sets, then these programmers' sanity could be preserved, whereas if they were somewhat different the programmers may not be able to switch machines with much alacrity. Furthermore, similarity between the two machines would allow compilers to be written which could produce optimized code for both machines using identical algorithms.

The evaluation of all these interrelated considerations can add up to a staggeringly complex problem. Indeed, many decisions cannot be finalized until after the system is implemented and used for a while. If the host computer is equipped with writeable control storage, post-implementation decisions are no problem. However, most hosts available today do not have such control storage, so that the design must be fairly well finalized before it is implemented. The best tool in this case is a good microcode simulator for the host, equipped with timing and debugging features.[24] Using the simulator, small applications programs and system software can be written in the target instruction set and tested. This simulation should turn up not only design and microprogramming errors, but also help determine the usefulness of experimental features and perhaps point out missing features. Uncountable hours of headaches can be saved in this way.

The first design of the BUGS general-purpose processor (META 4A) began with what were thought to be fairly concrete decisions concerning the considerations discussed above:

Application facilities. The intent of the META 4A design was to produce a general-purpose processor which could

support a variety of applications, the most important of which was graphics (keeping in mind that actual display regeneration was to be done with the META 4B). Therefore, complete data structure searching and manipulation operators, plus operators for manipulating arbitrary length character strings were included. In addition, requirements for communication with the IBM S/360-67 necessitated the inclusion of a microprogrammed interface between the META 4A and the IBM multiplexor channel, plus target instructions to control this communication.

Programming languages. This area was of definite concern in choosing the target instruction set. The ultimate goal was to use the LSD language for all programming, but, it would not be available for at least a year. Hence, for the interim, a powerful assembly language was needed, but it was necessary to think ahead and include facilities useful to a compiler. Unfortunately, the knowledge of just what these facilities should be was inadequate due to the fact that the compiler was only partially designed and partially implemented at the time. A limited set of instructions for procedure entry and exit were included, plus the idea of automatic storage was formalized and included in the firmware. Furthermore, it was decided to go ahead and include whatever instructions would be useful for assembly language programmers, and simply let the designers of LSD ignore them if they were of no use.

Speed. Because the actual graphics display regeneration was to be done in the META 4B, it was felt that the speed of the META 4A was not as crucial as its power and flexibility. The overall philosophy was to derive as much speed as possible without deterring from producing a powerful and easy-to-use instruction set.

Emulator size. The size of the emulator was limited to 1500 microinstructions due to available funds. For this reason there was not much choice but to code so as to save control storage space at the expense of speed.

Host considerations. The META 4 host seemed general enough so that any feature could be implemented; as it turned out, this was definitely not the case. Examples of the inadequacies that were discovered are discussed in the following section.

Human factors. This, too, was a distinct problem. Programmers would be working on three separate processors (S/360-67, META 4A, and META 4B) in parallel, and therefore the idea that the local processors should look like S/360's was a strong one. On the other hand, these programmers were also experienced on other processors and felt that working in two different environments simultaneously would not be entirely out of the question. It was decided that architectural similarity was of only secondary importance.

A FIRST ATTEMPT

The first design of the META 4A was begun by considering the great variety of computers already on the market and classifying them into categories. The evaluation of these computers would allow a choice of a base architecture, which could then be improved and customized in light of the considerations outlined previously. The following basic architectures were considered:

IBM 1130-like. This category is considered to be composed of computers with relatively simple architectures that would be easy to implement and run efficiently on the META 4. Such things as simple addressing schemes, short instruction formats, and a small number of instructions would contribute toward this ease and efficiency. On the other hand, programming on such a machine would be slow and tedious, and there were no high-level facilities for use by the compiler designer. Furthermore, the integration of data structure and character manipulation features would be difficult, due to the lack of a sufficient variation of instruction formats and too few operation codes. Experience with IBM 1130's, and the fact that the META 4 host was reasonably powerful, indicated that this was not the way to go.

IBM S/360-like. In this category were computers with more complex architectures, offering the programmer more instructions and more power. Such an architecture may include multiple target registers, general addressing schemes, and a wider range of application facilities, such as character manipulation, that make the programming problem simpler and smaller. However, with this power came complex instructions that require more time to emulate and more control storage to contain the emulator. One advantage to be gained by emulating an instruction set like that of the S/360 was the pre-existence of useful software such as assemblers and linkage editors. In a previous microprogramming project[V2], an S/360-like instruction set had been microprogrammed on an Interdata Model 3 with reasonable success.

