Hardware, Firmware, Software Technology in Microcomputer Systems

T. G. Lewis
Computer Science Dept.

H. Jafari
Electrical Engineering Dept.

Oregon State University

78 - 01 - 01

HARDWARE, FIRMWARE, SOFTWARE TECHNOLOGY IN MICROCOMPUTER SYSTEMS

T. G. LEWIS

COMPUTER SCIENCE DEPARTMENT
OREGON STATE UNIVERSITY
CORVALLIS, OREGON

For "Advances in Information Systems Science," Julius Tou ed., Plenum Publishing Company, 227 West 17th Street New York, New York 10011

Editor: Julius Tou
University of Florida
Gainesville, Florida 32611

1. Hardware Organization

1.1 Introduction

Computer systems advance by revolutions rather than evolutions. The jump from vacuum tube machines to solid state transistor machines was revolutionary. Never before had computers been reduced in size, cost, and computation time until this revolution.

Computer revolutions are enumerated by a generation number. First generation computers were based on vacuum tube technology, second generation was based on transistors. We can say that the current generation is based on large-scale-integration LSI.

The LSI age of computing is no longer denoted by a single generation number because LSI is causing many upheavals in computing. The upheavals are too numerous and spread over too short a time for numbering systems to keep up. Even the terminology needed to describe the changes is hard pressed to keep pace.

It is important to realize the significance of terminology. One measure of the rate of technological change is the rate of semantic shifts occurring in the language. For example, a microprocessor is a cpu in a single LSI transistor wafer. A few years earlier, however, a microprocessor was any microprogrammable cpu. To avoid confusion, the following definitions will be used throughout this chapter.

A microcomputer is a cpu memory, interfaces and boards needed to package a microporcessor to make it appear as a computer to the user. A microcomputer may be microprogrammable if it has a control memory and sequencing unit that allows firmware programming. For the purpose of this chapter, a firmware program will be any program residing in a read-only memory, ROM. This definition sidesteps the problem posed by microcomputers that emulate their instructions as compared with microcomputers that take instructions from either ROM or random-access-memory, RAM. In either case, a microcomputer is said to be microprogrammed if programs reside in ROM, regardless of the processor's architecture*.

A microcomputer that incorporates ROM and a microprocessor in a single unit is called a grand-scale-integration GSI microcomputer. An example of a GSI microcomputer is the common pocket calculator. Each calculator has a processor and a ROM containing the instructions for executing each button stroke. Since programming in the stored program tradition is not possible by the user, the calculator is considered a single unit of GSI equipment.

*Emulation can be roughly defined as simulation of one computer on another computer. Typically, the simulator resides in ROM as part of the control unit of the host cpu.

Microprocessors are packaged in dual-in-line packages called DIP chips. A DIP chip is typically a 40-pin ceramic package about one to two inches long, one-fourth to one inch wide, and less than 1/2 inch thick. Access to the resident circuitry is through the 40 pins. Because of their size and packaging, microprocessors are often called chip computers.

It is the decreased size, cost, and power consumption that is responsible for the chip computer revolution. This revolution permeates the application, design, programming, and manufacture of computers, their peripherals, and the people who use them. Such pervasion into science, technology, and society will have far reaching effects for the future.

The purpose of this chapter is to narrow the discussion of this revolution to a specific technical area. The discussion will focus on fundamental technological concepts underlying the revolution. For this purpose we examine three architectures, three software systems, and conclude with an analysis of resource sharing and the impact of microcomputing on sharing.

1.2 A Simple Microprossor

Perhaps the simplest microprocessor would be an LSI circuit for adding, subtracting, and performing I/O on a two-bit word of memory. Such a small processor holds little interest because of the elaborate encoding of data and extensive programming effort required to make the hypothetical processor useful. What then, is the lower limit of "size" acceptable for a practical microprocessor?

The first requirement for a practical microcomputer is that decimal numbers be easily represented in the microprocessor storage unit. This means a minimum word length of 4 bits, since digits 0-9 can be encoded in BCD with 4 bits. A four bit computer can perform most functions of a decimal calculator with relative ease.

Greater parallelism leads to speed and the potential for extensive programming. A four-bit word can address only 16 locations in memory while an 8-bit word can hold 256 addresses. Furthermore, an 8-bit instruction word has greater capability for an improved instruction set.

Obviously, the same arguments for 16-bit processors can be applied to 8-bit processors. The improvements of a 16-bit computer certainly make their development inevitable.

Before any technological device is made available on a widespread basis, there must be a <u>dollar-volume force</u> behind the technological device. <u>Dollar-volume force</u> is defined as the product of unit-price times market-volume.

$_VOL_FORCE = (Unit-price) (Market-volume)

A microcomputer valued at $10 and sold 10,000 times is a technological device with a $100,000 dollar-volume force behind it.

This leads to the concept of technology availability, which in turn partially determines the design of a simple microprocessor. An invention becomes available only when the dollar-volume force is significantly increased by the proposed invention*.

A two-bit microcomputer, while feasible for many years before microcomputers were generally available, lacked potential for increasing the dollar-volume force. A four-bit computer, because of its usefulness in pocket calculators and BCD processing machines successfully increased the dollar-volume force thereby making the first microprocessor available. Hence the simplest microprocessor that was both technologically possible and economically feasible was the 4-bit processor.

An 8-bit microprocessor offers many technological advantages over 4-bit processors. The advantages in themselves are not sufficient to bring about a true

*It could be argued that television arose without an initial dollar-volume force behind prior developments. However, it is also possible to view TV as an outgrowth of radio, in which case the dollar-volume force is increased. In general, consumer electronics are marketed only when new markets expand the dollar-volume.

8-bit microcomputer. Instead, the dollar-volume force had to increase before 8-bit microcomputers were possible.

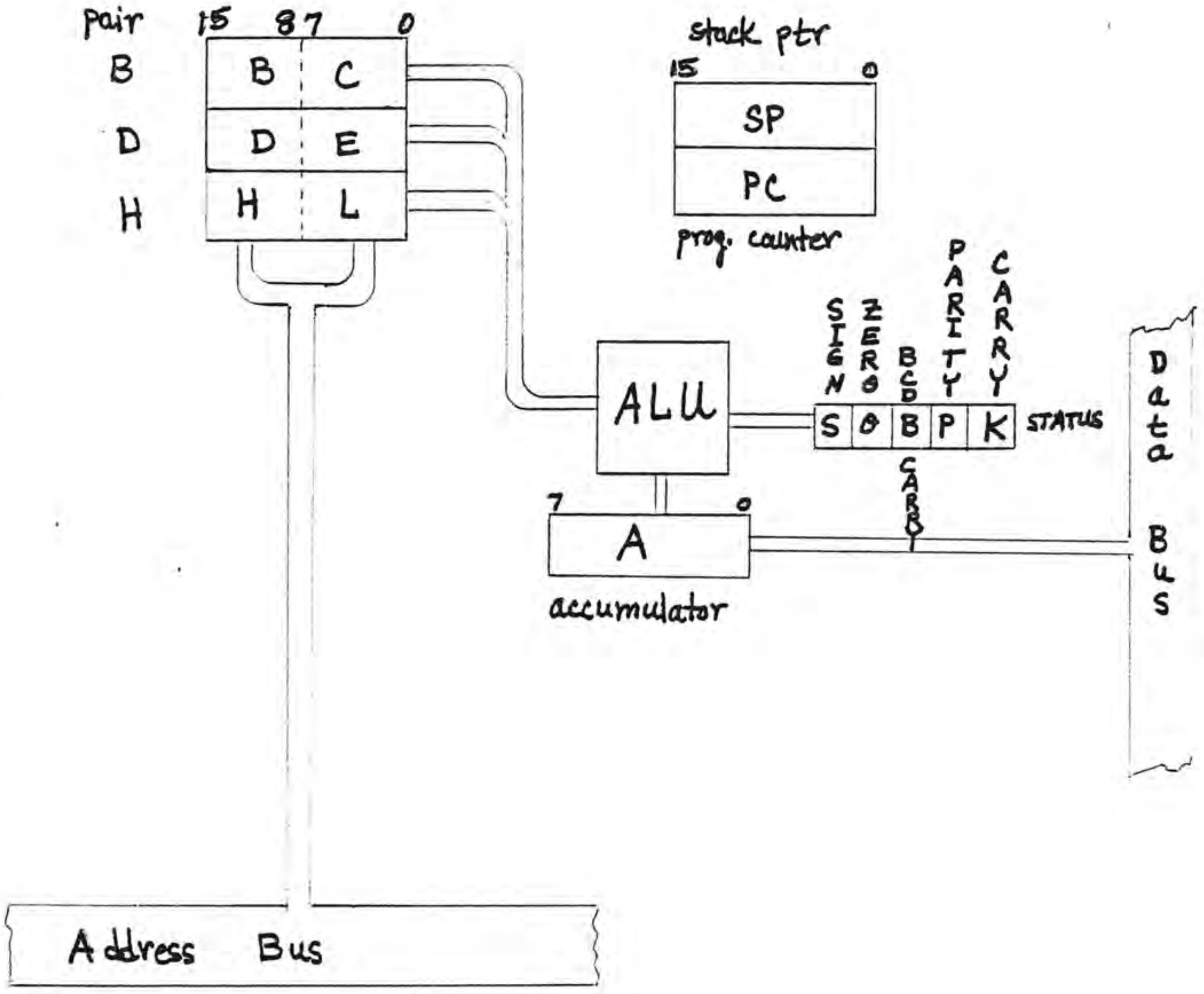
The 8-bit microcomputer increased the dollar-volume force sufficiently to drive 8-bit microprocessors into general availability. The reason is that ASCII and EBCDIC encoding are 8-bit codes, floating point arithmetic is facilitated, and addressability is improved. These primitive improvements manifest themselves in more sophisticated software packages that in turn expand the usefulness of 8-bit computers beyond the pocket calculator market. Therefore, the 8-bit microcomputer owes its driving dollar-volume force to applications that transcend pocket calculators.

The 8-bit microcomputer architecture of Figure 1 shows a simple microprocessor organization that typifies the early generation of microprocessors*. The machine of Figure 1 is a register-transfer machine. This means that all operands are either stored in the working registers or they are accessed by pointers stored in the working registers.

One of the dollar-volume driving forces behind the eventual availabilty of 8-bit microprocessors was the advantages of multiple precision calculators. This advantage is noted in the double register feature of Figure 1. Registers B, D, and H are treated as 16-bit operands in certain operations. Also, 4-bit

* The architecture of Figure 1 is an Intel 8080, see reference 7.

Figure 1. A Simple 8-Bit Microprocessor



precision is preserved by the "BCD CARRY" bit B in the status register. We will study the behavior of bit B in a later programming example.

Each instruction of the microprocessor of Figure 1 is one, two, or three bytes long. The niladic operators* such as "SET CARRY", "COMPLEMENT", and "DECIMAL ADJUST" require only one byte of memory. The monadic operators such as "ADD", "AND", and "COMPARE" require two bytes because of extended addressing. The dyadic operators such as "MOVE" and "LOAD INDEX" require three bytes. In most cases operands are working register or memory register, either one accessed through the pointer loaded into register pair H-L.

The A, B, C, D, and E registers are used to accumulate results from the ALU. The H and L registers combine to form a 16-bit memory address. The address in H-L is used to load or store single bytes from or to memory. The memory may be ROM, in which case store operations via H-L are meaningless.

*Niladic operators have zero explicit operands, monadic operators have one explicit operand, and diadic operators have two explicit operands. For example, in the DAA operation, the accumulator is implied.

The memory of this 8-bit microprocessor is hierarchial: Register A is immediately available while registers B, C, D, E, H, and L are available as operands in the instruction set. Main memory is at a third level of access because bytes come from main memory by way of the pointer in H and L.

The program counter, PC and stack pointer, SP operate as expected. The PC register holds the 16-bit address of the next instruction to be executed. The SP register holds the address of the top element of a push down stack. A "PUSH" operation causes 16 bits to be placed on the stack after SP has been decremented by two.

$$(SP) \longleftarrow (SP) - 2$$

After a "POP" operation, the SP register is incremented. Therefore, the stack grows toward the low end of memory. This feature guarantees that 16-bit register pairs are loaded and unloaded in the order needed during multiple precision calculations.

The stack facility provides recursive subroutining. During a "CAL" to subroutine, the "old PC" is saved on the push down stack. During a return from subroutine the "new PC" is popped from the stack.

Input and output is performed through the A register under program control. The "IN" instruction fetches an 8-bit byte from a specified device and "OUT" copies the contents of register A onto the data bus.

Interrupts are allowed with the "EI" instruction and disabled with the "DI" instruction. There is no automatic vectoring* of traps. This weakness must be overcome through considerable programming by the user.

Figure 2 demonstrates a short segment of machine level code for the microprocessor of Figure 1. The program computes the sum of two 3-byte numbers stored at symbolic locations, FIRST and SECND. The answer is stored back into FIRST.

$$\begin{array}{r} \text{32AF8A}_{16} \\ + \;\; \text{84BA90}_{16} \\ \hline \text{B76A1A}_{16} \end{array} \qquad\qquad \begin{array}{l} \text{SECND} \\ + \;\; \text{FIRST} \\ \hline \text{FIRST} \quad \text{(answer)} \end{array}$$

The program demonstrates how multiple precision calculations are performed and how the lack of indexing is overcome by programming. The "ADD:" segment of code initializes two pointers to the operands. The B-C

*Automatic I/O or vectored I/O is a feature on many minicomputers. An I/O vector is a memory cell containing status information and a pointer to a service routine. Upon interruption, the service routine is called.

register pair point to FIRST after the load-index-immediate LXI instruction. The H-L pair points to SECND after execution of the LXI instruction.

The "LOOP:" segment performs addition on three bytes, from right (least significant byte) to left (most significant byte). This is done by accessing the byte pointed to by B-C, accessing the byte pointed to by H-L, and performing the ADC instruction. The ADC adds with CARRY included so that multiple precision carry-outs are saved in bit K=CARRY. The STAX instruction uses B-C as a pointer to FIRST. The DCR decrement instruction subtracts one from register C because this segment of code also uses register C as a loop counter. This dual use of B-C (as pointer and counter) may lead to errors in the program unless the data is stored on a 256-byte page boundary. The next instruction tests for completion.

The operand pointer B-C also is used as a loop counter in the previous example. This is necessary in the limited architecture of a simple microprocessor.

The result, however, is greater software overhead, possible errors as pointed out above, and added effort.

Most microprocessors are oriented toward decimal BCD calculations. The "DAA" decimal adjust instruction is provided to translate partial binary results back into BCD results after an arithmetic operation. The following example will demonstrate this.

	25_{BCD}	=	0010	0101_2
+	7_{BCD}	+	0000	0111_2
	32_{BCD}		0010	1100_2

The BCD numbers 25 and 7 are stored as binary nunbers, internally. When the microprocessor adds them together, it produces the <u>binary sum</u> 0010 1100 as shown to the right. This sum must now be converted to a BCD numeral instead of a binary number. The DAA instruction tests the BCD CARRY bit in the status register. Depending upon the value of the BCD CARRY, the upper and lower byte of the result, and the condition of the CARRY bit, the DAA instruction will either add $+00_{16}$, $+06_{16}$, $+ 60_{16}$, or $+ 66_{16}$ to the result.

In the case of the sample calculation, above, the result is "corrected" by addition of $+06_{16}$.

	0010	1100	
+	0000	0110	
	0011	0010	$= 32_{BCD}$

The DAA operation produces a BCD result that would have been obtained had the micropressor been capable of direct decimal addition. Thus, 25+7 = 32 as desired.

We could modify the sample program of Figure 2 to produce BCD results by giving the data in BCD format and using a DAA instruction after each addition. This would mean inserting a DAA instruction between the ADC and STAX instruction in the LOOP segment of code.

The simple microcomputer described here has an extensive instruction set and a 16-bit addressing capability. It has found applications in a variety of first-time computer uses. Indeed, its simplicity is a virtue in many new applications.

