DESIGN OF A USER-MICROPROGRAMMABLE BUILDING BLOCK

Michael Kraley, Randall Rettberg, Philip Herman,
Robert Bressler, and Anthony Lake

Computer Systems Division
Bolt Beranek and Newman Inc.
Cambridge, MA 02238

Abstract

A user-microprogrammable computer has been developed for use as a building block in general-purpose and dedicated computer systems. The architecture is designed to be easily microprogrammed and features a 32-bit, vertically oriented microinstruction. The processor has a 135-nanosecond cycle time, either 16- or 20-bit macro data paths, and 1024 hardware registers. A significant fraction of the processor bandwidth may be budgeted for I/O processing to allow the substitution of microcode for expensive peripheral controllers. Furthermore, the micromachine is well suited to the emulation of other computer architectures in that it provides a large writable microcode memory and a minimum of special processor data paths. The design goals and strategies which determined the machine architecture are discussed, as well as an overview of the architecture and hardware organization. Finally, we report a number of specific applications developed to date.

1 Introduction

Recognizing the need for a flexible, inexpensive, high performance computer for use in Bolt Beranek and Newman's systems development, our group set out in 1978 to build a general-purpose Microprogrammable Building Block (MBB) computer system. The initial applications included replacement of a technologically obsolete minicomputer to preserve an investment in real time communications software, and development of an intermediate language machine supporting a higher level programming language for general-purpose timesharing and dedicated systems. Although these two initial applications influenced certain design choices, we nonetheless sought to define a highly flexible hardware architecture which could be easily tailored for specific applications.

One of the key decisions in the design of our computer was that the underlying instruction set processor would be user microprogrammable. The choice of a microprogrammed architecture follows from the need for a general-purpose, flexible, high performance machine. Programmed architectures are capable of supporting a wide variety of applications, and allow multiple solutions to many system design problems. Also, since we would be doing most of the initial microprogramming in-house, it was felt that taking the time to design a clean, easily microprogrammed machine would later produce dividends of lower costs in application design, development, and maintenance. Providing both microprogramming support software tools and a large, dynamically alterable control store, the system is classified as a dynamically user-microprogrammable computer, after the terminology in [1]. See this tutorial for further background and material on the current state of microprogramming.

Although microprogramming has been used to implement the control structure of digital computers since M. V. Wilkes originated the concept in 1951 [2], user-microprogrammable computers are still the exception in the marketplace. There are many reasons for this lack of enthusiasm: the extra costs associated with providing both software and firmware development tools, the need to reload microprogram memory on power up (which requires an external load device), and the higher costs and lower densities of random access memories and the extra complexity introduced by an additional level of control. All of these reasons encouraged ROM-based microcode. There are some exceptions (see [3] for summaries of currently available microprogrammable computers), but most microprogrammed machines today have the bulk of their microcode fixed in ROM.

Those computers which do have a writable control store often have architectures oriented towards specific macromachines. Therefore, the hardware/firmware/software tradeoffs are usually optimized in favor of one specific architecture. We sought a more general machine than any one macroarchitecture would allow.

Other groups have designed general-purpose microprogrammable machines, for example, the Xerox ALTO [4], Nanodata QM-1 [5], and Burroughs B1700 [6]. However, the Xerox ALTO is not commercially available for use in systems integration; the Nanodata QM-1 minimum configuration necessary to support our applications is an order of magnitude bigger and more expensive than our system price; and the Burroughs B1700 is both too expensive and too complex for our applications. Finally, the control of technology and manufacturing has become increasingly important in providing state-of-the-art systems. For these reasons we set out to develop our own microprogrammable building block architecture.

The following sections provide an overview of the MBB design goals and the resulting architecture, and a discussion of some application

0194-1895/80/0000-0106$00.75© 1980 IEEE

systems developed to date. Currently, the two major applications are a minicomputer emulation system called the C/30, and a high level language oriented system referred to as the C/70. Various dedicated application systems which are completely microcoded are also under development.

2 Design Goals

The most important design goal of the MBB was that it be user-microprogrammable, as discussed in the previous section.

Providing the convenience and accessibility of current minicomputers was another important design goal, which was achieved by careful design of the microinstruction and support tools. Common disadvantages of minicomputers are limited main memory and processor registers [7, 8]. These problems are minimized by use of 20-bit macroinstruction data paths (allowing 1 Megaword of address space) and 1024 hardware processor registers. Both of these features are discussed in more detail in the next section.

A related goal was the efficient support of modern higher level languages. The rapid context switching and subroutine calling allowed by proper management of a large processor register set is an important facet of the language support, as is the efficient emulation of intermediate languages provided by a fast microprogrammed processor. The support of advanced memory systems, such as virtual memory capabilities, is also important in modern programming systems.