DEC PDP-11-like. This category basically included only the PDP-11 family[D3]. It was considered separate and distinct simply because the PDP-11 contained a blend of features not found on other machines, such as stack operations and a highly flexible addressing mechanism. Instructions were generally variable in length, so that only necessary fields need be included—this was felt to be an advantage since the BUGS configuration had only 32K bytes of storage. Some considered the variable formats to be unnecessarily complex and confusing; one prospective user went so far as to state that he would refuse to code for the system if such an architecture were adopted.

Stack machine. The final category was that of a stack architecture. Although such an architecture was ideal for high-level languages, it was difficult to program in assembly language. More importantly, such an architecture requires special hardware to make up for having the stack in core (e.g., the A and B registers on the Burroughs machines,[3] and without this hardware on the host, execution could be intolerably slow.

After evaluating the above architectures, the 1130-like architecture and the stack machine were ruled out for the reasons mentioned. At this point the decision became difficult, but the PDP-11-like architecture seemed to have a better blend of instruction power and size than the S/360-like architecture. The fact that it was perhaps overly complex seemed somewhat irrelevant since those people who were to do the initial assembly language programming were extremely experienced. For these reasons it was decided to go with the PDP-11-like architecture. The implementation of the PDP-11 architecture proceeded smoothly during the summer of 1971, until finally, when the bulk of the microprogramming was complete, timing measurements were made on the resulting emulator. It was discovered that a register/storage ADD instruction, requiring only three storage cycles (approximately three microseconds), took a total of 10 microseconds to execute. This time was considered to be completely inadequate.

In retrospect, the bottleneck became glaringly apparent. The instruction formats that had been adopted for this architecture consisted of many small fields (two or three bits) of information specifying such things as register numbers, addressing modes, and operation codes. These fields had to be isolated into various microregisters in order to fetch the target registers, branch on the operation codes, and to perform other necessary functions. Not enough attention had been paid to the META 4 host to realize that such isolation would require many control cycles since the only shifting that could be performed was right and left shifts of one or eight bits. If a 3-bit field must be isolated from bits 8 - 10 of a 16-bit word, for example, five shifts of "right one" must be performed, requiring about half a microsecond on the META 4. Performing such shifting many times in the course of a target instruction decreased the efficiency of the emulator drastically.

From the failure of this first design attempt came the knowledge that tailoring the target architecture to the host machine is of great importance and cannot be underestimated. As stated, the speed of the META 4A was not the most important factor, thus the slowness could perhaps be justified by arguing that programs would be significantly smaller with the PDP-11-like format than with the other architectures considered. To ascertain the validity of the justification, a set of benchmark programs was written using the PDP-11-like instruction set and a slightly modified S/360 set. Such programs as storage allocation routines, matrix inversion algorithms, and text processing functions indicated that not only did the S/360 instruction set outperform the PDP-11 by a speed factor of two to one, but that the PDP-11 programs were never more than 10 percent smaller than the others.

A SECOND ATTEMPT

Once the PDP-11-like architecture was abandoned, the only remaining possibility indicated by the investigations outlined above was an S/360-like architecture. The previous microprogramming of an S/360 emulator had been done in order to investigate the properties of the META 4 host, and this microprogramming indicated that S/360 instruction formats would be relatively free of complex shifting operation and hence faster to decode as compared to the PDP-11 formats. Indeed, when the second microprogramming task was finished, it was found that an equivalent ADD instruction took only 4.5 microseconds, as compared to the 10 for the PDP-11-like set; this was considered a satisfactory improvement, particularly in light of the small difference in program size.

The final implementation of the META 4A general-purpose processor, although S/360-like in nature, has many major departures from the actual S/360. In terms of architecture it is almost identical, except for the fact that the major numeric data type is the 16-bit (halfword) integer rather than the 32-bit (fullword) integer, due to the fact that the META 4 host has 16-bit registers. The two features omitted were the decimal data type, as this was considered unnecessary, and the floating-point data type. Floating-point is not included for two reasons. The first is that it is extremely difficult to implement in microcode without any hardware assistance; the resulting instructions would be extremely slow and consume a tremendous amount of precious control storage. The second reason is that any large amount of floating-point processing could be performed in the S/360-67 and the results transferred across the multiplexor channel to the META 4A. The META 4A has 16 target registers, implemented using 16 of the META 4 host's 32 registers, and instruction formats identical to those of the S/360. In terms of instruction set, however, it has many improvements over a S/360.