There are both obvious and subtle deficiencies in the simple microprocessor design we have just examined. Basically the deficiencies stem from the microprocessors weak indexing and addressing capability and underdeveloped interrupt handling facilities. The next microprocessor studied partially overcomes these deficiencies and represents a typical second generation microcomputer processor.

Figure 2.   A Sample Program For the Microprocessor
             of Figure 1.

```
FIRST:       DB      90H       Hexadecimal data bytes...
             DB      BAH       stored in reverse order.
             DB      84H

SECND:       DB      84H       Hexadecimal data bytes...
             DB      AFH       stored in reverse order.
             DB      32H
*
*     (A)   sum of multiple precision add
*     (B-C) index to FIRST operand.
*     (C)   length of operands, in bytes.
*     (H-L) index to SECND operand
*     FIRST operand and answer (sum).
*     SECND operand
*
ADD :       LXI   B,FIRST   set (B-C) pointer to FIRST.
            LXI   H,SECND   set (H-L) pointer to SECND
            XRA   A         clear CARRY bit, set A=0.
LOOP:       LDAX  B         get a byte of FIRST
            ADC   M         (A)  -(A) + (H-L) + (CARRY).
            STAX  B         put a byte into FIRST
            DCR   C         done...
            JZ    DONE      ...otherwise, continue.
```

```
         INX   B      increment to next byte of FIRST
         INX   H      increment to next byte of SECND
         JMP   LOOP   add next byte.
DONE:                 continue
         END
```

1.3 An Improved Microprocessor

Once a dollar-volume force is set into motion by a technological breakthrough, many minor stepwise improvements follow. The improvements act as minor pertubations in the revolution. Nonetheless, it is by way of these smaller steps that we build-up to a subsequent breakthrough.

The advantages of 8-bit microcomputers soon become obvious to many who would use them for purposes not forseen by the designers. These new applications were implemented by custom made software resident in the microprocessor memories. It became evident to many software engineers that the simple 8-bit microprocessor studied in section1.2 could be improved to alleviate some of the problems associated with more general applications.

The improved microprocessor of Figure 3 gives the false impression that the microprocessor is actually less capable than the one just studied.* Actually, the simpler organization belies a more powerful microprocessor instruction set. The reason for its improved organization centers on the index register, IX and 72 unique instructions.

* The architecture of Figure 3 is a Motorola 6800 MPU, see reference (8).

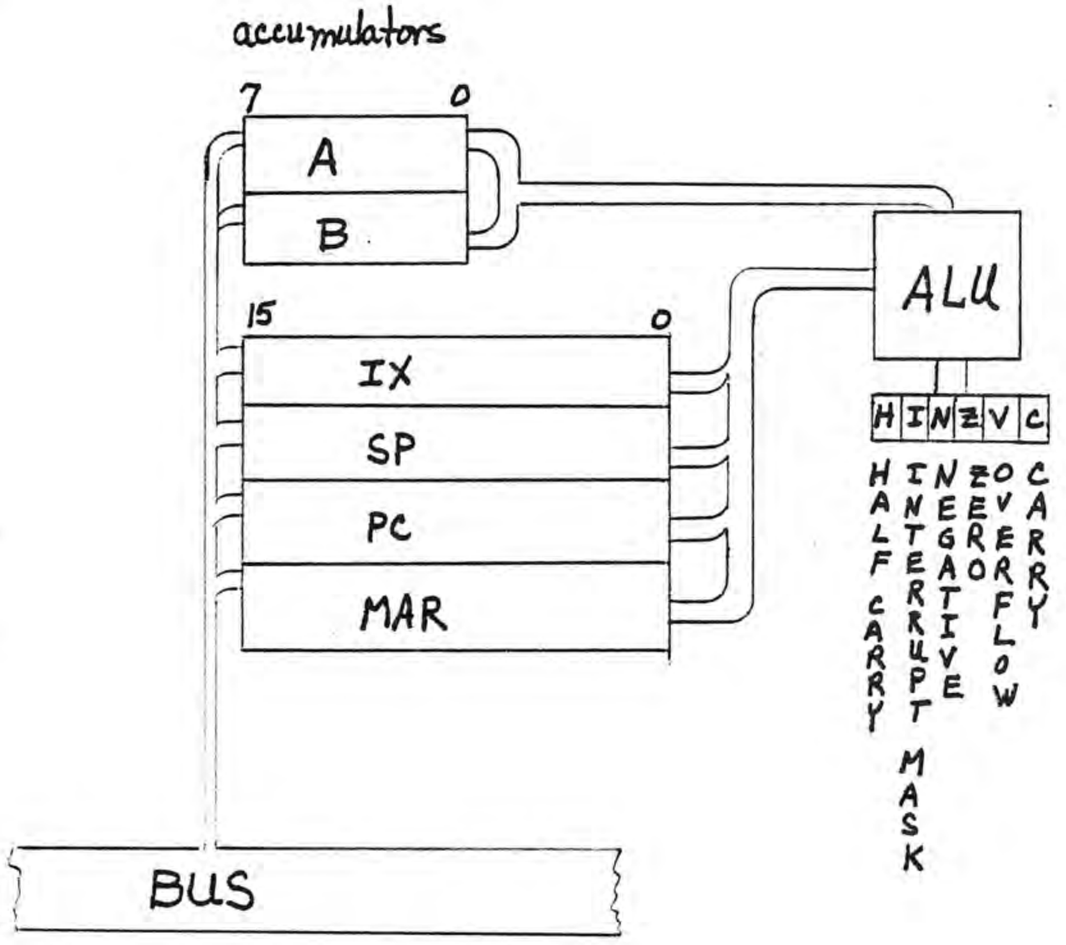
All operations are performed between registers A, B and memory, or between registers A and B themselves. For example, the "ADDA" and "ADDB" instructions sum the contents of a memory byte at the location specified by register MAR. They also can sum the contents of A or B and store the result back into A or B.

Instruction operands are fetched from either A or B and via MAR from memory as stated before. The index register, however, may enter into addressing via MAR. The contents of IX are added to the address in MAR to compute an effective address. This added capability greatly simplifies programming and requires smaller programs as illustrated in Figure 4.

The SP and PC registers operate as before with the stack in main memory. The stack is used for recursive subroutine calls but may also contain intermediate results or parameters to a subroutine.

The condition codes include HALF CARRY, and CARRY as in the simple microprocessor. This enables the improved microprocessor to perform both BCD arithmetic on single bytes and multiple precision arithmetic on strings of bytes. The INTERRUPT MASK bit provides control over interrupt enables. For example, I is set (=0) with the "SEI" instruction. Interrupt service routines are entered recursively.

Figure 3. An Improved 8-bit Microprocessor



Input/output is generalized through the use of a central bus. The bus handles addressing of memory, peripherals, and other microprocessors through generalized interface chips. Each interface chip is dedicated to either cycle - stealing direct memory access, or to jamming data into the A or B accumulator. Therefore, to output a byte from register A, it is necessary to perform a store instruction, STAA IOBUF. This store accumulator A instruction addresses the interface chip IOBUF as if it were a location in memory. Whatever device is attached to IOBUF receives the byte of data.

Examination of the multiple precision addition of FIRST and SECND byte strings of Figure 4 reveals a much simpler, shorter, and understandable program when compared with Figure 2.

In Figure 4 the microprocessor is programmed to add together any two byte strings of length N and store the result back in the FIRST string. This is done by using the index register as a pointer into the strings. The addition is done right-to-left with the CARRY bit linking together partial results.

The DAA decimal adjust instruction can be inserted where shown if BCD arithmetic is desired. The DEX decrement instruction sets the zero Z indicator when X has been reduced to zero. This signifies termination of the loop.

Figure 4.   A Sample Program For the Microprocessor
            of Figure 3.

```
FIRST     FCB    $84       Hexadecimal data bytes...
          FCB    $BA       stored in forward order.
          FCB    $90

SECND     FCB    $32       Form Constant Byte for...
          FCB    $AF       second operand.
          FCB    $84

N         EQU     3        length of operands.

AD2N      CLC              clear CARRY.
          LDX     #N       load length of operands...
                           into index register.
LOOP      LDAA   FIRST,X   get least significant byte
          ADCA   SECND,X   add with CARRY
*        (DAA)             (could go here for BCD add--
                            see text)
          STAA   FIRST,X   put result back into FIRST.
          DEX              decrement index pointer.
          BNE    LOOP      Done?
DONE
```

The realization of greater capability in an 8-bit processor suggests that other improvements may be possible. In the next demonstration microprocessor we examine several areas of improvement representing the third generation of microprocessor organization.

1.4 A Sophisticated Microprocessor

While the 4 and 8-bit microprocessors discussed earlier represent sophisticated programmable logic, the truly sophisticated microprocessor has the replacement of mini and midi computers as its dollar-volume driving force. For any microcomputer to move into applications traditionally held by minicomputers greater software development potential must be possible through improved architecture. Therefore, it is the software development capability that differentiates the sophisticated microprocessor from earlier generations of microprocessors.

The dollar-volume force is increased by a microprocessor with ease of programming, sophisticated memory addressing, and expansion capability built into the chip. Such a microprocessor competes with minicomputers for acceptance. For this reason, the improved microcomputer must also be an improvement over many contemporary minicomputer architectures. For example, it must overcome limitations placed on main memory size in favor of a large memory address space.

The sophisticated microprocessor is a 16-bit parallel, word, byte, and bit addressable machine with versatile memory addressing facilities, strong interrupt handling features, automatic memory mapping, and context switching* ability. How can all of these requirements be met in a single microprocessor?

The first architectural innovation needed to satisfy the stated requirements is the elimination of working registers. Architectures based on working registers as a separate resource invite inefficiencies in at least two fundamental ways. First, they invite unnecessary software overhead because the registers must be loaded and stored, frequently. The loads and stores do not produce results; they only prepare operands for operations that produce results. It would be more efficient to directly perform the operations on the operands regardless of their location in memory.

Secondly, the registers of a traditional architecture are shared by every process in the system. Whenever processing switches contexts, the registers must be saved and then restored. Context switching may occur whenever a subprocedure is invoked either through an interrupt or else by normal program execution. Clearly, the need to share working registers has caused many problems in the design of operating sytems.

Register allocation and management problems are avoided in machines organized around a pushdown stack architecture. The stack is stored in memory and every

*Context switching is defined here as a state change requiring a new environment. The context of this machine changes whenever a subroutine, interrupt, or process change occurs.

operation in the instruction set operates on the top elements of the stack. Context switching is simple and fast because the stack has the natural ability to nest environments or mark the top of the stack in order to return later to a previous state. Unfortunately, stack machines restrict access to other portions of memory and require wasteful loads (push) and stores (pop) to prepare operands for processing.

An example that illustrates an inherent weakness of stack architectures is the process of dynamic storage allocation. Dynamic storage allocation is performed by programs written in block-structured languages such as ALGOL and PL/1. Upon entry into a nested block, the local variables are allocated space by creating a segment of storage on the top of the pushdown stack. As processing continues, the stack continues to grow, and indeed when a second nested block is encountered, it is possible to require a second block of local storage to be allocated on the top of the stack. At this point, the stack architecture must be able to also access the data stored in the outer block. This non-local data is not on the top of the stack, but instead it is many stack frames into the stack. Thus, the top-of-stack operations no longer are able to access the outer data without additional modes of addressing.

The stack machine can be stressed further with the problem of global dynamic allocation posed by PL/I derivatives. The ALLOCATE construct of PL/I makes it possible for a programmer to create a variable (and its space) at any time in the execution of a program. Conversely the FREE construct allows a programmer to destroy the variable (and its space) at any point in the program. These operations fragment pushdown stack storage disciplines and the resultant overhead becomes prohibitive. Typically, this problem is handled by bypassing the stack and resorting the traditional addressing modes and traditional load/store overhead operations. In otherwords, pushdown stack mechanisms have only limited advantages over traditional organizations.

A sophisticated microprocessor must be able to switch contexts as easily as a pushdown stack machine and yet access data as randomly as a register machine.

In addition, it would be highly desirable to either maintain a very large number of working registers or else eliminate them entirely in favor of direct access to memory words. The microprocessor and RAM (random access memory) of Figure 5 is a step in this direction.

The microprocessor of Figure 5 contains three internal registers called CONTEXT POINTERS and three files of 6 registers each called the MEMORY MAPS. All other registers are part of main memory and are accessed through the CONTEXT POINTERS working in harmony with a MEMORY MAP.

The CONTEXT POINTER WP (working pointer) is a 16-bit address that is modified by BIAS i, where i is determined by the value of WP and the LIMIT registers. If LIMIT 2 $<$ WP $\leq$ LIMIT 1 then a 20-bit effective address is formed by adding (BIAS 1) $*2^5$ to WP. The BIAS register is shifted left 5 bits before addition to effect the multiplication by 32. If LIMIT 2 $<$ WP $\leq$ LIMIT 3, then BIAS 2 is used to compute a 20-bit effective address. Finally, when WP $>$ LIMIT 3 the BIAS 3 offset is used.

Clearly, the memory mechanism adds to the power of this microprocessor. Programs and data are all referenced through the MAP. This means that large memory spaces can be addressed and segments containing data or programs need not be contiguous.

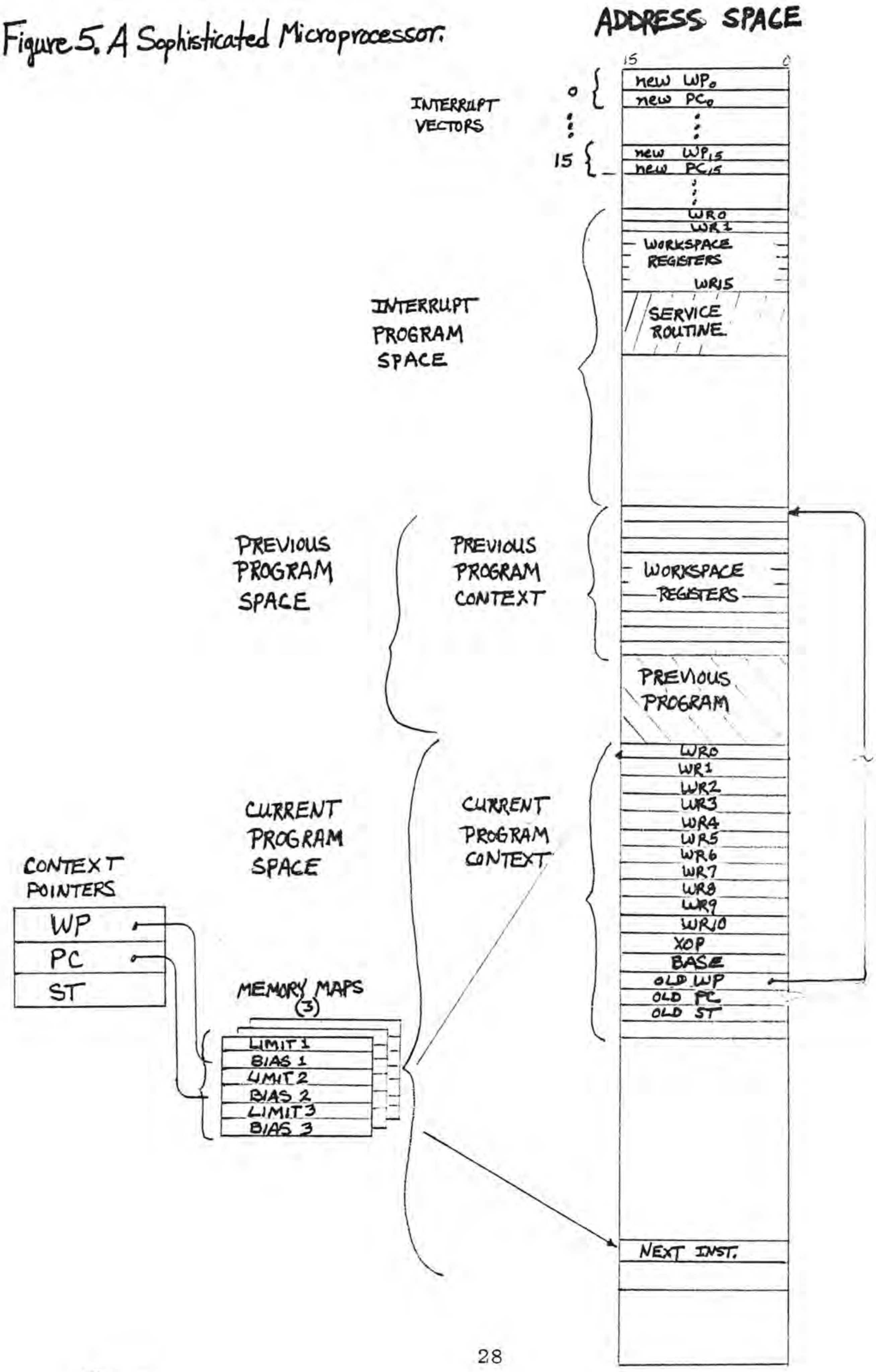
* This is the Texas Instruments 990 series processor, see reference (5).