The high cost of developing and maintaining complex I/O hardware led to the minimization of peripheral interface hardware as a general design goal. The microprogramming of complex I/O transactions is analogous to microcode emulation of higher level instruction sets, and reaps the same benefits of simple memory-dominated hardware and flexible implementation. This goal is also realized by use of processor-controlled I/O bus with no DMA capability. The I/O bus design runs somewhat counter to the current trend towards distributed control and intelligent peripheral controllers, and represents our desire to build a simple, flexible building block oriented around one powerful computing engine.

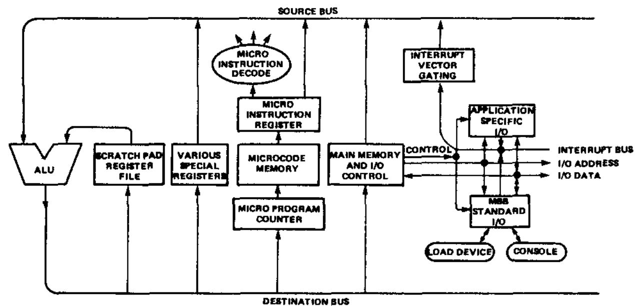
A final design goal was that the machine be easily manufactured, tested, and maintained in the field. Accordingly, only readily available, multiply sourced components, such as Schottky ALUs and 16K dynamic RAMs, were considered in the design. The physical implementation is oriented towards two large (18" x 14") boards containing all of the control processor and memory units, thereby minimizing costly inter-board connections and allowing spares to be easily stocked in the field.

3 Overview of Architecture

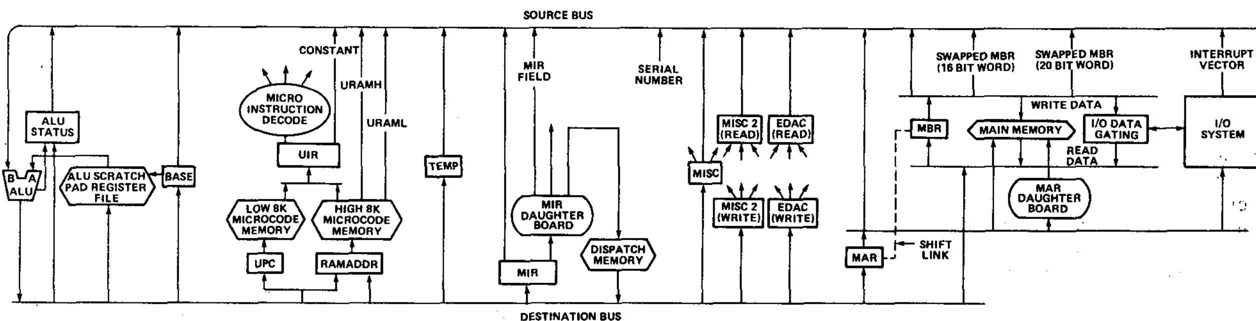
This section provides an overview of the micromachine architecture, with emphasis on the unique features of the hardware design. A more complete specification may be found in the MBB Microprogrammer's Handbook [9].

3.1 Processor Design

Figure 1 shows the logical structure of the MBB processor data paths. The arithmetic logic unit (ALU) is the heart of the machine. Forming a loop with the ALU are the source and destination buses and various internal registers. This loop is referred to as the "processor" and is shown in more detail in Figure 2.



MBB Logical Structure
Figure 1



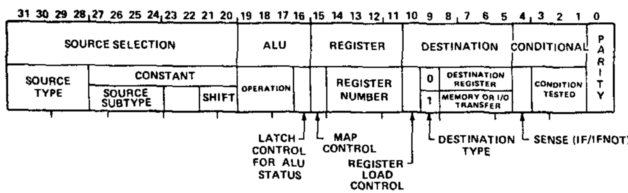
Detailed Block Diagram of the Processor
Figure 2

Internal processor registers include the general purpose scratch pad register file and dedicated system status and control registers. The dedicated processor registers provide the firmware interface to the I/O and memory subsystems. They allow monitoring, and control of processor resources such as the Macroinstruction Register (MIR), Microprogram Counter (UPC), and ALU status. This technique of narrow microinstructions manipulating hardware resources via control registers is known as residual control, and results in significant savings of microcode storage versus specifying hardware control simultaneously in each microinstruction.

In a typical microinstruction, executed every 135 nanoseconds, a "source" is selected to gate its contents on the source bus. This is combined with the selected word from the register file in the ALU, where the desired operation is performed. The result is then routed to a selected "destination" and/or to the register file. This entire operation may be conditionally executed depending on selected processor status (e.g. zero, negative, interrupt pending).

In parallel with these activities, the next microinstruction is being fetched from the address contained in the UPC. Since the UPC is also a destination, the full computational facilities of the machine can be used to modify the microprogram sequence.

The microinstructions necessary to control this machine are quite simple. Since all microcycles are similar, there is only one format of instruction (see Figure 3). The 32-bit word contains fields for the source, register, ALU operation, destination, and conditional execution tests.