(1) The instruction address register, or Program Counter (PC) as it is called on the META 4A, is actually target register 1. This feature allows more complex branching techniques, such as can be obtained by performing an addition into the PC, or by loading an address from storage into the PC. Although this is a powerful facility, it does add to the inscrutability of the user's program logic. More importantly, as long as all local data is placed beyond any instructions which refer to it, the PC can be used as the program base register, thus freeing another precious general-purpose target register from this function.

(2). If the Pc is used as the program base register, it is impossible for an instruction to perform a backwards reference. This is no problem for data references, but branch instructions must be capable of diverting control to a previous instruction. For this reason, the format of the branching instructions has been changed from including a base-displacement address to including simply a *signed* displacement considered relative to the PC. Not only does this alleviate the backward branch problem, but it makes the decoding of branch instructions much faster, since a base-displacement address, requiring a register number isolation, fetch, and addition,

does not have to be performed. Branch instructions on the META 4A execute faster than those on an S/360-50!

(3) A new instruction format, called Register-Immediate (RI) format, has been added to the instruction repertoire. This format allows the programmer to perform the most common arithmetic and logical instructions using a register and an immediate halfword as the operands, thus saving both a base-displacement calculation and the halfword of storage that would be required for the remote constant. This proves to be a major factor in making most META 4A programs smaller than the equivalent S/360 programs. Additionally, the RI format instructions execute anywhere from one to two microseconds faster than the equivalent register/storage instructions.

(4) Instructions are provided which operate upon arbitrary length character strings. With these instructions the programmer can assign, compare, scan, translate, and initialize character string up to 64K bytes in length.

(5) SEARCH, ENQ, and DEQ instructions are provided for manipulating linked lists and tables. The SEARCH instruction can scan a table or a linked list for an arbitrary length key anywhere in its members which is a logical function of an argument key. If an entry satisfying the function is found, a register is set to point to it. Once a SEARCH is performed on a linked list with DEQ, the satisfying entry can be deleted from the list, or a new entry can be added following it with ENQ. These instructions have proven invaluable time and space savers for implementing queue searches in the BUGS operating system.[17] Such queues as the free storage queue, dispatch queue, and the interrupt exit queue, are searched and maintained by these three instructions. Unfortunately, they will not be generated by the LSD compiler, except perhaps via an explicit primitive.

(6) One interesting architectural experiment which was performed was to include a set of crude stack manipulation instructions, allowing the programmer to set up an arbitrary size stack and then push and pop information into and from it. The point of such an experiment was to learn whether or not programmers who were not experienced with a stack machine could learn to make use of such facilities, e.g., for expression evaluation. To date, no BUGS software has utilized the stack instructions.

(7) Two instructions, ENTER and RETURN, exist in order to simplify, and particularly to speed up, the subprocedure entry and exit protocol. Each procedure has associated with it a Stack Frame area, which contains a register save area and an arbitrary amount of automatic storage. These Stack Frame areas are maintained by the META 4A operating system. Experimentation with the format and content of these areas is continuing today.[17]

(8) Special facilities were included in the firmware to support the addition of Extended instructions. An Extended instruction is an instruction which has an operation code not recognized by the firmware, but which has special meaning to the operating system. When such an instruction is executed, the firmware stores the parsed instruction into a special area in storage, and causes a target-level interrupt. At this point the operating system can simulate any function desired and then return control to the program. Such a facility has proven invaluable for testing a new instruction before it is placed in the firmware, and for use as a communication method between user programs and the operating system software.

In addition to the major points listed above, a great number of miscellaneous instructions have been added to the standard S/360 repertoire in order to fill out what were considered "gaps" in the instruction set. This included such things as storage to storage arithmetic, more address manipulation features, and indirect addressing on certain instructions. Although these instructions are sometimes used and can help decrease the size of a program, they also tend to add to the difficulty of learning and digesting the instruction set. In particular, they increase the number of ways in which a problem can be coded and make it extremely difficult to select the most optimal algorithms. The question which remains, and which is currently being investigated, is just how much it is worth adding facilities which, although they may cut the size of a program by 5 percent, clutter up the instruction set and use up control storage. In addition, although these instructions were added to fill gaps in the S/360 repertoire, they have introduced their own gaps. An example is the addition of address manipulation instructions. Such an addition suggests that perhaps the address should be recognized as a valid data type, just as an integer is, and a full address manipulation instruction subset should be added. The META 4A does not have such a full subset, so that a new gap is introduced. Similarly, many a META 4A programmer has been heard to mutter such things as "if I can add two halfwords in storage, why can't I OR two halfwords?"