Notice in the description thus far, that the microprocessor manipulates pointers to data as opposed to manipulating data directly in internal registers. This level of indirection is the source of much of the microprocessor's power and sophistication. Indirection supplies the ability to do context switching with the ease of a pushdown stack machine.

The WP pointer (with modification by the MAP) references a segment of memory called the CURRENT PROGRAM CONTEXT. The first 16 words of this context serve as "workspace registers". WR0 through WR15 appear to a programmer as working registers. Each context has its own bank of workspace registers WR0 through WR15. Observe that WR11 through WR15 are special purpose registers as well as being general registers. In particular, WR14 is used to hold the OLD WP of the previous context. Thus the advantages of a pushdown stack are realized while at the same time the advantages of random access remain. Furthermore, local register space is protected from non-local contexts that endanger the integrity of data stored in the registers.

The example of Figure 5 also demonstrates how interrupts are vectored to the appropriate service routine. The INTERRUPT VECTORS contain "new WP"

Figure 5. A Sophisticated Microprocessor.



and "new PC" addresses that point to the service routine and its workspace. Since the old WP and old PC are automatically saved in the service routine workspace, returns from interrupts are simplified. Interrupts may be nested inside of interrupts.

The XOP (extended operations) register in WR11 of the workspace provides a means for extending the hardware or software of the sophisticated microprocessor. There are 16 instructions not implemented in the processor. When one of the unimplemented opcodes is encountered, a trap occurs and the microprocessor tests the effective address generated by the "illegal" opcode to determine if the address points to hardware or software. This pointer is stored in WR11 (XOP) of the new context. If the instruction is simulated by software, the routine at XOP is executed. If the instruction is interpreted by hardware, the execution at XOP is performed and the results returned to the workspace. The reader is advised to keep this feature in mind for a later discussion concerning LSI software, see section 2.2.

The driving force behind the sophisticated microprocessor is the dollar-volume expansion that results from replacing minicomputers with microcomputers.

Figure 6. A Sample Program For The Microprocessor of Figure 5.

```
        TITL        'MULTIPLE BYTE ADD ROUTINE'
        IDT         'ADDITION'
*
*       Set-up      WP,PC,ST and Workspace Registers
*
OS      DATA        WS, PC, >F     initialize WP,PC and ST.
WS      DATA        FIRST          WR0 points to FIRST
        DATA        SECND          WR1 points to SECND.
        DATA        > 3            WR2 indexes operands.
        BSS         26             WR3-WR15 not used.
FIRST   DATA        >0084, >BA90   right justified operand.
SECND   DATA        >0032, >AF84   right justified operand.
*
*       COMPUTE SUM
*
        CLC                        clear carry bit in ST
LOOP    ABC  @SECND(2),@FIRST(2)   add low-to-high bytes.
        DEC    2                   decrement WR2 index
        JNE  LOOP                  done?
```

This technological jump is reflected in greater programming "power" when compared with the previous microprocessors. Figure 6 illustrates how the 3-byte addition routine of Figure 2 and Figure 4 appears if programmed in the assembly language of the sophisticated microprocessor.

Notice the actual executable segment of Figure 6 is only 4 words long. This is a 50% reduction in program length and corresponding execution time over the improved microprocessor routine in Figure 4. The improvement is possible because memory-to-memory operands are allowed and WR2 is used as an index-counter. This mode of addressing is possible without sacrificing the advantages of rapid context switching.

Also, a fair comparison of microprocessors must account for the overhead required to set up the work-space and data. This overhead was sizeable in the sample of Figure 6, but of minor consequence in realistically sized programs. Once the context environment is set up for each context, the advantages of rapid switching offset the set-up inconvenience.

The OS statement illustrates how three words are used to initialize WP, PC, and ST in the CONTEXT POINTER registers. The >F bit pattern supplies initial condition codes for the active ST register.

The WS statement initializes the workspace for this context. A pointer to FIRST and SECND are set up in WR0 and WR1, and the length of operands is set up in WR2. The BSS pseudoop simply reserves space for the other workspace registers.

The FIRST and SECND operands are stored in two 16-bit words. They are hexadecimal constants designated by the assembler " > " notation.

The summation is performed by clearing the CARRY bit in ST, performing an "add with carry" ABC, and looping until all three bytes have been summed.

The @ notation indicates that the data is at FIRST plus index register 2, and at SECND plus index register 2, respectively. The first operand is added to the second operand. The sum is stored in FIRST.

Finally, the index register is decremented and the loop is repeated as long as WR2 is not equal to zero (NE). Execution of these four instructions takes ten machine cycles to sum all three bytes.

The idea behind this microprocessor is to gain sophistication through elegant simplicity. The elegance of stack processing and direct memory-to-memory random access processing are retained without loss of simplicity.

The goals of this microprocessor are futile if we cannot find ways to tap the simple elegance of this architecture. This requires programming in a form consistent with the cost of a microcomputer. How can we cope with programming a microcomputer?

2. Firmware

2.1 Definitions

Firmware is programmed hardware. It is software merged into hardward because it combines programming with non-alterable hardware. How can this be?

A <u>microprogram</u> was defined earlier as a program residing in ROM (read-only-memory). Because it is unalterable, the microprogram is called firmware.

Software for a microcomputer is turned into firmware by storing* it in ROM. This means that constants may be taken from the program space but that results can never be returned to the program space. Thus program and data must be separated. The side effects of alterable program spaces are eliminated.

Traditionally, (and more precisely) the concept of microprogramming applies to the firmware resident in the control unit of a computer. Since the control unit directs a computer during hardware interpretation of machine language instructions, microprogramming

*The ROM is initially "burned" by passing a high voltage through the memory. This high voltage distructively alters the ROM leaving a bit pattern which can be output during emulation.

is meant to determine the nature of machine language. In a sense, the control unit is another computer inside of the machine language level computer.

The invention of microprocessors and microcomputers blurred the precise meaning of microprogramming. The trend is to accept the definition used here. We will see why this definition may persist when the concept of LSI software is expanded.

2.2 Software LSI

LSI (large scale integration) is responsible for the microcomputer revolution. LSI hardware technology reduced the cost of entire cpus, memory, and peripherals to the point where hardware is "free".

Unfortunately, software costs have continued to climb due to increased complexity in systems and the fact that software production is essentially a custom manufacturing process. Daily, software programmers implement their customized versions of mathematical routines, payroll routines, etc. Most of these software packages have been written hundreds of times with little knowledge of their duplication.

Manufacturing of software must turn to "software" LSI techniques analogous with hardware LSI techniques before advances can be made in reducing software costs.

An obvious step toward reducing the cost of software is to share identical programs with many different microcomputers. Pocket calculators, for example, share the same SIN (x), %, and 1/x routines with thousands of microcomputers. This is done by encapsulating software in a ROM which is mass produced as an LSI software module.

An <u>LSI software module</u> is a microcomputer and companion ROM memory containing firmware dedicated to a specific function or functions. The scientific subroutine package of a large computer can be economically replaced by an LSI software module similar to the pocket calculator. Once this module is "plugged in" it is never "reinvented" by a software programmer. Instead, it is forever encapsulated in firmware.

A software module must be used frequently and be thoroughly tested before it is committed to LSI encapsulation. Wide acceptance of the module is based upon frequent use, and recognizing that the module is a "primitive". Obviously, since it is shared by thousands of microcomputers, it must work properly.

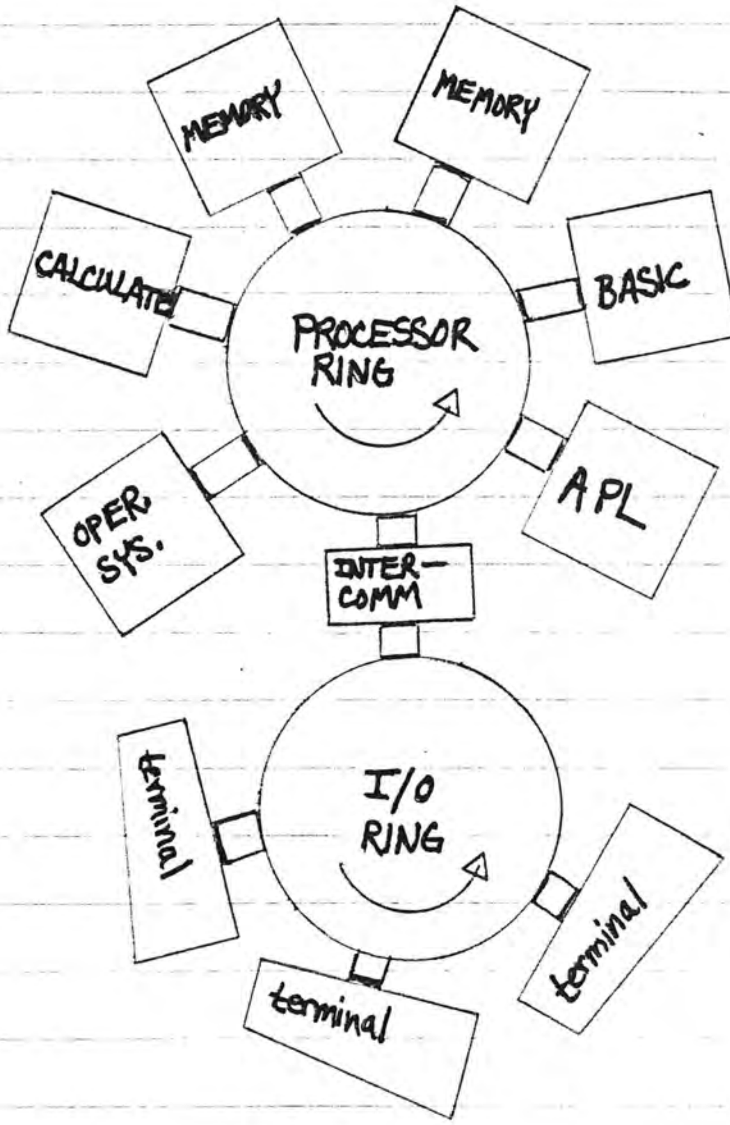
A language interpreter such as BASIC or APL is easily encapsulated as an LSI software module. These languages have an ad hoc standard that assures relative stability. Thus, the standard APL and BASIC are recognized as primitives. Mass production of APL or BASIC interpreters is accomplished by mass production of ROMs containing bit patterns for controlling a microcomputer. The result is low cost "software".

The concept of pluggable LSI software leads to the concept of distributed microcomputer processors. Such processors are constructed from LSI software modules. Each module is a ROM and microprocessor dedicated to a specific function.

Distributed microcomputers must be managed in a simple, yet elegant way or else the same complexities that plague larger computers and networks will also plague distributed LSI software microcomputers. The microcomputer ring, Figure 7, is one such approach.

In Figure 7, two LSI software rings have been formed from LSI software modules. The PROCESSOR ring is formed by plugging APL, BASIC, a CALCULATOR, an OPERATING SYSTEM, and two MEMORY units into a circular shift register. In addition, an INTERCOMM module is plugged into both rings to handle communication between the PROCESSOR ring and the I/O ring. Each ring consists of a large circulating shift register memory. Information is introduced into the

Figure 7. The Microcomputer Ring Concept



shift register by any LSI software module interfaced to the ring. Each module has an address corresponding to the address of the shift register word or words connected to its LSI software module interface.

Information is circulated in the ring forever or until removed by one of the LSI software modules. Thus, the counter clockwise circulating shift register provides input on one side and output on the other side of each interface.

The I/O ring consists of a circulating shift register with LSI software modules for terminals plugged into it, and a connection to the PROCESSOR ring. The terminal LSI software modules consist of CRT/KEYBOARD and local memory/microprocessor for controlling terminal activities. The ring interface allows information to flow to other terminals of the ring or to the PROCESSOR ring via the INTERCOMM LSI software module.

The microcomputer ring concept eliminates system software. System programs are LSI software programs encapsulated in ROM. The firmware eliminates the need for protection and increases reliability.

As an example, suppose a terminal user decides to execute an APL statement.

$$A \leftarrow +/A \div \rho A$$

The user logs onto the I/O ring by typing a password into his terminal. This password is copied into the circulating

I/O ring where it continues to circulate until the INTERCOMM module takes it from the I/O ring and enters it into the PROCESSOR ring.

Clearly the messages entered into a ring are accompanied by a source and destination address. Hence, at each interface these addresses are compared to determine which LSI software module should respond. When the addresses match, the LSI module may be busy, thus the message is circulated one full cycle before reaching the destination again. This process is repeated until the message is absorbed by the destination LSI module.

The password is circulated in the PROCESSOR ring until picked off by the OPERATING SYSTEM module. The OPERATING SYSTEM module reverses source and destination addresses and formats a return message. This process also initiates the necessary control tables for this user. These tables are kept in the OPERATING SYSTEM module's local memory or in one of the MEMORY modules.

The terminal user types in a command, next:

APL

This command travels around to the OPERATING SYSTEM again and when the APL statement is entered, the following steps take place. The OPERATING SYSTEM intercepts the statement. Since the user is in APL mode, the OPERATING SYSTEM forwards the statement to the APL LSI software module. The APL module parses the statement and sends out a series of CALCULATOR messages to perform the +/, ⍴ , and ÷ operations. These operations are eventually performed and the results returned to the APL module. The APL module returns a message to the OPERATING SYSTEM. Finally, the user receives a message from the OPERATING SYSTEM and the dialog continues.

The ring structured microcomputer is simple and elegant. Network complexity is not allowed to get out of control because a ring is the simplest kind of network.

System software is manageable in a ring microcomputer because it is modular and encapsulated as firmware primitives. Even when testing a new module, the interaction between the untried module and the other modules is localized. This eases system integration problems in the same way that top-down structured programming does.

The ring network of microcomputers is untested. For example, when the ring shift register becomes full, a contention will arise. In this sense, the ring is a buffer. Further investigation into the properties of rings is needed before conclusions can be drawn.

LSI software is an outgrowth of good programming technique. Programming in single statements is analogous to building a computer from flip-flops. Programming in subroutines or structured control structures with single entry/single exit flow of control can be compared with building a computer from medium scale integrated circuits. Programming with firmware modules can be compared to building microcomputers from LSI microprocessor chips.

2.3 Grand Scale Integration

Grand scale integration, GSI is the concept of combining LSI memory and microprocessor units into a single chip. The memory is "charged" with a firmware program at the factory. The firmware charge customizes the GSI chip into a tailored device. A firmware charge may turn one GSI chip into a memory management processor and another chip into a language processor.

GSI chips may be used to build ring microcomputers, or they may be used in applications previously untouched by microcomputers. Since GSI expands the dollar-volume force, we should expect to see GSI in widespread use in the future. The author conjectures that GSI is the next step following the LSI age.

In the next section we examine alternatives to the problem of developing end-user applications through programming LSI and GSI computers.

3. Software

3.1 Problems

The dollar-volume force driving a technological advance ultimately owes its power to applications. In microcomputer technology, applications are realized only after considerable programming effort. Historically, programming effort has grown to the point where software cost is the economic determining factor.

In the previous section, we studied LSI software approaches to reducing software costs. In both cases, the cost is reduced after the software is produced. In this section, we study methods of reducing the implementation costs of first-time systems.