Microinstruction Format
Figure 3

This simple instruction format is an important tactic in our strategy of making it easy to program the MBB. Freed of the need to deal with wide microinstructions filled with bits that trigger obscure hardware functions or to manage many simultaneous parallel operations, the MBB programmer does not have the "fear of microprogramming" that is so prevalent in the industry today.

We now discuss some of the more innovative and unique features of the MBB's architecture.

3.1.1 Word Width

Most of the machines that we wished to emulate were traditional 16-bit minicomputers. We wanted, however, to avoid the "small address problem" [8]. A 16-bit word can, at most, directly address a 64K word memory. We continually find that this is not a large enough address space for many program tasks. Various tricks have been used to overcome this problem, e.g. overlays, bank bits, mapping, 2-word addresses, but none are really efficient or clean. The only real solution is to have a word size large enough to hold a complete address. One obvious solution for the MBB was to make it a 32-bit machine, but this would double much of the hardware in the machine.

Instead we decided on a word width of 20 bits. Now the MBB can directly address one million words, which though not large enough for every conceivable program, seemed ample for the kinds of tasks we felt appropriate for the MBB. At the same time, 20 is not that much larger than 16 and we would not be wasting large amounts of hardware when directly emulating a 16-bit machine.

Thus, all data paths, registers, and the ALU are 20 bits wide. When emulating a 16-bit machine, all 20 bits of data are computed and stored, but the high 4 bits can be ignored. A program-settable flag determines whether shifting and ALU status latching are performed on a 16- or 20-bit basis.

With this strategy, we have the opportunity to incrementally expand the capability of a 16-bit processor to increase the address space without doing great violence to the instruction format, thus preserving software compatibility. The expansion might take the form of adding some bits to the address field, or perhaps just expanding the size of an indirect address.

3.1.2 Register File

The MBB has 1024 scratch pad registers, each 20 bits wide. The large number of registers facilitates context switching, since different routines may use different registers, thus internally preserving register contents. In the C/70 timesharing application the microcode actually maintains a cache of macroprocess register sets in the scratchpad, to speed up process context switches. The large register file is also particularly useful in emulating I/O, where each of many I/O devices can have its own block of registers, allowing service of a device with minimum overhead.

3.1.3 Flexible Emulation

Typical microarchitectures that are designed to emulate a specific instruction set often include many special data paths that optimize the machine for this purpose. An important goal of the MBB design was to be able to emulate many different kinds of macromachines. Thus, our design stresses a simple, fast micromachine that has no such

special paths and relies on microcode to support the various macro features. There are some common areas of macro emulation, however, that could consume a large amount of micro machine bandwidth. By providing some minimal hardware support in these areas, the performance of the machine is greatly increased.

A good example is the selection of registers from the macroinstruction. In emulating a multiple register macroarchitecture, the microcode must select the proper register based on the contents of the macroinstruction. Masking and shifting consumes many microcycles. A special path from the MIR to the register selector seems well worth the performance improvement. But these register fields are in different places for different macroarchitectures. Our solution is to provide a connector on the processor board, on which is mounted an application-tailored board, which performs a few basic functions. One of these is to supply the appropriate bits of the MIR to the register selector.

This technique of using a small amount of target-machine-specific hardware to optimize performance is employed in two areas of the MBB: macroinstruction decoding and main memory addressing. These two "connectors" (actually small daughterboards with a minimum of logic) are specific to each application, and plug into the basic MBB.

The macroinstruction daughterboard (which also includes the register selection function just described) computes a perfect hash of the MIR. This is used to address a 1K x 12 "dispatch" memory that holds addresses for the appropriate microcode. Thus, in a single microinstruction, we can perform an arbitrary multi-way dispatch based on the macroinstruction. Several "types" of dispatch can be specified, so that, for example, we could have one for addressing mode and another for op code. As compared with the alternative of several consecutive "test and branch" operations, the ability to quickly branch broadly allows us to minimize the number of microinstructions used. Other commonly used multi-way branch techniques constrain the number and location of the branch destinations, causing non-intuitive code, complexity in the microassembler, and wasted microcode space. By using this dispatch lookup technique, we avoid all these problems. Dispatch memory may also be used to do "fast lookups" of macroinstruction parameters. Dispatch memory is, of course, program loadable and can be thought of as another piece of the microcode itself.

The other target-machine-specific connector sits logically in the path between the Memory Address Register (MAR) and the memory itself. An application MAR decoder can thus implement appropriate functions for: (1) address translation, (2) protection, and (3) memory management. For example, in a demand-paged, virtual-memory-oriented macroarchitecture, maintenance of a bit indicating whether the memory page had been modified is easy to perform on the MAR decoder, and costly to compute in firmware. As in the MIR connector, this is a case where a small amount of simple hardware can greatly assist the microcode in specific macromachine emulation.