Another problem with the addition of "random" instructions is exemplified in an instruction by instruction analysis of the code to be generated by the LSD compiler. A full third of the META 4A instruction set is never utilized by the compiler; once the programming load is shifted over to LSD, these instructions will become virtually useless. The control storage taken up by them could probably be put to a much better use.

The BUGS system has been in standalone production use since August 1972, primarily for research into N-dimensional mathematics. The performance of the META 4A has proven to be rather impressive. Based on

the applications programs written so far, a typical META 4A program is between 3/4 and 2/3 the size of the equivalent S/360 program. Furthermore, the speed of execution of these programs (using halfword operands) lies somewhere in between the speed of an S/360-40 and an S/360-50, which is extremely pleasing considering the cost ratio between the META 4 host (about $30,000) and an S/360.

One interesting benchmark was a comparison of the IBM 1130 emulator supplied with the META 4 host (which is twice as fast as an actual 2.2 microsecond IBM 1130) and the META 4A emulator. A version of the META 4 simulator used for debugging microprograms[1] was written for both of these target machines and compared. The comparisons showed that the simulation speed of these two programs were indistinguishable. However, the META 4A version is 1/3 the size of the 1130 version, due primarily to the lack of character manipulation on the 1130.

In light of the above benchmarks and hand simulations of test algorithms written for the Data General NOVA and DEC PDP-11 series, it appears that arbitrary algorithms typically do not run appreciably slower and do use less storage on the META 4A, while algorithms that take advantage of the special purpose instruction on the META 4A both run considerably faster and use considerably less storage.

CONCLUSION—A RATIONAL APPROACH VIA FORMALIZATION

Although an initial framework around which a user can design and implement his own target machine has been built, there is nothing to prevent this design from being *ad hoc*. Indeed, in light of the many considerations which must be synthesized, an *ad hoc* design is inevitable. It has been shown that an architecture can be both an improvement over previous architectures and reasonably efficient while still being disorganized and incomplete. Such deficiencies cause the assembly language programmer many headaches in terms of choosing algorithms, optimizing code, and generally writing programs. Furthermore, if the choice of target instructions is disorganized, many instructions will be included which are minimally useful to the assembly language programmer and useless to the compiler designer, simply because a far-fetched use of the instruction was envisioned by one of the members of the design team. An interesting example of this is the LXB instruction on the META 4A, which reads a 16-bit halfword from storage and loads it into a register after exchanging the two bytes. This instruction was included for no other reason than the META 4 host machine had a byte swapping facility. It should be obvious that LXB has rarely if ever been used. All in all, the ad hoc architecture is due to a random combination of features useful to the application, features easily adaptable to the host, and features useful to the programmer.

How can such architectures be eliminated? A look at many of the high-level languages in use today, particularly PL/I, will reveal a general philosophy that deals with the problem. In PL/I, there is a well-defined set of data types and a well-defined set of operators, and any combination of these data types and operators is defined, except where meaningless. If such a formalization is adopted, it becomes easier to choose the statements necessary for each individual step of a program, and reduces the obscurity which results when "unnatural" statements are required. Furthermore, it reduces the tendency toward operators and data types which are added for special cases and hence do not fit into the overall language scheme. There is no reason why this same formalization cannot be applied at the lower level of target machine instruction sets. Since the average program is concerned chiefly with data manipulation, as opposed to I/O or interrupt handling, a formalization of the data manipulation facilities of the machine would have major impact.

The most apparent programming benefit of such formalization would be to the compiler designer. In order for him to allow the aforementioned generality to a programming language on most current machines, he must generate unnatural code which performs the operations indicated by the programmer. This code could be eliminated if the operations were made natural by allowing them to be performed directly by single instructions. The Burroughs family of stack machines has adopted just such a philosophy and the results are well known: ALGOL compilers which can generate efficient code at an incredibly high rate. In addition, the assembly language programmer gets an equivalent benefit in that he can code the individual steps of an algorithm in a more straightforward manner, without regard to such irrelevant considerations as whether an operand is in storage or in a register or whether a character string is longer than 256 characters. Programs coded in such an environment should be shorter, more free of errors, and easier to modify in the future.