The problems associated with microcomputer programming stem from 1) the limitations of the architecture, 2) the transient period of bootstrapping from one machine to another machine, and 3) the problems that have always plagued programming.

The previous study of three typical microprocessors revealed features that facilitate assembly language programming. The use of index registers and a pushdown stack were noted as improvements over simple register transfer architectures. The sophisticated microprocessor example demonstrated how context switching and direct memory access to operands can ease the burden of system implementation.

Thus, the architecture of a microcomputer is fundamentally important to software development.

Once an architecture manifests itself in the form of a microcomputer, there is a time delay between hardware design and software design. The contemporary generation of microcomputers suffer from a lack of software. This shortage will continue until the transient period passes. Several temporary solutions are employed to overcome the software development transient.

A cross-translator is a program running on one machine that produces object code for another machine. The cross-translator runs on a parent computer and generates code to be executed on a child computer. The parent computer typically executes an assembler or high level language compiler written in a common language like FORTRAN. The output from the parent computer is loadable object code for the child.

A portable software package is a software package written in a language that is "easily" moved from machine to machine. The mobility of a portable software package may be due to its self-compiler feature or due to a collection of primitives that can be easily transported onto another machine.

In the case of self-compile portability, a cross-compiler is employed on the parent computer. The cross-compiler produces code for the child computer regardless of source input. Suppose the source input is the cross-compiler itself. Then the object code that results from self-compile is used to transport the cross-compiler onto the child computer. Once moved to the child, the cross-compiler becomes a stand-alone compiler and may be used in the same way that it was used on the parent computer.

A portability software package may also be written in a primitive portable language. The primitive portable language may actually consist of a set of macros whose expansion is determined by the child computer's architecture. A different prototype model is needed for each new child computer.

The primitive portable language may manifest itself as a hypothetical child machine. The hypothetical child instruction set is used to implement all portable software. When the software needs to be moved, a <u>transportation program</u> is written that maps each hypothetical child instruction into an equivalent actual child instruction (s).

Both approaches to portability are being used in contemporary microcomputer systems. The central problem hindering both approaches is code efficiency. Further work is needed to improve the object code resulting from transportation of software.

Ultimately, the problems that microcomputer programming faces are the general problems of software production.

The need for more "powerful" and expressive languages, for example. There are some indications that programming is about to make a grand departure from traditional procedural language techniques to other forms of man-machine communication. In the following sections, we examine alternate approaches to programming. In particular, we concentrate on forms of man-machine communication that fit well into the microcomputer dollar-volume force.

3.2 A System Implementation Language

The obvious approach to implementing software on a microcomputer is to use a high level language. The high level language should have several features of an assembler language, however, because the language is used to implement control programs, compilers, etc and requires the ability to access machine level resources. Such languages are called SIL's (system implementation languages).

Typically a SIL for a microcomputer executes as a cross-compiler. Although, this may be a transient mode of operation, the limited memory of many microcomputer systems prevent implementation of sophisticated SIL's. Often the resulting object code being produced is on the order of 16KB while the SIL translation may require 128KB of memory.

Figure 8. Example Of A Systems Implementation Language

```
MATCH:     PROCEDURE (PTR1, PTR2) BYTE  ;
  DELCARE  (PTR1, PTR2) BYTE  ;
  DECLARE  (STR1, BASED PTR1  ,
            STR2 BASED PTR2)    ADDRESS  ;
  DECLARE   I    ADDRESS  ;
  DECLARE  (J1, J2)  BYTE  ;

  J1, J2, I  =  0 ;

LOOP:  DO WHILE  J1 =  J2 ;
     IF  J1  =  OFFH THEN RETURN (0)  ;
     J1  =  STR1 (I)  ;
     J2  =  STR2 (I)  ;
     I   =  I  =  1  ;
  END LOOP:
  RETURN  (-1)  ;   Return (-1) when no match,. 0 when match.
END MATCH  ;
```

Figure 8 illustrates a SIL* for implementing software on the microcomputer of Figure 1. This program computes a zero if the two strings at location PTR1 and PTR2, respectively are equal. An 8-bit (-1) is returned as a hexadecimal OFF, otherwise.

Upon entry into PROCEDURE MATCH, the first string STR1 is located by pointer PTR1 and the second string STR2 is located by PTR2. This is indicated by the ADDRESS attribute that declares STR1, STR2, and I as symbolic labels for addresses in memory.

The BYTE sized pointers PTR1 and PTR2 contain the address of STR1 and STR2. Since they are passed by value, the MATCH routine is useful for comparing any two strings at location specified by PTR1 and PTR2.

In the sample program, each character of the two strings is moved to J1 and J2, respectively. J1 and J2 are compared and as long as they are equal, the next byte pair is compared. The code OFFH is used to indicate that the end of the string STR1 has been reached. In this case, the strings are equal and a zero is assigned to location MATCH.

The LOOP segment of the demonstration program repeats as long as the character in J1 matches the character in J2. When the value of J1 = OFF hexadecimal, the last character of the string has been reached.

* The SIL in this example is a version of PL/M for the Intell 8080 system.

Each byte of STR1 is copied into J1 and each byte of STR2 is copied into J2. This is done by indexing STR1 and STR2 by I. The index value stored at location I is incremented and the loop executed again unless J1 does not match J2.

This program is compiled into machine language for the microprocessor of Figure 1. Since the microcomputer is an 8-bit architecture and we know that considerable effort is required to overcome its limitations, this language greatly improves the prospects for programming the machine. The language "covers-up" the limited architecture and yet allows a programmer access to data bytes and addresses. Perhaps the greatest improvement is that the SIL provides indexing and addressing capability lacking in the machine itself.

The SIL approach is an outgrowth of language development on large machines. Since microcomputers are revolutionizing the way we think about computing, perhaps it is also time to question the SIL approach. Are there better ways to program extremely low-cost hardware without paying dearly for software?

3.3. Pushbutton Programming

One of the startling revelations of the LSI hardware era was the significance of pocket calculators. Pocket calculators are partially successful because of their simple man-machine interface. Their interface eliminates the traditional operating system, language processor, utilities, and computer terminology and replaces them with the finger. A pocket calculator is programmed by push-button.

Pushbutton programming can be elegant and sophisticated in spite of its simplicity. Elegance is usually achieved in one of two ways, 1) identifying primitive "button" operations for a given application, or 2) building primitive "button" operations on top of other primitives in a hierarchy of modules.

Primitive button operations are implemented in LSI software modules or as software programs. The LSI software module approach is based on firmware encapsulation of accepted standards. We discussed the encapsulation process for a ring structured microcomputer, earlier.

The software program approach typically represents an experimental or intermediate step in developing a truly pushbutton microcomputer system. Once the function represented by each "button" is known to be primitive to the application, the software program for the function should be encapsulated as an LSI software module. This has been done, for example, with BASIC interpreters and I/O controllers.

Figure 9. Sample Pushbutton Program For Business Primitives

```
ACCOUNT        FILE
NUM            FORM  8
NO             FORM  8
INDEX          FORM  8
NAME           DIM   40
BALANCE        FORM  5.2
LENGTH         FORM  "3997"
               OPEN  ACCOUNT, "LOOK UP"
START          DISPLAY "ENTER ACCOUNT NUMBER"
               KEYIN   *N, "ACCOUNT?", NUM
               MOVE    NUM TO INDEX
MOD            SUBTRACT LENGTH FROM INDEX
               COMPARE LENGTH TO INDEX
               GOTO MOD IF LESS
LOOK           READ ACCOUNT, INDEX; NO, NAME, BALANCE
               GOTO ERROR IF OVER
               COMPARE NO TO NUM
               GOTO LOOK IF NOT EQUAL
               DISPLAY NAME, "HAS BALANCE=", BALANCE
               GOTO START
ERROR          DISPLAY  "NO SUCH ACCOUNT IN FILE"
               GOTO START
               STOP
```

Figure 9 demonstrates a pushbutton program for a pushbutton microcomputer.* The microcomputer is assumed to consist of a CRT/Keyboard, microprocessor and memory, and a diskette mass storage device for file storage.

The program of Figure 9 assumes a file containing names and balances. Upon entry of a name, a balance is retrieved and output to the CRT console.

The FILE button establishes a file named ACCOUNT. The FORM buttons declare (NUM, NO, INDEX) as numbers requiring 8-digit accuracy. The DIM button reserves space for a 40-character string. The BALANCE number is a dollar and cents figure with up to 5 digits for the dollar amount and 2 digits for the cents amount.

The ACCOUNT file is OPENed for "look-up". The DISPLAY and KEYIN buttons perform I/O via the CRT/Keyboard. The account number NUM is moved into variable INDEX where it is reduced modulo LENGTH (notice that LENGTH is 3997). The remainder produced by the MOD segment of code is used to index into the ACCOUNT file.

The LOOK segment searches the ACCOUNT file by directly indexing into the ACCOUNT file. If NO, NAME, and BALANCE are not the desired matching record, then the file is searched sequentially until the matching records are found. If no matching records are found, then the search terminates with a DISPLAY message at ERROR.

* The language is DATABUS which is used on Datapoint computers, see reference (4).

Each time a READ is executed, the value of INDEX is incremented to the next record in the file. Thus, each time through the LOOK loop another record is retrieved from the diskette file.

The pushbutton microcomputer illustrates how programming is simplified for business data processing applications. The primitives are data processing primitives as opposed to mathematical, word processing, or graphical primitives.

The disadvantage of the type of microcomputer system shown in figure 9 is that the system is limited. The primitives are fixed, and although sufficient for the intended novice user, they cannot be combined into subprocedures, "superbuttons", or extended by adding other functions. The next section illustrates a more sophisticated button pushing language that overcomes these limitations.

3.4 Improved Pushbutton Programming

It is desirable to have a powerful pushbutton language that is simple and easy to use. Simplicity and power do not always go hand-in-hand, however. How can we reach a compromise between the two within the limits of microcomputer based systems?

Suppose a primitive set of "buttons" are used to build more sophisticated structures through modular construction of "superbuttons". A superbutton is a procedure that invokes many lower level buttons. It is the concept of a subprocedure as applied to pushbutton programming.

Extension through superbutton programming requires a table mechanism to manage the names of the buttons. A dictionary and interpreter are needed to process the superbutton primitives.

The dictionary contains the name of each button and a pointer to a code segment. The code segment is a chain of other pushbuttons (all of which are contained in the dictionary) or a segment of microcomputer executable machine code.

Since each button could possibly have one or more parameters passed to it or generated for it by another button, a parameter passing mechanism is needed. Thus, a pushdown stack processor is used to execute the superbuttons and process their parameters.

The interpreter performs dictionary look-up and manages the pushdown stack. Obviously, since the interpreter is nothing more than a program, it too can be written in the pushbutton language. In fact, the interpreter is an example of a superbutton, see below.

```
:INTERPRET       BEGIN
                     WORD FIND IF EXECUTE
                     ELSE NUMBER
                     THEN QUERY
                 END ;
```

The : denotes that this is a superbutton named INTERPRET. The chain of buttons to follow define what it means to push INTERPRET. Since the interpreter runs forever, the BEGIN-END pair brackets a never-ending loop.

WORD extracts the name of a button from the input device (we assume a microcomputer like the one in the previous section). FIND searches the dictionary and returns TRUE if the name previously input matches an entry in the dictionary.

The pushdown stack maintained by the superbutton processor contains either a TRUE or FALSE after FIND is performed. The IF is performed if the stack contains a TRUE. The ELSE clause is executed if a FALSE appears on the stack. Suppose the TRUE condition results, then the EXECUTE button performs the function indicated by the button found in the dictionary.

If the FALSE condition results, then NUMBER is executed. This button attempts to convert the input name to a binary number. Failure aborts the execution of a user's button stream. The interpreter expects either valid names for buttons or valid numbers as input.

THEN marks the end of the IF-ELSE clauses. Control returns to QUERY in either TRUE or FALSE cases. The QUERY button puts the interpreter in idle mode until more input is available.

The dictionary and interpreter combine to give a user powerful, yet simple access to increasingly complex structures. Extensibility results from building superbuttons on top of relatively low level primitives.

A simple pocket calculator example shows how the INTERPRET button processes an expression.

```
12  50   *   10   /    .
```

The 12 and 50 are pushed onto the stack as they are input. This happens because the FIND button returned a FALSE condition (12 and 50 do not occur in the dictionary). The FIND button does locate on * in the dictionary, though, and the result is that EXECUTE performs a multiply. The result (600) is placed back on the stack and QUERY waits for another input. The 10 is pushed onto the stack, and the / is EXECUTED, leaving a 60 on the stack. The period causes the 60 to be printed out. The interpreter idles.

As a final example, suppose we want a superbutton to compute absolute value. Assuming that ABS is not a primitive button already, we could add it to the dictionary merely by defining it with an : control character.

```
: ABS
     DUP
     0 <
     IF    MINUS
     THEN   ;
```

This code strings together a chain of buttons to perform sign reversal when desired. DUP produces a duplicate on the stack. This duplicate is absorbed by the 0 $<$ test button that sets a TRUE or FALSE condition on the stack. If a TRUE condition exists, then MINUS performs a sign reversal, and replaces the number on the stack, otherwise nothing is done to the number originally on the stack.

Since the superbuttons are constructed from primitives, the problem of portability is partially solved. Each button is defined in terms of a particular microcomputer machine language. A package of superbuttons for a special application can be moved from one microcomputer to another by rewriting only the basic primitives*. These primitives occur in the dictionary, so the actual re-coding is done by changing the code segment referenced by each dictionary entry.

In summary, we can say that SIL's and pushbutton languages both strive to cover-up the limited architectures underlying microcomputer design. The user sees only a symbolic manifestation of the microcomputer.

In the transient period between the large machine era and the LSI era, we should expect a re-examination of the problems and solutions of the past. Pushbutton programming has no precedent in earlier systems because of the easy access by novice users. In the next section,

* This is the approach taken by FORTH, Inc, see reference (3).

we study some of the trends brought on by LSI hardware and software.

4. What Computing Has Come To.

4.1 How Large Should a Computer Be?

The microcomputer invasion is bringing an end to the Renaissance Computer* Age. The reasons for this are both technological and economical.

Hardware costs have, because of LSI technology, diminished below the cost of complexity making general purpose k-way shared systems uneconomical for large values of k. On the otherhand, software development costs remain high due to complexity. Therefore, software complexity is forcing duplication of integrated hard/soft systems in place of hardware systems running a variety of programs.

The hardware shift, as it is called, is also responsible for a shift in the type and number of computer applications. Shifts in applications lead eventually to greater hardware shifts. Viewed from an economic point-of-view, the hardware shift is an "acceleration force" whose rate of change determines the size of future computers.

The first 3 computer generations were charaterized

* A Renaissance Computer is a general purpose, large, central computer. Its purpose is to do all things. Its size and cost are justified by its multipurpose, multiprogrammed, and often timeshared operation.

by cost and physical size. A typical computer installation consisted of millions of dollars worth of hardware and required massive support in terms of air conditioning, tape libraries, programmers and administrative personnel.

These large, costly computers quickly became Renaissance Computers or what IBM popularized as General Purpose computers. A Renaissance Computer is capable of doing a variety of things: business data processing, scientific calculations, telecommunications, word processing, information storage and retrieval, etc. Actually, however, it was only the very expensive processing problems that were attacked by Renaissance Computers. That is, space age calculations, business for large corporations, and information storage and retrieval for large private universities that could afford to experiment. Small scale computing was a very expensive hobby carried out mostly by aerospace engineers who bootlegged time on the company's Renaissance Computer to simulate Las Vegas games of chance, or academic people who experimented under the name of artificial intelligence or CAI.

There were valid reasons for the Renaissance Computer. Any computer was expensive to fabricate and maintain and so <u>had to be multipurpose</u>. The Renaissance Computer, because of its cost, was an affordable machine only for those with a variety of uses in mind.