An additional use of the MIR and MAR connectors is in testing MBB boards. Since the two connectors contain or affect important data and control paths, certain functions can be verified or diagnosed in a non-invasive manner.

3.2 Memory Subsystems

The memory hierarchy directly reflects the multiple layers of control in the MBB system architecture. At the micromachine level, the 1K register file and up to 16K of microcode memory are used for temporary data and state storage and microprogram storage, respectively. General-purpose retention of macrocode and data is provided by up to 1 Megaword of comparatively slow main (macro) memory. Ample disk storage and a rich ensemble of communication network interfaces are available for higher level distributed computation and storage tasks. For efficient emulation, the MBB memory hierarchy also includes macroinstruction dispatch memory and provision for macromemory address mapping hardware, as mentioned previously.

Microcode memory is 32 bits wide, including one parity bit, and has a 55 nanosecond access time. The memory is divided into programmable read-only memory (PROM) and microcode random access memory (URAM). The current system includes a small PROM-based bootstrap that is able to load microcode and all other memories via a cassette interface or a communications network. Disk-based microcode loading is under development.

The dispatch memory is 1K x 12 bits, and has a 30 nanosecond access time. The 12-bit width permits the macroinstruction emulation code to dispatch to any address within a 4K block of microcode memory.

Dynamic 16K RAMs are currently used for main memory, requiring refreshing (performed by reading) every 2 milliseconds. We chose dynamic memory because it provides significantly higher density, lower cost, and lower power consumption than available static memory. The dynamic RAMs, however, are relatively slow, having a 350 nanosecond access time. Rather than using expensive specialized hardware for refreshing, we have the runtime system firmware assume the memory refresh responsibility. This refreshing, driven by a 100 microsecond system clock, takes approximately 3% of the machine's processing power, and is an example of hardware being replaced by firmware. Upgrading to 64K RAMs will be accomplished by changing jumpers.

In addition to the 20 data bits in each main memory word, there are 6 additional bits used by internal hardware for data error detection and correction (EDAC). Single bit errors are automatically corrected by the hardware and EDAC status is retained for diagnostic purposes. Double errors are uncorrectable; machine operation is stopped and a reload and restart is triggered. In addition, the complete EDAC logic can be placed under program control for full testing.

A disk interface controller compatible with CDC's Storage Module Device (SMD) disk drives is available for the MBB. The interface provides data synchronization, FIFO buffered transfers, and error detection and correction for two disk drives; more drives can be supported with multiple interfaces. (Some of the issues in the design of the disk controller are discussed in section 3.4.3 below.) SMDs cover a variety of devices, ranging in capacity from 20 to 600 megabytes per drive, using both fixed and removable media.

A cassette interface is used in applications where no disk or communications network interface is available for reloading main and microcode memory.

3.3 Microcode Organization

As stated in the last subsection, microcode is contained in both PROM and URAM. PROM is read-only and non-volatile. URAM is writable, and its contents are lost on power failure. PROM code is intended as a system bootstrap; code size is kept small, but code speed is not an important issue. For URAM code, in contrast, size is less important and speed is more important, since this directly affects the efficiency with which a macromachine can be emulated.

The PROM code performs the basic functions of loading, refreshing main memory, and running a basic debugger (DDT). When started at PROM address 0, the machine, after initialization, runs code to perform these functions simultaneously. The loader routine can load all the writable memories: URAM microcode, main memory, dispatch memory, and the ALU scratch registers. Refreshing of main memory is performed to preserve its contents, as noted in the previous section.

The basic PROM-resident DDT provides the functions that would otherwise be performed by a front panel. It allows an operator to examine and change contents of URAM microcode memory, I/O device registers, dispatch memory, main memory, and the ALU scratch registers. The operator can also start and stop both macro and microprograms. This DDT approach allows complete control of the machine via any standard ASCII terminal, saving the cost of dedicated front panel hardware. Terminals may be shared between machines, servicemen can carry pocket terminals, or a dial-up link may be used. In a communications application, the MBB may even be controlled over the operational network data links.

URAM contains the microcode for application functions. These functions consist of emulating macroinstructions and I/O functions (including interrupts). Because the URAM microcode, dispatch, and main memories are writable, new versions of application microcode may be installed easily. This feature greatly aids debugging and field upgrading of application microcode. New macrocode may be installed simply by loading the appropriate part of main memory.

3.4 Input/Output and Interrupts

In this section, we discuss the issues involved in the design of the MBB I/O system. The major topics are the structure of the I/O system and the microinterrupt system used to allocate the microprocessor bandwidth. The I/O system of the MBB is designed to capitalize on the microcoded structure of the machine. While most current minicomputers separate the I/O functions from the processing functions, the MBB integrates them through the microcode. Instead of designing dedicated interfaces around a complex, multiple-master, public I/O bus, we have chosen to nominally budget half of the processor bandwidth to I/O processing in return for simpler interfaces and a centrally controlled I/O bus. In addition, the design of the main memory bus is simplified since the hardware does not arbitrate among different bus masters.