The programming benefit gained from formalizing a proposed architecture is not the most important one, however. The purpose of this paper has been to outline a set of considerations which the target computer designer must keep in mind when creating a new machine. If the design is approached in a haphazard fashion, the designer will have perhaps 100 or 150 assorted features and instructions about which he must ask such questions as "are they useful to the application?", "how useable will they be by assembly and high-level language programmers?", and "will the host machine support them efficiently?" Such an overwhelming number of questions may be impossible to answer, particularly when the interrelationships between the operations is unclear. By formalizing this machine, the synthesis of these considerations can be made simpler and more productive. The designer need only answer these questions about perhaps twenty operations and five data types, a much smaller task. If these features are proven to be useful and efficient, then the designer can feel assured that the final

implementations of the instruction to perform the operations will be useful and efficient.

Current research at Brown is attempting to deal with the formalization question. An analysis of the use of the current META 4A instruction set is being performed via modifications to the firmware with the hope of determining instruction and instruction sequence characteristics and applying these characteristics to a determination of an optimal instruction set. A formalized architecture will then be designed, experimentally implemented, and an evaluation made of the improvement in program coding ease, speed, size, etc. Once this is done, a more detailed guide to computer design and implementation can be written.

REFERENCES

1. Anagnostopoulos, P. C., *META 4 Simulator Users' Guide*, Brown University, Center for Computer and Information Sciences Technical Report.
2. Anagnostopoulos, P. C., Sockut, G. H., *META 4A Principles of Operation*, Brown University, Center for Computer and Information Sciences Technical Report.
3. *B6500 Reference Manual*, Burroughs Corp., 1969.
4. Baker, F. T., "System Quality through Structured Programming," *AFIPS Conference Proceedings*, Volume 41, Part I, 1972.
5. Clapp, J. A., "The Application of Microprogramming Technology," *ACM SIGMICRO Newsletter*, April 1972.
6. *META 4 Computer System Reference Manual*, Digital Scientific Corporation, May 1971.
7. Dijkstra, E. W., *Notes on Structured Programming*, Technische Hogeschool, Eindhoven, 1969.
8. *PDP-11 Processor Handbook*, Digital Equipment Corp., 1971.
9. *System/360 Principles of Operation*, International Business Machines Corp., 1968.
10. *CP-67/CMS User's Guide*, International Business Machines Corp., 1971.
11. *Model 4 Micro-instruction Reference Manual*, Interdata Corp., 1968.
12. Kleir and Ramanoorthy, "Optimization Techniques for Microprograms," *IEEE Transactions on Computers*, July 1971.
13. *Mircroprogramming Handbook*, Microdata Corp., 1971.
14. Mandell, R. L., "Hardware/software trade-offs - Reasons and directions," *AFIPS Conference Proceedings*, Volume 41, Part I, 1972.
15. *QM-1 Hardware Level User's Manual*, Nanodata Corp., June 1972.
16. Rosin, R. F., "Contemporary Concepts of Microprogramming and Emulation," *ACM Computing Surveys*, Volume I, December 1969.
17. Stockenberg, J. E., Anagnostopoulos, P. C., Johnson, R. E., Munck, R. G., Stabler, G. M., van Dam, A., "Operating System Design Considerations for Microprogrammed Mini-computer Satellite Systems," *Proceedings of the National Computer Conference and Exposition*, June 1973.
18. Stabler, G. M., *Brown University, Graphics System Overview*, Brown University Center for Computer and Information Sciences Technical Report.
19. Stabler, G. M., *META 4B Principles of Operation*, Brown University, Center for Computer and Information Sciences Technical Report.
20. van Dam, A., "Microprogramming for Computer Graphics," ACM *SIGMICRO Newsletter*, April 1972.
21. van Dam, A., Schiller, W. L., Abraham, R. L., Fox, R. M., "A Microprogrammed Intelligent Graphics Terminal," *IEEE Transactions on Computers*, July 1971.
22. van Dam, A., Bergeron, D., Gannon, J., Shecter, D., Tompa, F., "Systems Programming Language," *Advances in Computer*, Volume 12, Academic Press, Oct. 1972.
23. Wilner, W. T., "Design of the Burroughs B1700," *AFIPS Conference Proceedings*, Volume 41, Part I, December 1972.
24. Young, S., "A Microprogram Simulator," *ACM SIGMICRO Newsletter*, October 1971.