The emergence of minicomputers heralded the end of the Renaissance Computer Age. LSI technology has greatly accelerated the coming of the end by decreasing hardware costs to the point where cpu costs were negligible. Indeed, the only obstacles remaining for "computing for the millions" is the cost of peripherals and the amount of effort required in developing software.

The mini/microcomputer provides a hardware basis for the emergence of the Common Computer Age. This age is characterized by inexpensive hardware, novel I/O devices, inexperienced users/programmers, and expanding market and applications, and reorientation of the economics of computing. As in the Renaissance Computer Age, the new age will be governed by economic forces more than technical forces (even though LSI technology brought about the revolution).

The logic of the economic force behind the Common Computer Age goes as follows: The cost of computing is controlled by the number and kind of applications. The number and kind of applications are determined by the cost of computing. Thus, a feedback loop is completed. The delay in this loop is speculated to be 3-5 years, but decreasing with each computer generation.

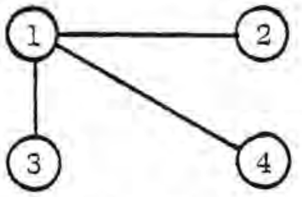
The topics of 1) novel I/O devices, 2) inexperienced users/programmers, and 3) applications are not central to the issues addressed here, but suffice it to note that TV/keyboard devices are on the increase, BASIC as a programming language is rampant, and computer games are in tremendous demand. The reader can easily make predictions based upon these trends.

4.2 The Cost of Complexity

A general system is a collection of interacting parts, each part having well defined features. An understanding of these features does <u>not</u> guarantee an equal understanding of the general system. Indeed, a system often behaves in unexpected ways even after careful study of its parts. Unexpected behavior is frequently observed in computer systems, much to the chagrin of programmers, hardware designers, and users.

A simple model of complexity may be applied to computer systems to determine optimal degree of sharing of hardware, optimal degree of sharing in software design, and to make conjectures about the <u>best</u> size for a "computer".

Suppose a system is made of 4 parts as shown below.



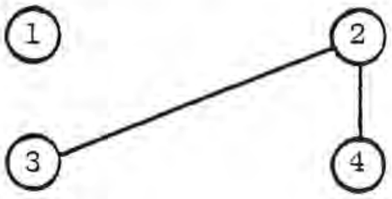
3 connections

The first part is allowed to interact in some way with the other 3 parts, also shown above. "Interaction" is a generalized concept. It may refer to communication, a physical connection, an effect, or some other tangible or intangible connection.

Let us define complexity and its corresponding "cost" as follows:

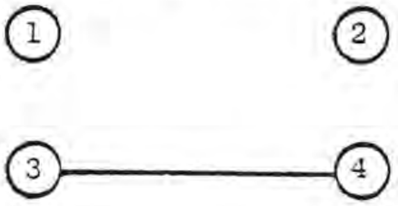
$C_n = C_0$ (the potential maximum number of interactions possible in system of n parts)

We can compute the potential maximum number of interactions possible in a system of 4 parts by completing all of the connections in the 4-part system, above.



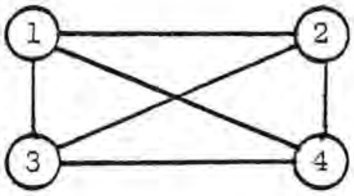
2 connections

The remaining number of connections between the second part and all other parts is 2. The remaining number of connections from part 3 is shown below.



1 connection

The composite of all of the above shows that in a system of n parts, there are $(n-1) + (n-2) + \ldots 1$ connections.



$$\sum_{i=1}^{n-1} i \quad \text{connections}$$

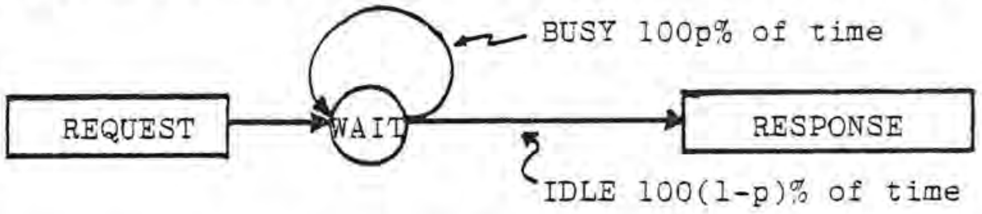
Thus, the cost of complexity in an n-part general system is proportional to the sum of the first (n-1) integers.

$$C_n = C_0 \frac{n(n-1)}{2}$$

4.3 Large-Scale Versus Micro Hardware

The Renaissance Computer was, and is, made affordable by time-multiplexing the hardware. This is done in a variety of ways, all falling under the misnomer of "timesharing" or "multiprogramming". Actually, what goes on inside of a multiplexed Renaissance Computer is a division of cpu power into k parts by a k-way multiplexing scheme, The purpose of the k-way division is to keep the expensive cpu busy in order to spread its cost k ways.

Extensive sharing is a modern day fallacy for two reasons: 1) the cpu is no longer the most expensive part of a system, and 2) the ability of the cpu to render service increases as the cpu becomes idle. This is demonstrated by the simple Markov model of a request for service, below.



The request enters a WAIT state that may or may not hold the request for W units of time, say, and then when the cpu is idle, the response is given in R units of time. The average delay is given by the simple formula, below.

$$\text{Avg. Response Time} = R + \frac{P}{(1-P)} W$$

Examination of a plot of Avg. Response Time versus busy time p shows that the smaller p (more idle time) the better is the expected response.

In light of the Common Computer Age, the rule of multiuser cpu design should be to keep the cpu idle as much as possible. This can be done by increasing the cpu speed so that every request takes zero time (R = 0), thus freeing the cpu. Alternatively, we can decrease R by increasing the number of cpu's. Hence, R is decreased, and so is p, by incorporating multiple copies of cpu's.

Let us look now, at the cost of a k-way shared computer. The cost is conjectured to be the sum of the single-unit (k=1) system plus the cost of k-way complexity.

$$H_k = h_0 + h_1 \frac{k(k-1)}{2}$$

where h_0 = cost of a single system

and h_1 = cost of each additional unit needed to provide shared service

The value of h_1 includes the cost of the added complexity in hardware and software (reflected in main memory size) needed to share the basic hardware. This includes protection and addressing mechanisms, communications equipment, large central stores, scheduling algorithms, etc.

This model may seem pessimistic at first, but when compared to other "laws of complexity" is actually rather generous*. This cost is even more generous when distributed over all k of the parts.

$$H_k/k = h_0/k + h_1 (k-1)/2$$

The corresponding cost function for non-shared hardware/software systems is obtained when k=1.

$$H_1 = h_0$$

A collection of k non-shared "mini" systems would cost kH_1. When is it cheaper to use kH_1 systems in place of one H_k system?

Set $H_1 = H_k/k$ and solve for k. This produces the quadratic formula:

$$k^2 - (1 + \frac{2h_0}{h_1}) k + \frac{2h_0}{h_1} = 0 \qquad [A]$$

with solution:

$$k = \frac{2h_0}{h_1}$$

*Grosh's Law states that doubling the cost of a system can only be justified if its performance is quadrupled. Why? Minsky's conjecture claims $\log_2 k$ utility in a k-way parallel system. Thus, we are encouraged to speculate that k-way redundancy will cost somewhere between $(h_1 k^2)$ and $(h_1 2^k)$.

When h_0 and h_1 are known, formula A gives the optimal k-way sharing strategy for a Renaissance Computer. In the case $h_0 \gg h_1$ (expensive hardware) the result is that $k \gg 2$. Hence, multiplexing the hardware is indeed a valid strategy.

In the case $h_0 \ll h_1$ (cheap cpu hardware) the optimal strategy is to limit sharing, $k \ll 2$. If more than 2 users are to share the same cpu, we are advised to duplicate the basic system instead of multiplexing it.

A _balanced system_ is one in which $k = 2$. Thus, when $h_0 = h_1$, we see that there are advantages to foreground-background processing. It is only fair to note, however, that the cost of sharing, h_1, is also declining as memory, communications, and programming techniques decline in cost. In summary, it appears to be wiser to expect shared systems for small values of k in the future. The age of larger scale k-way Renaissance systems has passed*.

*Large-scale special purpose systems _are_ expected, as long as a narrow objective is kept in mind. The airlines reservation systems, credit check systems, etc. are examples. These systems minimize complexity by trading-off vast objectives, and do not represent Renaissance Computer systems.

4.4 Large-Scale Versus Micro Software

Large-scale hardware systems imply large scale software efforts. The exception, of course, is when the band of applications is narrow or the system is designed for a special purpose. The software effort expended on Renaissance Computers is documented elsewhere and need not be repeated here. Instead we seek to determine possible boundaries on software effort regardless of hardware limits.

Brooks [1] reports that the effort needed to develop M instructions of software is proportional to $M^{1.5}$. If we divide the M instructions into n optimal-sized modules, we can prove that

$$S_n = C_0 \ (M/\sqrt{n})$$

This is sketched for the reader as follows:

Given, $S_n = C_0 \ M^{1.5}$

Let $S_i = C_0 \ M_i^{1.5}$, and $S_n = \sum_{i=1}^{n} S_i$

$$M = \sum_{i-1}^{n} M_i$$

The object function F is minimized:

$$F = C_0 \sum_{i=1}^{n} M_i^{1.5} - \lambda \left[M - \sum_{i=1}^{n} M_i \right]$$

$$\frac{dF}{dM_j} = 0; \text{ yields } M_j = M/n$$

and substitution produces S_n.

A software project that is large enough to be broken-up into n parts also suffers from a loss proportional to the complexity of an n-part system. The cost function for S_n must be amended to show this.

$$S_n = C_0 \frac{M}{\sqrt{n}} + C_1 \frac{n(n-1)}{2} \qquad [B]$$

C_0 = man-months effort per instruction

C_1 = man-months effort per interaction

The parameters of formula B depend upon vague quantities like "human communication" and "type" of application. Brooks [1] indicates that $C_0 \doteq 0.001$ for operating systems programming, while $C_0 \doteq 0.01$ for applications programming. In general, very little is known about the behavior of C_0 or C_1.

A plot of S_n versus n reveals an optimal value of n, see Figure 10. This point gives the smallest investment needed to successfully complete the software.

Minimization of S_n gives the formula for software size as a function of the number of software parts.

$$M = \frac{C_1}{C_0} n^{3/2} (2n-1)$$

The inverse of this function is plotted in Figure 11. It shows that even for small software projects, the number of parts should be relatively large (8 to 12), unless C_1/C_0 is extremely large. In short, programming is costly even though the software project is relatively small. There is an "economy of scale" possible, however, because large scale software projects diminish in cost as their size grows, <u>if subdivided into the proper number of parts and</u> C_1/C_0 <u>is large enough</u>.

4.5 Summary

It is clear that a shift in hardware costs is causing revolution in computing. In the past, a single piece of hardware employed a variety of software to solve a (limited) variety of problems. In the future, a (limited) variety of hardware pieces will employ a single piece of software to solve a variety of problems. The most dramatic contemporary example of this Common Computer Age fact is the pocket calculator. The pocket calculator market is built from the notion that a variety of hardware pieces can be applied to a single software piece.

A subtle example of the effects of the hardware shift is found in the many "turnkey" minicomputer systems designed for business data processing. Duplicate hardware systems are married to a single copy of software. The software is

Figure 10. Software Effort versus n. The * marks the minimum point for S_n.

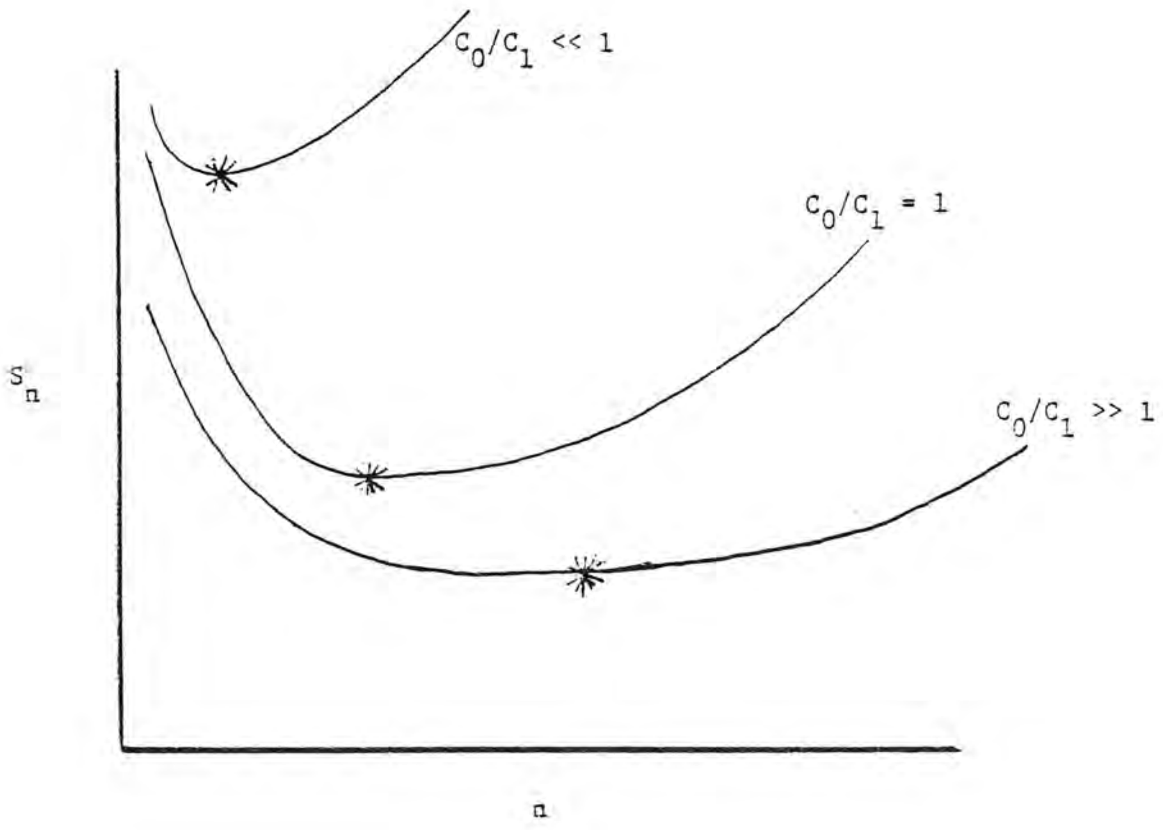
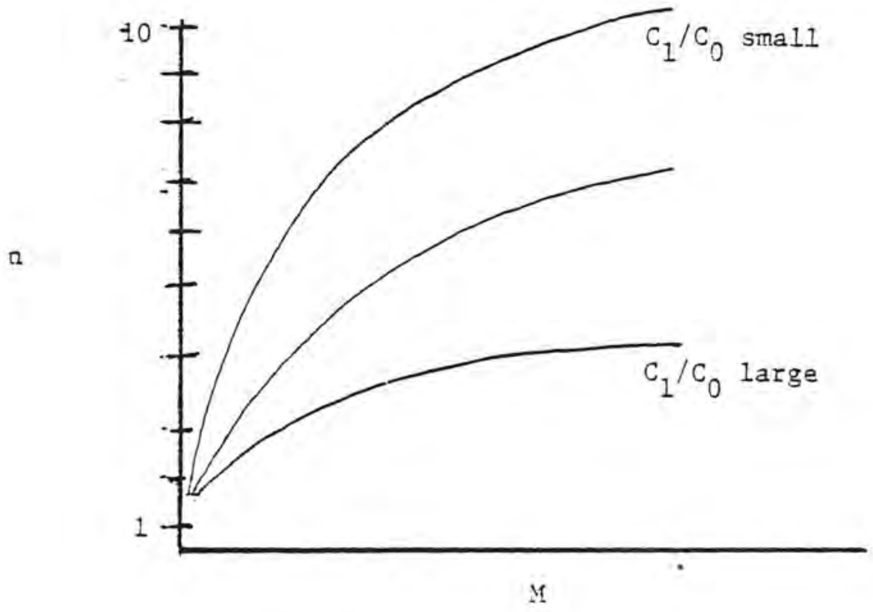


Figure 11 n versus M for software development



packaged for "vertical" lines of applications, e.g., payroll, accounts receivable/payable, etc. These packages are called vertical because they cut across many industries with small changes in parameters. These systems also demonstrate the principle of limited k-way sharing, because they are restricted to $k < 16$ in most cases.