Modular interfaces are now available to support terminal or modem multiplexing in groups of 32 ports, an ARPANET-compatible Local/Distant Host interface, HDLC/ADCCP bit-oriented interfaces, and a SMD disk controller. Currently under development are a high speed local network interface, a digital satellite communications interface, an interface to magnetic tape drives, and floating point hardware.

3.4.1 Hardware/Software Tradeoffs

A key goal of the MBB design is to minimize the hardware devoted to I/O interfaces. Emphasizing use of available LSI interfacing technology, MBB I/O hardware performs only very basic tasks such as handshaking, electrical level conversion, serial-parallel conversion, and the request of a microinterrupt after transferring a unit of data. The microcode then has responsibility for such functions as maintaining the finite state machine for the line, transferring data to or from memory (replacing DMA functions), status reporting, word assembly/disassembly, padding, and checksumming. Such microcode is considerably cheaper, easier to develop, and easier to support than the corresponding hardware.

I/O emulation is not always a trivial demand on the micromachine, however. In one of our first applications, that of the C/30 communications processor, half of the processor bandwidth is nominally budgeted to support the I/O system, leaving the other half for instruction emulation. This bandwidth trade-off is, of course, dynamically allocated based on the instantaneous I/O processing requirements of the machine.

3.4.2 The Microinterrupt System

The I/O service request system of the MBB is a hybrid of polling mechanisms, vectored interrupt systems, and the Pluribus Pseudo-Interrupt Device [10, 11]. The interrupts do not, in fact, "interrupt" the processor; rather, the microcode must periodically poll in order to service any pending interrupt. Hardware is provided to

synchronize interrupt service requests, to perform priority ordering, and to generate an interrupt vector address. It is the microprogrammer's responsibility to access this hardware often enough to meet latency requirements. If no interrupt request is pending, the interrupt vector directs the MBB to the start of emulation of the next macroinstruction, which is the lowest priority service request.

Interrupts, such as those for device transfer completions, are queued by the interrupt system as service requests. Such requests are serviced only when polled by the microprogram, as follows. The hardware supplies a "vector address" corresponding to the highest priority pending request -- as a readable register in the micromachine. Transferring control to that address services the request. If no interrupt request is pending, the processor hardware supplies an address "MAIN" corresponding to the microprogram's main task. For the C/30 communications processor, as well as for other emulation applications, this task is to emulate macroinstructions: transferring control to "MAIN" would cause the next macroinstruction to be emulated. To dismiss and transfer control to the handler for the highest priority pending request (or to "MAIN" if no requests are pending), the microcode loads the hardware-supplied vector address into the UPC. Interrupt vectors point to locations in writable microcode memory so that service routines may be changed easily.

Microcode routines are thus free from being interrupted. Code is divided into logical segments called strips, each of which dismisses by loading the interrupt vector into the UPC. Strips must be short enough to provide a timely interrupt response; the most severe constraints are imposed by high speed communications interfaces. For each communications interface operating at 64 Kbps, response to a byte of received data must be within 12.5 microseconds, or 100 microinstructions. Although the microprogrammer must always be aware of this limitation, particularly when calling system routines, so far we have found that the use of strips has imposed a welcome structuring on the microcode, and that most strips are naturally short enough to meet these latency constraints.

This approach has several advantages over real interrupts. First, the hardware is simpler and overhead is lessened since it is not necessary to automatically save and restore the UPC and other micromachine contexts. Second, it eliminates the mechanism of enabling and disabling microinterrupts. This would require at least one extra bit in the microinstruction since it would be too expensive to execute many interrupt enable and disable microinstructions. Finally, this approach has considerably simplified the design of the microcode, since interrupts are more readily controlled.

3.4.3 Example Interface

In the case of the disk interface, the hardware provides the functions of synchronization, level conversion, data buffering, error control, and operation-complete interrupts. The firmware has the complex tasks which are typically performed in hardware.

The allocation of simple functions to hardware and complex functions to firmware results in an IC parts reduction on the order of 4:1 compared to hardwired disk controllers. Current microprocessor-based intelligent controllers are still twice as large as the MBB controller. The load on the micro machine is only 4% of the total processor bandwidth in typical timesharing operation, as the disk firmware is only active when a transfer is in progress. The three firmware tasks which control the disk hardware (called SEEK, SECTOR and TRANSFER) may dismiss to allow further instruction emulation while the hardware waits for the operation to complete. The multiple tasks run concurrently, with the SEEK task positioning the arm over the right cylinder, the SECTOR task keeping track of the SECTOR count within a cylinder, and the TRANSFER task actually moving data to and from memory.