Application of the hardware shift to larger systems leads us to believe that either 1) distributed network of microcomputers, or 2) integrated network of microcomputers are advisable. The distributed network consists of isolated cpu's each with access to a common mass storage unit (s). This provides a way of limiting the local complexity by spreading it over several levels of the network hierarchy.

The integrated network approach consists of a central dispatching unit, cdu, and access to/from special purpose "organs". The organ computers are actually special purpose computers akin to the controllers of current Renaissance computers. The integrated network system copes with complexity by compartmentalizing it inside each special-purpose organ. Before the total system can be made to operate efficiently, however, some means of intercommunication must be devised so that large-scale breakdowns can be avoided. This problem has not been solved, but the ring structured microcomputer represents an approach to coping with network complexity.

In answer to the question posed by the title of this section, we must say that a computer should be large enough to support a limited application and a small number of users. Additional applications and number of users justifies additional systems rather than additional complexity of a single system. The trend should be toward dedicated microcomputer systems with large memories and about 2 users. The number of users may be increased, but the application must then be narrowed to compensate for the added complexity.

Acknowledgements

The author gratefully appreciates the painstaking review and comments of Ed Towster. His suggestions improved the quality and style of this chapter.

REFERENCES

(1) Brooks, F. P., The Mythical Man-Month, DATAMATION, 20, 12 (Dec. 1974), p. 44-52.

(2) Lewis, T. G., How Large Should A Computer Be? ACM SIGMINI NEWSLETTER, Vol. 2, No. 1, 1976.

(3) Rather, E. D., and Moore, C. H., Minicomputer programming is FORTH, personal communication, 1976. (FORTH, Inc., Manhattan Beach, California).

(4) Datashar 3.1 User's Guide, Datapoint Corp. San Antonio, Texas, 1975.

(5) 990 Computer Family Systems Handbook, #945250-9701, Texas Instruments, Inc., Austin, Texas, 78767.

(6) Gorman, W. and Broussard, M., Minicomputer Programming Languages, Proc. ACM SIGMINI/SIGPLAN Interface Meeting On Programming Systems in the Small Processor Environment. March 4-6, 1976. p.4-15.

(7) Intel 8080 Programmer's Reference Manual, Intel Corp. Santa Clara, California, 1974.

(8) Motorola 6800 Applications Handbook, Motorola Corp., Phoenix, Arizona, 1974.

EVOLVING MINICOMPUTER ARCHITECTURE

T. G. Lewis
Associate Professor
Oregon State University
Corvallis, OR 97331
(503) 754-3278

Prepared for Minicomputer Systems Report, INFOTECH International Ltd., Nicholson House, Maidenhead, Berkshire SL6 1LD, England

May 1976

I. EVOLUTION OF MINICOMPUTERS (1,2,3,4)

The terms maxi, midi, mini, and micro recently appeared in the computing literature. While it is usually clear to the informed what a minicomputer is and what a maxicomputer is, there have been few precise definitions of either. A working definition offered by Lewis (4) proposes that a minicomputer is a hypothetical computer designed with a minicomputer attitude in mind. Thus, the discussion of minicomputing centers on attitude about limited, special purpose computing instead of concentrating on a description of a representative machine.

The minicomputing attitude started in the mid nineteen-sixties with the introduction of the Digital Equipment Corp's PDP-8 computer. It is a 12-bit/word minicomputer with limited instruction set, small memory, and a low price tag. The first mini was designed for limited applications, and yet it has become one of the most prolific architectures ever designed. This mini, in it's many reincarnations, sold over 40,000 units in its first ten years of production.

At the turn of the last decade, over 50 companies were marketing minicomputers. The lower cost of limited architecture machines was more important to a user than the fact that the architecture delivered limited performance. Consequently, new applications opened up and the demand for more minicomputers accelerated their development.

A revolution in electronic technology added impetus to an already rapidly evolving minicomputer industry. Large scale integration, LSI, lowered the cost of cpu hardware to the point where basic philosophies of computing are being questioned. For example, the maxicomputing attitude of sharing a central processor may be threatened in light of the trend toward "free" central processors.

A Renaissance Computer is the term used to describe a large, general purpose, shared computer system (4). Minicomputer attitudes are in conflict with the Renaissance Computer attitude. The future of computing depends upon the outcome of this conflict. The philosophy of sharing, as it is currently practiced by Renaissance Computer systems, may be misplaced philosophy.

Perhaps this question and others being re-examined by the minicomputer advocates can be answered by looking at evolving minicomputer systems as they have unfolded in recent designs. Basically, these systems incorporate features derived from the need to overcome limitations in past mini architectures. What are these limitations?

The low-cost of minis has led to an expanding market. These new applications require special purpose solutions, and as a result there is an increasing need for software aids. In response, a flourish of activity in languages and operating systems for minis has produced a variety of novel systems. In short, the demands of an end-user market have led to an:

1) expanding market/applications, and
2) more software.

These top-level requirements eventually find their way into the design phase of new systems. Ultimately, the architecture of new systems must support these new requirements.

Currently, the "power" of a typical minicomputer architecture is limited because of:

1) small address spaces resulting in small memory,
2) weak run-time support of high level languages, and
3) limited operating systems, file structures, and communications support.

In addition, it is clear that the same technology that reduced the cost of central processing units must be applied to the construction of peripherals and memory before corresponding reductions in overall system cost are realized.

The move toward architectures that support user requirements and the ever increasing need for peripherals and memory indicate that minicomputers of the future will continue the trends toward:

1) low-cost peripherals,
2) larger memories, and
3) architectural extension through micro-programming.

The last trend above indicates continued interest in firmware development. Indeed, it appears that "firmware sets" in the form of add-on ROM (read-only-memory) are becoming common place. For example, sort packages, scientific subroutine packages, and text editors are offered by several manufactures as ROM firmware extensions to basic systems.

The evolution of minicomputer architecture can be characterized in a variety of ways. The approach taken in this presentation is to concentrate on two fundamental limitations: addressability and run-time support mechanisms. These two basic properties of computer architecture have far reaching implications in terms of minicomputer processor "power".

After establishing a formalism for describing addressability and the run-time "environment", four representative systems are used to illustrate the evolution of minicomputer architectures.

II. EVOLUTION OF ADDRESSABILITY IN MINI ARCHITECTURES

The addressability, A, of a computer architecture is said to be the total number of memory cells accessible by a "typical" instruction defined within the architecture. Clearly, it is desirable to be able to address all memory locations in main memory. On a 16-bit mini, this usually sets a limit on A of 2^{16} words or bytes.

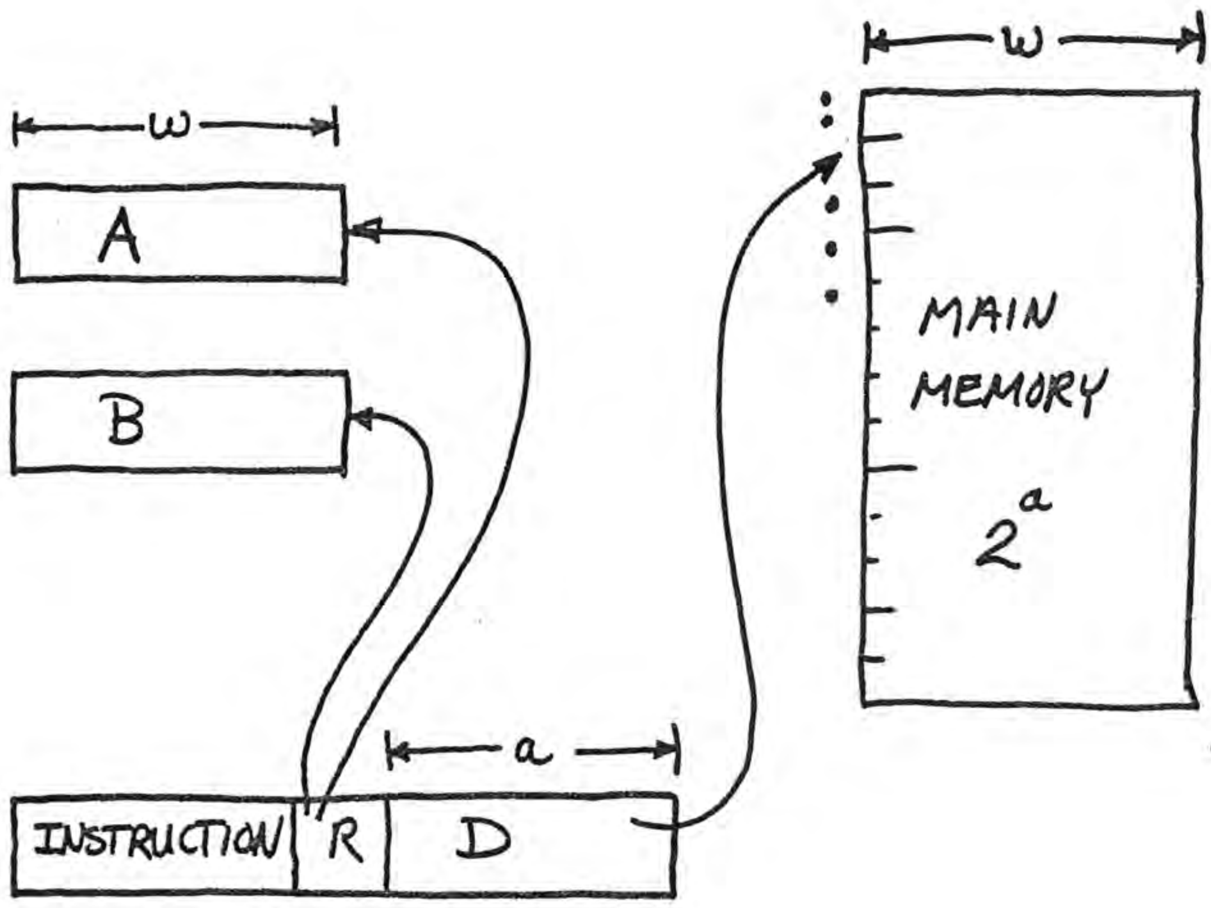
A "typical instruction" is defined loosely as any instruction requiring two operands. Thus, add, move, and exclusive-or are considered typical while branch, and shift are considered atypical instructions.

The addressability of a two-operand instruction is the cross product set of all locations potentially containing operands. The cross product set of the special register architecture of Figure 1 consists of the ordered pairs obtained from register A and each of the 2^a memory cells plus the set obtained from register B and each of the 2^a memory cells. The size of the cross product set of accessible locations is used as a measure of addressability:

$$\begin{aligned} A_{SR} &= \text{(number of registers)} \times \\ &\quad \text{(number of memory cells)} \\ &= (2)\ (2^a) \\ &= 2^{a+1} \end{aligned}$$

The value of A_{SR} depends on the number of bits (a) dedicated to the direct address of an operand. Suppose a particular mini implemented an ADD instruction in 16 bits, where a=10 bits. Then the two-register SR architecture of Figure 1 would have addressability $A_{SR} = 2048$.

FIGURE 1. ADDRESSABILITY OF AN SR ARCHITECTURE WITH TWO WORKING REGISTERS.



A, B	Working Registers
R	Register Designation
D	Direct Address

The addressability of an SR architecture is severely limited. Typically the SR design is modified by adding an index register. This results in an SRX architecture with greater addressability, A_{SRX}.

$$A_{SRX} = (2) \ (2^a) \ (2^w) = 2^{a+w=1}$$

When an index word of length w bits is included, the addressability of the two-register SR architecture of Figure 1 is greatly increased. For example, when w=16 bits, a=10 bits, then A_{SRX} = (2048) (65k) = 130K.

Minicomputer architectures rapidly evolved to multiple, general purpose register architectures for a variety of reasons.

General purpose register machines typically are able to access data in working registers through index registers, and by way of return address registers. The addressability of GR architectures shown conceptually in Figure 2 is even greater than special purpose index register organizations. When oerands are stored in the GP registers, the value of A_{GR} is n times that of a single register SR architecture.

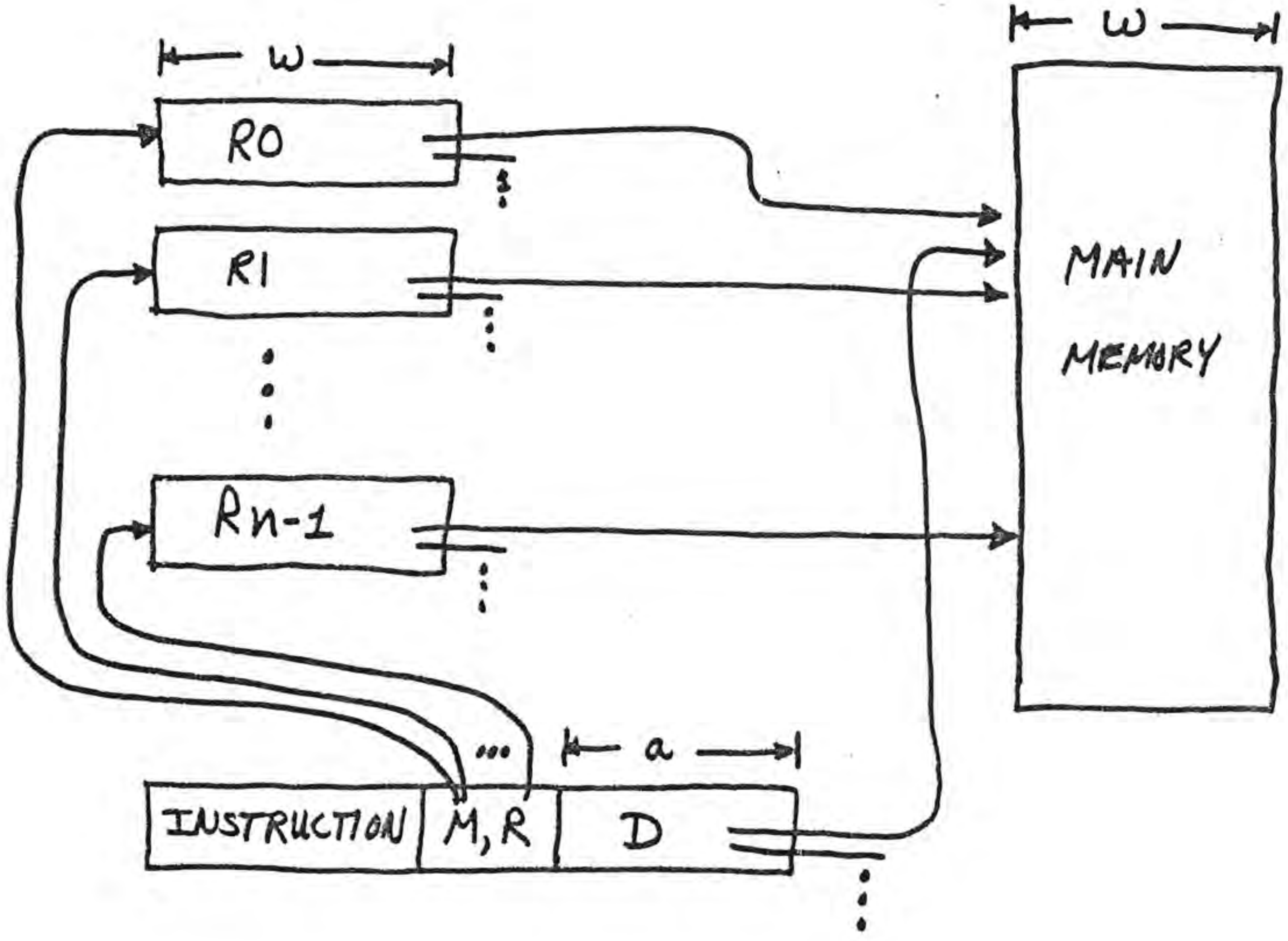
$$A_{GR} \ (\text{operand}) = n \ (2^a) = n2^a$$

For example, when w=16, a=10, n=8, the operand addressability of Figure 2 is 2^{18}.

When operands are stored in main memory, but accessed via the index mode, the addressability of the n-register GR architecture is 2^w times greater.

$$A_{GR} \ (\text{index}) = n \ (2^w) \ (2^a) = n2^{w+a}$$

FIGURE 2. ADDRESSABILITY OF GR ARCHITECTURE
WITH INDEX MODE OF ADDRESSING



M Mode of Addressing

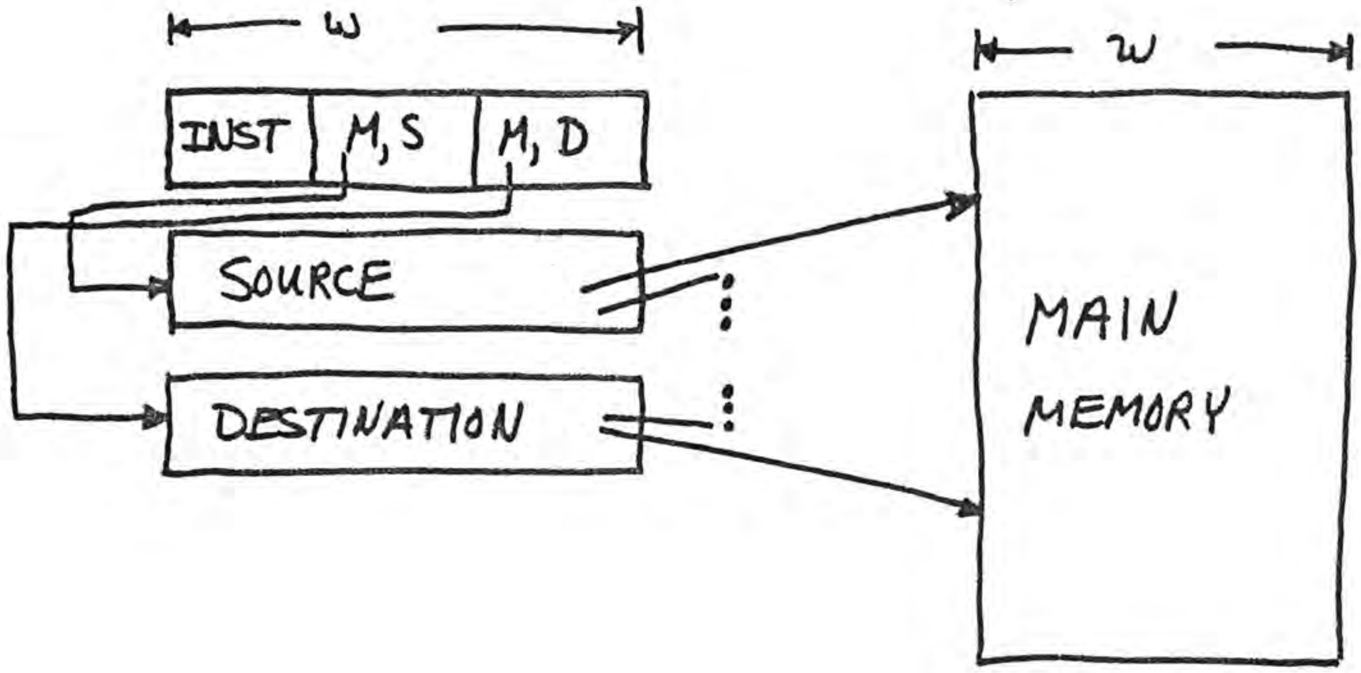
Finally, minicomputer architectures have evolved in two directions beyond the classical register transfer organization. The two-address organization employes a variable word instruction format and relative or direct address modes to improve addressability. Figure 3 illustrates the TA (two-address) feature of contemporary mini architectures.

The value of A_{TA} is simply the size of the cross-product set produced by the two pointers in the 3-word instruction.

$$A_{TA} = (2^{W})\ (2^{W}) = 2^{2W}$$

When W=16 bits, this yields and addressability of 4225K. This dramatic increase in addressability is costly, though, because the instruction occupies more program space.

FIGURE 3. ADDRESSABILITY OF TA ARCHITECTURE WITH 3-WORD INSTRUCTION FORMAT.

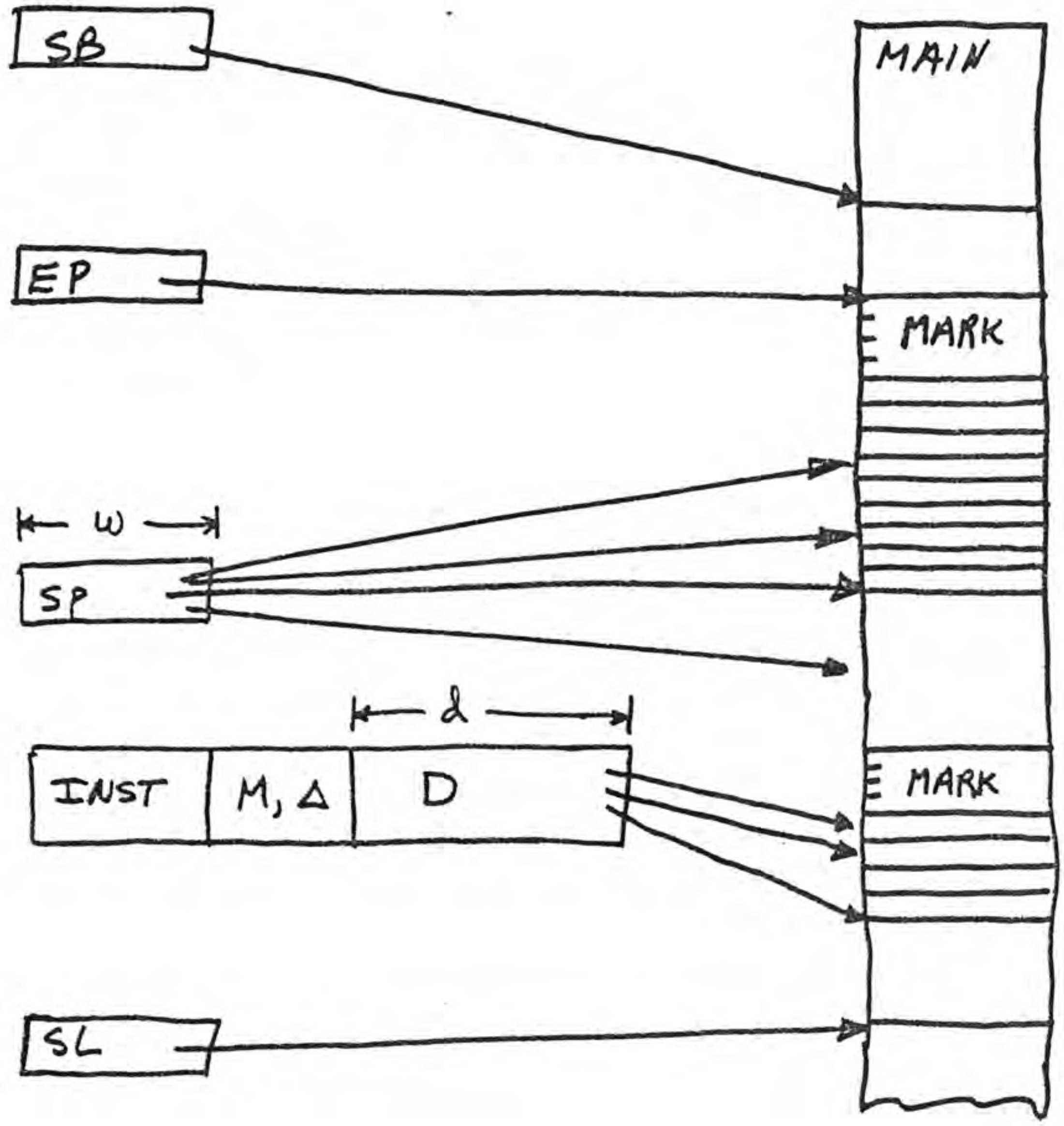


The second direction taken by minicomputer designs was motivated not by addressability, but instead by requirements for run-time support of high level languages. The SA stack architecture of Figure 4 purposely restricts addressability to gain control over a name space called the environment. We discuss the impact of environment upon architectures in the next section.

The SA addressability of Figure 4 is limited by either the stack limit register, SL, or by the width of the stack pointer SP, plus the displacement field in a "typical" stack instruction.

$$A_{SA} = \text{Min} \left\{ (SL-SB+1)^2, \quad 2^{w+d} \right\}$$

FIGURE 4. ADDRESSABILITY OF SA ARCHITECTURE WITH ENVIRONMENT CONTROL



The architecture of Figure 4 is designed around the notion of an environment. The local environment of data is established by a special cell called the MARK. A set of pointers establish the location of one or more MARKs. Addressing is relative to the MARK, stack base SB, or stack pointer SP.

The data environment of Figure 4 is limited by SB and stack limit register SL. Thus, (SB-SL+1) is the size of the set of accessible data cells.

Furthermore, depending upon the value of d in the instruction format, the addressability may be limited greater than indicated by the value stored in SB and SL. For example, if w=16, d=3, and SB = 0, SL = 65K, then $A_{SA} = \text{Min} \left\{ 4225K, 512K \right\} = 512K$.

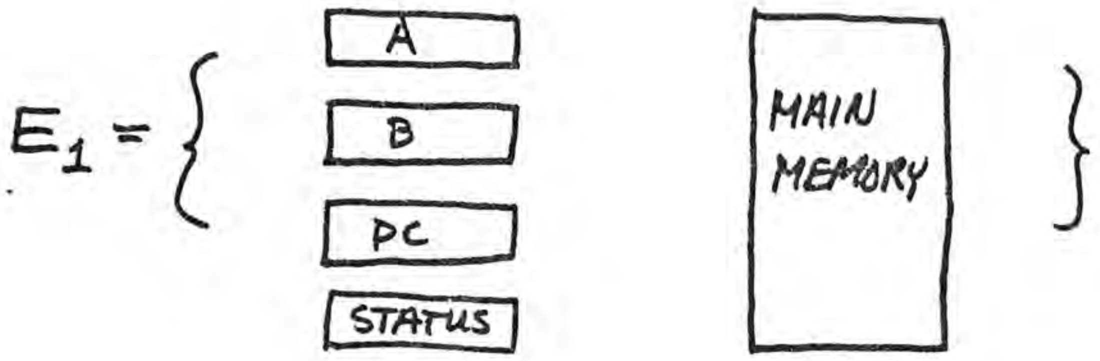
The SA architecture evolved expressly for the purpose of controlling high level language environments. What these environments are, and how they influence architectural trends is discussed next.

III. THE EVOLVING E-SWITCH POTENTIAL IN MINI ARCHITECTURES

An environment is established in an active program and its corresponding data. The activation of a program is called a process. Thus, the environment of a process is the set of resources accessible to the process. Often the process runs in a nested environment as in the case of recursive execution of code, or in the case of block-structured run-time support for block-structured languages.

An example of a single process environment is shown in Figure 5.

FIGURE 5. SINGLE PROCESS ENVIRONMENT FOR THE SR ARCHITECTURE



A,B	Speical Registers
PC	Program Counter
STATUS	Status Register
MAIN MEMORY	Main Memory
E_1	Environment for Process 1.

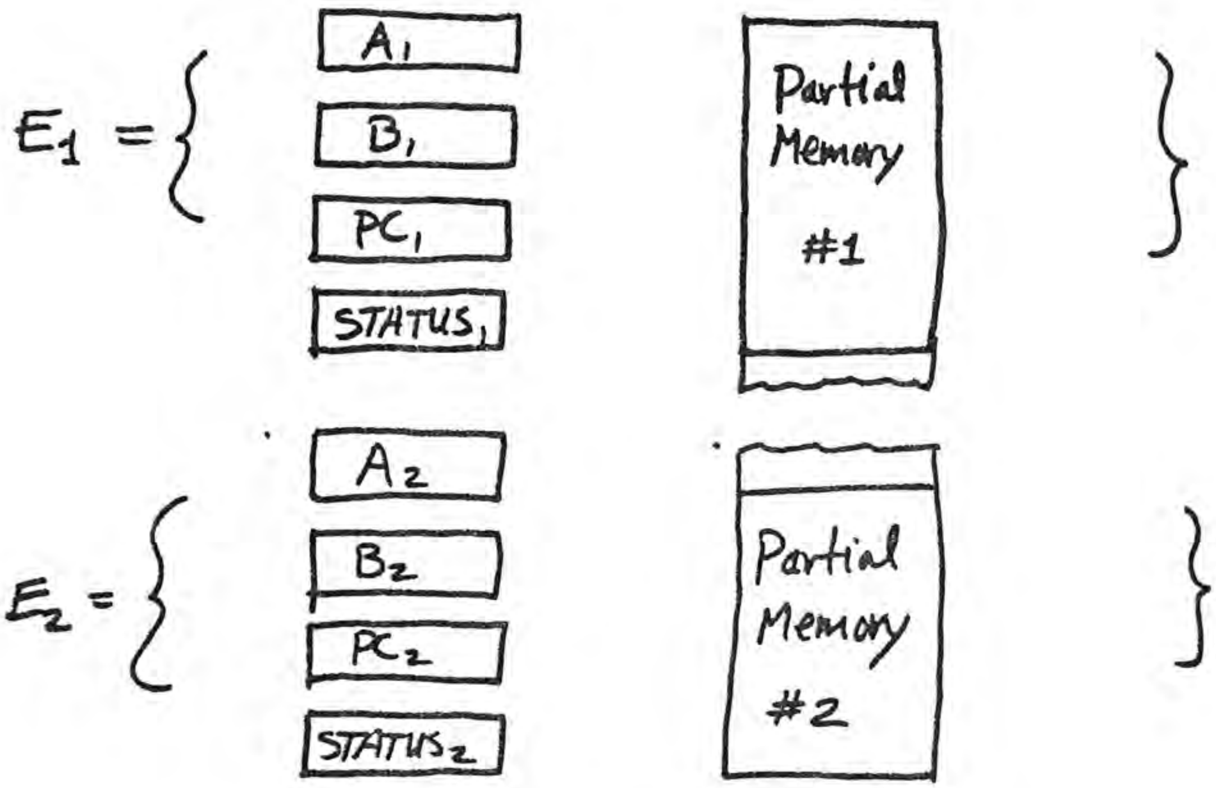
The SR architecture shown in Figure 5 easily supports a single process because there is a one-to-one correspondence between machine resources and the process.

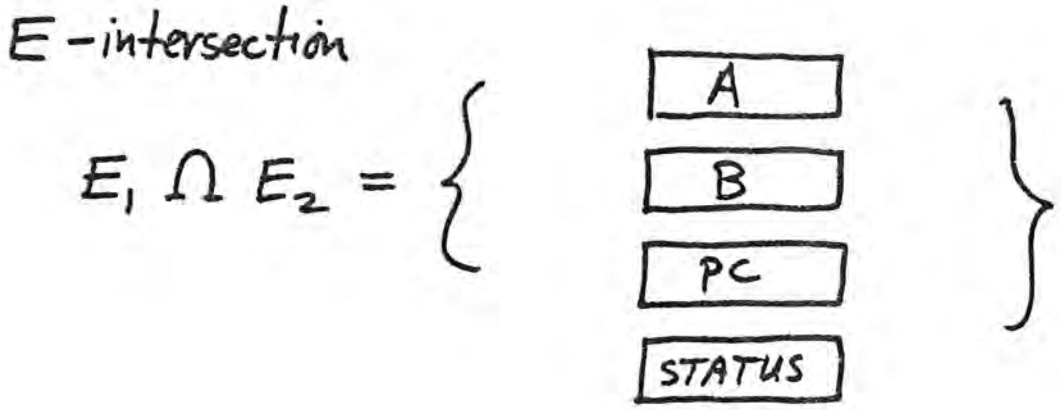
When the architecture of Figure 5 is used to support two or more processes, an environment is needed for each process. Two or more process environments may be needed when resources of the architecture are shared over time by multiplexing. The process that performs this multiplexing is called an E-switch. An E-switch, then, is a process that transforms environments into other environments.