3.5 Example Microcode

The following is a stylized example of a portion of the microcode necessary to emulate a simple macromachine. This example is included to demonstrate the techniques of mirroring hardware registers of the emulated macromachine in the scratchpad registers of the MBB and the use of dispatch memory in decoding different aspects of the macroinstruction.

The architecture being emulated in this example is that of a single accumulator machine. Microcode routines to handle two different addressing modes (direct and indirect) and two simple macroinstructions (load and jump) are shown.

The scratchpad registers "R0.PC" and "R1.ACCUMULATOR" are used to mirror the macromachine's program counter and accumulator, respectively. The microcode at "MAIN" expects that the next macroinstruction to be emulated will have been fetched from main memory, and is waiting in the Memory Buffer Register (MBR). The instruction is first placed in the Macro Instruction Register (MIR), where it can conveniently be decoded. The first decoding results in a branch to either DIRECT or INDIRECT depending on the macroinstruction's address mode.

If direct, the address is in the instruction itself, which the source MIRFIELD extracts. The operand is fetched and a dispatch depending on opcode is performed. If indirect, we use MIRFIELD to fetch the indirect address word, wait for main memory, and then use that to fetch the datum itself. We can then perform the opcode dispatch.

The opcode dispatch results in either LDA or JUMP. In each case we perform the action appropriate to that instruction. Each emulation is concluded by prefetching the next instruction and dismisses with the instruction "INTS -> MAIN", which transfers control to the service routine for the highest priority pending interrupt. If there are no interrupts pending, control passes to MAIN, where the next macroinstruction is emulated.

```
;the main emulation loop
MAIN:     MBR -> MIR
          DISPATCH(ADDRESSMODE) -> UPC

;handle direct addressing mode
DIRECT:   MIRFIELD -> MAR(READ)
          DISPATCH(OPCODE) -> UPC

;handle indirect addressing mode
INDIRECT: MIRFIELD -> MAR(READ)
          NOP
          MBR -> MAR(READ)
          DISPATCH(OPCODE) -> UPC

;emulate "load the accumulator"
LDA:      MBR -> R1.ACCUMULATOR
          R0.P1 + 1 -> R0.PC, MAR(READ)
          INTS -> UPC

;emulate "jump"
JUMP:     MAR -> R0.PC, MAR(READ)
          INTS -> UPC
```

3.6 Tools

A hardware system is no better than the tools available to help develop and use it. MBB microcode development is done with the assistance of programs which run on a time-sharing system. A microcode assembler eases the task of creating microcode. The code can then be debugged on a simulation of the hardware. Work on the actual hardware itself is mediated by a debugger program, running on the assisting machine. This debugger program converts the symbolic language of the programmer to the primitive commands of the PROM DDT, allowing the user to load, run, and debug programs with "big machine" facilities.

4 Applications

4.1 Emulation of Obsolete Hardware

In this section we outline the development and current status of the C/30 computer system, which is an MBB tailored to replace a technologically obsolete minicomputer in a data communications network environment.

Since the late 1960s BBN has been involved in research on and development of computer communications networks. In particular, under the sponsorship of the Defense Advanced Research Projects Agency (DARPA) and other agencies, we have invested dozens of man-years in the development of packet-switching network software for state-of-the-art digital communication networks. Systems such as the ARPANET Interface Message Processor (IMP) [12], and compatible Terminal Interface Message Processor (TIP) [13], have been developed for use in high speed terrestrial networks. Satellite-based demand-assignment multiple access networks nodes have also been built for DARPA's SATNET system [14, 15].

A large body of this proven software is written for a 16-bit minicomputer, which was designed using technology available in the late 1960s. Accordingly, one of the first applications of the MBB has been to emulate this minicomputer in order to preserve the enormous investment in communications software.

As the C/30 must both replace and coexist with the original computers on networks such as the ARPANET, careful attention was paid to physical and functional compatibility, performance, and operator interfacing. Microcode was written to emulate the instruction set, I/O system, interrupt system, and Direct Memory Channels (DMC). In writing this emulation, we slavishly followed the model of the existing machine, even to the extent of recreating several bugs that existed in the original implementation. Each time we tried to fix one of these, we found that generations of programmers had accepted the existence of these bugs and written their programs to adapt to their presence. "Fixing" the bug only made existing code not work.

In the end, we achieved our goal of directly emulating the original processor. This was demonstrated by loading and running an unmodified binary version of the existing macroprograms on a C/30. The C/30 has an overall performance somewhat better than that of the original IMP at a third to a quarter of the price, half the power consumption, and one-fifth of the physical size.

As of this writing (June 80) pre-production prototype C/30s have been successfully installed on ARPANET-like networks, running the same IMP application software as existing mainframes.

4.2 A High Level Language Machine

In this application, rather than implementing a system based on an existing minicomputer's assembly language, we developed a new system oriented towards efficient execution of a high level language. The C/70 is an MBB provided with firmware to execute an intermediate instruction set developed by BBN, and oriented towards the C programming language. The C language was developed at Bell Labs [16] and has been found to be a flexible and well-structured language system with a large body of existing software, programmer expertise, and development support tools. A further reason for this choice is the existence of the UNIX operating system [17], an increasingly popular timesharing system. Since almost all of UNIX is written in C, it has proved to be a straightforward job to bring up UNIX on the C/70.

The C/70 intermediate instruction set has the interesting characteristic of combining conventional operations (load, store, add) with fairly unconventional addressing modes. There are no fewer than 19 distinct addressing modes which may be used with any one of 40 instructions. Many of these addressing modes are provided for use in referencing local variables, arrays, and structures, thereby giving the compiler more efficient mechanisms for handling these data items. In addition, another 44 instructions are provided in cases where the full addressing generality is not required.

Of particular interest is the subroutine and function calling mechanism. Since the MBB has 1024 hardware registers available to the microcode, register saving is done by simply designating a new set of registers for the macroprogram to use. This procedure dramatically reduces register save and restore time. Of course, should the macroprogram require a subroutine depth greater than may be accommodated by the register file, then registers are cached in main memory. Furthermore, this register saving and restoring mechanism has been "bundled" into a CALL instruction, thereby reducing execution time and memory references. The end result is that C/70 macrocode executes only 4 macroinstructions to perform a subroutine call and return, including stack adjustments for local variables and arguments; the DEC PDP-11 requires 16 instructions to perform this same operation. This result is important due to the statistically large number of subroutine calls and returns found in C programs.

The CALL instruction does not differentiate between microcode and macrocode subroutines; the only difference is in the address specified. Therefore, a frequently used function may be moved from macrocode to microcode for efficiency, without the knowledge of the calling programs. In one particular case, a speed improvement of a factor of ten was obtained for a critical inner loop by writing about 20 microinstructions.

The C/70 emulation hardware is more complex than the corresponding C/30 application hardware, owing to the increased functionality of the C/70 macroinstruction set and memory system. The instruction decoder daughterboard assists the emulation microcode by: (1) extracting the macroregister specifications from three different possible locations in the 20-bit macroinstruction, (2) providing multiple instruction dispatch vectors for address and operand calculation, and (3) providing three possible transformations of the MIR for use in constant encoding, branch calculations and other program transfers.

The C/70 macroaddress application logic daughterboard provides a fully segmented memory management unit with associated access control, fault handling logic, and segment status flags. The unit provides storage for 1024 segment descriptors, divided into 8 process areas of 128 descriptors each. Segments range from 512 words to 4096 words in increments of 512 words, so that the virtual address space of each process is 512 Kwords. Segment status flags include a "modified" and "used" bit maintained by the hardware, which assists the operating system in context switching and gathering memory statistics. Access control is also provided on a per-segment basis.

The disk controller and macroaddress logic is provided when the C/70 is running an enhanced version of the UNIX operating system. In those cases where disk storage and memory management capabilities are not required, such as with dedicated process controllers or communications switches, a much smaller runtime system, called CMOS (for C-oriented Micro Operating System), is used. CMOS provides the basic I/O and interprocess communication primitives necessary in dedicated real-time application systems, and requires only the basic MBB processor and memory to be present. Both the UNIX and CMOS versions rely on the C/70 intermediate language for their macromachine instruction set.

The technique of creating an intermediate language tailored both to the high-level language it is compiled from and to the microarchitecture that implements it appears to have been a good choice. We can take advantage of the programming ease of the high-level language without fear of inefficiency. The combination of this technique with strategic choice of routines to microcode allows the performance of a C/70 running UNIX to be comparable to that of machines selling for two to three times as much.

4.3 Dedicated Applications

The above applications are oriented around specific macroinstruction sets. However, in some cases we directly microcode dedicated applications to gain throughput and efficiency. For example, a large body of software exists at BBN for a terminal-oriented system, called the Pluribus TIP [18], which is based on the Pluribus multiprocessor technology. To upgrade the terminal-handling capabilities of the Pluribus TIP, we can replace the old terminal-multiplexing hardware with MBBs, with each MBB supporting up to 64 communications lines. The terminal-handling program is microcoded, with main memory used only for data buffering. By replacing complex multiplexing hardware with a microprogrammed machine, we expect both higher data rates and lower maintenance costs to be realized.

Summary

We have described the design goals and resulting architecture of a high performance, inexpensive microprogrammable building block computer. The significant aspects of its design are: (1) simple general-purpose architecture with few special data paths; (2) large writable microcode memory; (3) ease of microprogramming; (4) I/O complexity handled by the processor; (5) no public DMA or memory bus; and (6) extensive software tools to support user microprogramming.

A number of application systems have been developed to date on the MBB: the emulation of a technologically obsolete minicomputer to preserve the investment in software; the development of a new high level language oriented architecture; and the use of dedicated functions which are completely microcoded without the presence of a higher level macro machine. The successful engineering of these different systems has further proven the inherent flexibility and simplicity of a carefully designed microprogrammable machine.

Acknowledgments

The design of a system such as the MBB must of necessity involve many more minds than the authors

themselves represent. Frank Heart provided overall guidance and support throughout the project. Alan Nemeth, Carl Howe, Bernie Cosell, and Sheng-Yang Chiu helped make the C/70 application a reality as did Robert Weissler for the C/30. Steve Geyer and Rob Greene have supported the software and hardware efforts respectively. Bill Mann helped with the system microcode.

We would like to thank the growing body of users for their patience and helpful feedback during the design and implementation of the MBB and its various applications.

In addition, we gratefully acknowledge the assistance of Kathe Unrath and Bob Brooks in the preparation of this manuscript.

[UNIX is a trademark of Bell Laboratories. PDP-11 is a trademark of Digital Equipment Corp. Storage Module Device (SMD) is a trademark of Control Data Corp.]

References

[1] T.G. Rauscher and P.M. Adams, "Microprogramming: A Tutorial and Survey of Recent Developments," IEEE Trans. on Computers, Vol. C-29, No. 1, pp. 2-20 (Jan. 1980).

[2] M.V. Wilkes, "The Best Way to Design an Automatic Calculating Machine," Report of the Manchester University Computer Inaugural Conference, Manchester University, England, July 1951, pp. 16-18.

[3] A.B. Salisbury, Microprogrammable Computer Architectures, New York: Elsevier Publishing, 1976.

[4] C.P. Thacker, et al., "Alto: A Personal Computer," Computer Structures: Readings and Examples, Siewiorek, Bell and Newell (eds.), McGraw-Hill, 1980.

[5] Nanodata Corp., QM-1 Hardware Level User's Manual, 2nd Ed., Rev. 4, March 1976, Williamsville, NY.

[6] W.T. Wilner, "Design of the B1700," AFIPS Conference Proceedings, Vol. 41. (1972 FJCC) pp. 489-497.

[7] C.G. Bell and J.C. Mudge, "The Evolution of the PDP-11" Chapter 16 of Computer Engineering, C.G. Bell, J.C. Mudge, J.E. McNamara (eds.), Digital Press, 1978.

[8] W.A. Wulf and S.P. Harbison, "Reflections in a Pool of Processors," Carnegie-Mellon University Report CS-78-103, Feb. 1978.

[9] R. Weissler, M. Kraley and P. Herman, "MBB Microprogrammer's Handbook," BBN Report No. 4268, Feb. 1980.

[10] D. Katsuki, E. Elsam, W. Mann, E. Roberts, J. Robinson, S. Skowronski, and E. Wolf, "Pluribus--An Operational Fault-Tolerant Multiprocessor," Proc. of the IEEE, Vol. 66, October 1978.

[11] S.M. Ornstein, W.R. Crowther, M.F. Kraley, R.D. Bressler, A. Michel and F.E. Heart, "Pluribus--A Reliable Multiprocessor," AFIPS Conference Proceedings, Vol. 44, May 1975, pp. 551-559.

[12] F.E. Heart, R.E. Kahn, S.M. Ornstein, W.R. Crowther and D.C. Walden, "The Interface Message Processor for the ARPA Computer Network," AFIPS Conference Proceedings, Vol. 36, June 1970, pp. 552-567.

[13] S.M. Ornstein, F.E. Heart, W.R. Crowther, H.K. Rising. S.B. Russell and A. Michel, "The Terminal IMP for the ARPA Computer Network," AFIPS Conference Proceedings, Vol. 40, June 1972.

[14] S.C. Butterfield, R.D. Rettberg and D.C. Walden, "The Satellite IMP for the ARPA Network," Proceedings of the Seventh Annual Hawaii International Conference on System Sciences, Honolulu, Hawaii, January 1974, Computer Nets Supplement, pp. 70-73.

[15] I. Jacobs, E. Hoversten, N. Abel, R. Binder, R. Bressler, W. Edmond and E. Killian, "Packet Satellite Network Design Issues," National Telecommunications Conference Record, November 1979.

[16] B.W. Kernighan and D.P. Ritchie, The C Programming Language, Prentice Hall, 1978.

[17] D.M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," CACM, 17, No. 7 (July 1974), pp. 365-375. (Also reprinted in Bell System Technical Journal, Vol. 51, No. 6, pp. 1897-2305, July-August 1978, special issue on UNIX).

[18] W.F. Mann, S.M. Ornstein, and M.F. Kraley, "A Network-Oriented Multiprocessor Front-End Handling Many Hosts and Hundreds of Terminals," AFIPS Conference Proceedings, Vol. 45, June 1976, pp. 533-540.