Typically, the E-switch is performed by the hardware, but when it is not, it must be protected from the processes that it multiplexes. Such protection is afforded by priviledged execution modes or other operating system schemes. Dual state minicomputer architectures have evolved for the purpose of protecting E-switch processes. It must be noted that similar solutions for monostate architectures are evolving. This topic is under study by Shriver et.al (9).

Figure 6 shows the environments of two processes running on a single SR architecture. During an E-switch, resources belonging to the intersection of the two process environments must be saved. This set of perishable resources is called the E-intersection, and is one source of complexity in contemporary shared computer systems. The evolution of minicomputer architectures is shown in the following sections to be

FIGURE 6. THE ENVIRONMENTS OF TWO PROCESSES RUNNING ON AN SR ARCHITECTURE





partially governed by the E-intersection. This observation follows from a rule governing secure E-switch processes.

<u>E-switch Rule #1</u>: During an E-switch from environment E_1 to environment E_2, the E-intersection set of resources, $E_1 \cap E_2$ must be saved in the complement address space of environment E_1.

Complement = $E_1 - (E_1 \cap E_2)$

When this rule is applied to the two environments of Figure 6, the complement space is partial memory #1. Thus, when switching from E_1 to E_2 we must save the E-intersection set of resources in partial memory #1.

The partial memory space #1, of Figure 6, contains both program and data. Since we want to avoid destroying instructions and also to keep programs reentrant, the E-intersection resource set must be saved in the data portion of E_1.

Two environments of a stack architecture are shown in Figure 7. If the advise of the previous argument for reentrant code is heeded, then the E-intersection resources must be stored in the data portion of each environment. Since the E-intersection consists of the set of pointer registers, this leads to storing the set $\{$ PB, PC, P1, SB, EP, SP, SL, STATUS $\}$.

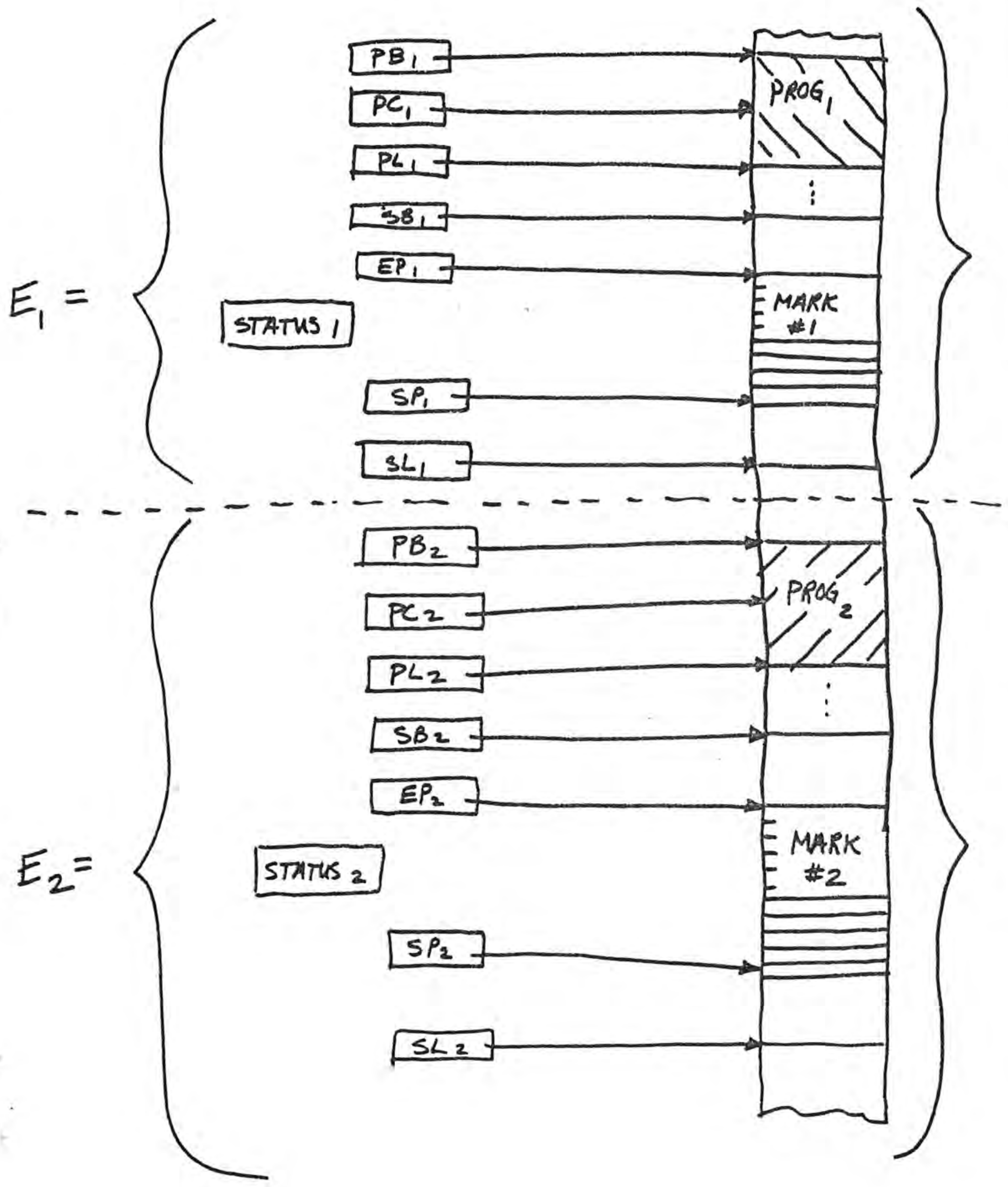


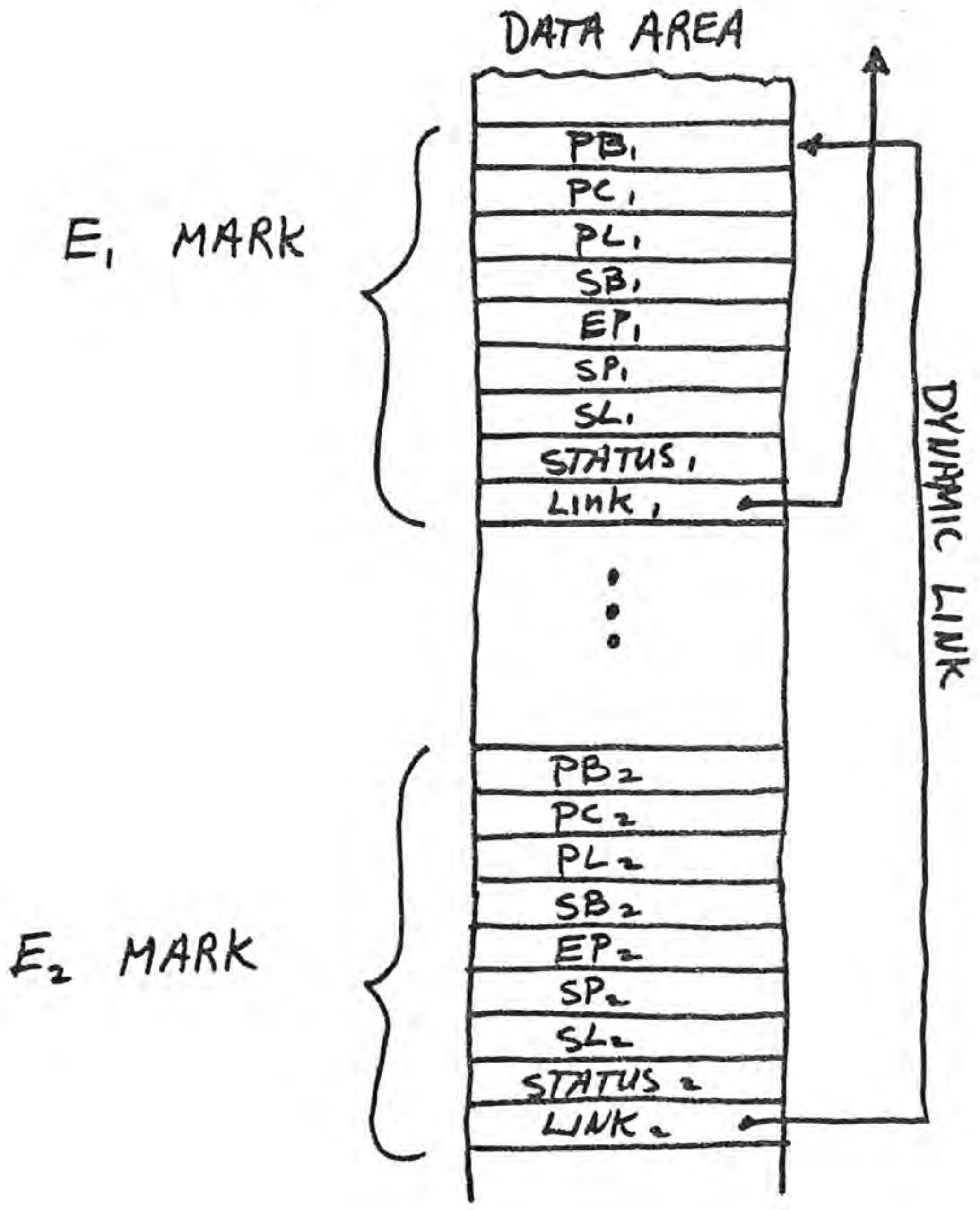
FIGURE 7. ENVIRONMENTS FOR TWO PROCESSES ON A STACK MACHINE

Rule #2 for Nested E-switch Architectures: Save the E-intersection in the data portion of the complement set, and provide a dynamic link between environments.

This rule is implemented in the SA architecture illustrated in Figure 8. The E-switch rule for non-nested environments will be different than the one proposed above. In general, the E-intersection resource set is stored in a protected area managed by the E-switch.

With these two fundamental considerations in mind, the evolution of "typical" minicomputer architectures can be studied and evaluated. In the next four sections, four architectures are shown to represent a progression from limited addressability/E-switch control. These four architectures were selected from a variety of commercially available minicomputer systems to indicate how far minicomputer architectures have evolved toward the goals of addressability and E-switch control.

FIGURE 8. IMPLEMENTATION OF E-INTERSECTION SAVE ON AN SA ARCHITECTURE



IV. A LOW-COST MINI ARCHITECTURE (5)

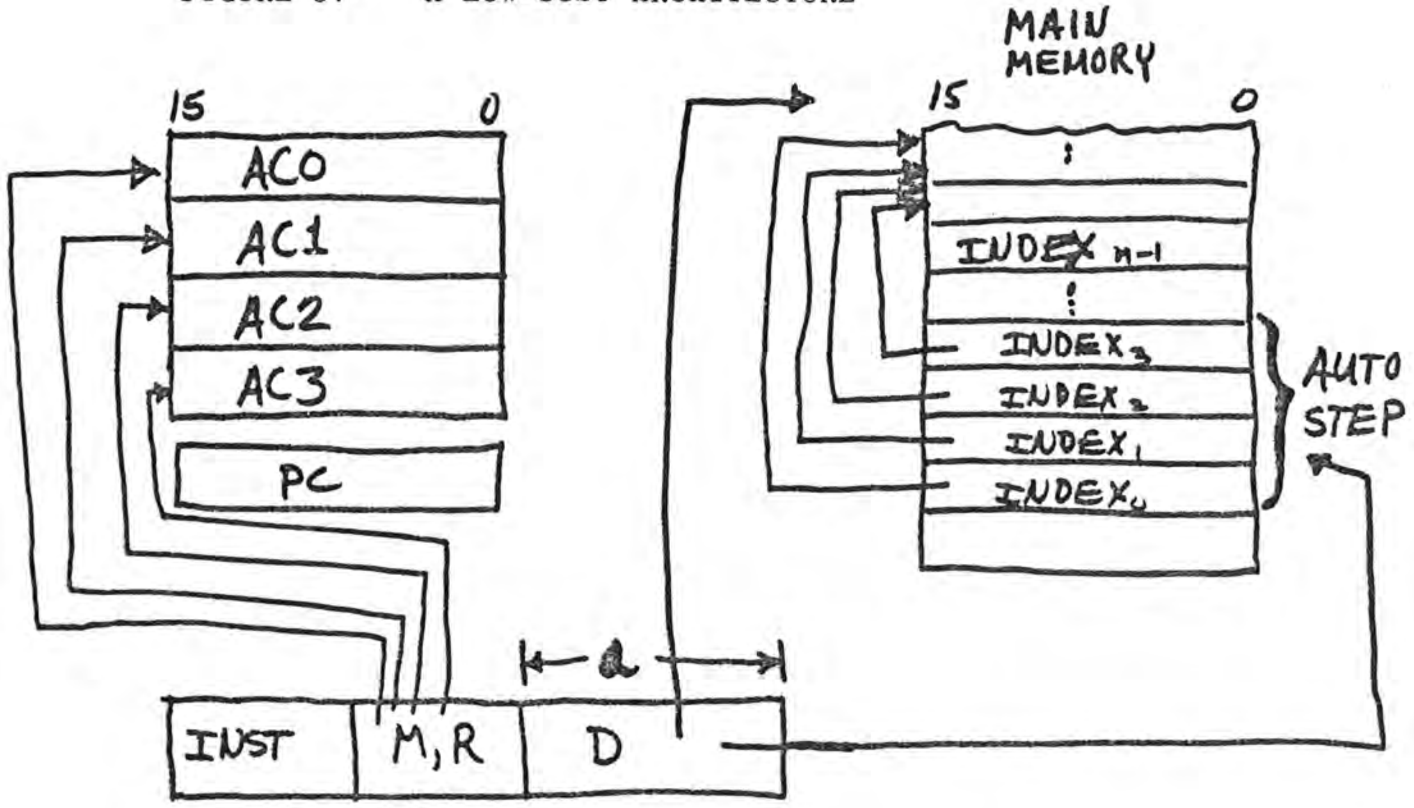
The mini architecture LC of Figure 9 shows an organization with 4 general purpose working registers, and a typical instruction set format. Operands are obtained from one of the registers, and either a register or memory location. In addition there are reserved memory locations dedicated to auto-increment, or auto-decrement indexing. Each time one of the INDEX words is used as a pointer to data, it is either incremented or decremented by one. The idea is to gain addressability and processing efficiency through auto stepping combined with indirect addressing.

The working registers are used as operands. AC2 may also be used as an index and AC3 is used to save the return address (old PC) during subroutine calls.

The addressability of LC is computed from the 4 accumulators ACO-AC3, the 2^a direct address locations, and the n INDEX locations each accessing 2^{16} other locations. This yields an addressability of 2^{26} when n=16, a=4. If we include the index capability of AC2, the result is an addressability of $4\ (n+1)\ 2^{a+16}$.

The following assembly language example demonstrates the use of the indirect auto step registers in LC (1). Suppose the problem is to move 30 words from location 2000_8 to 5205_8 in reverse order. The @ symbol indicates an indirect address mode.

FIGURE 9.   A LOW-COST ARCHITECTURE



E-intersection = $\{$ ACO, AC1, AC2, AC3, PC, INDEX_0, .. INDEX n-1 $\}$

$A_{LC} = 4.n.2^{a+16}$

```
COPY:       LDA       0, CNT      ;Set-up autoincrement...
            STA       0, 21       ;... in INDEX location $21_8$.
            LDA       0, CNT+1    ;Set-up autodecrement...
            STA       0, 35       ;... in INDEX location $35_8$.

LOOP:       LDA       0,@21       ;Get a word...
            STA       0,@35       ;... and move it.
            DSZ       CNT+2       ;decrement counter and test...
            JMP       LOOP        ;...otherwise repeat
            JMP       0,3         ;return thru AC3.

CNT:        001777                ;$2000_8$-1 pointer

            005206                ;pointer to destination
            000036                ;counter $36_8$ + 30.10
```

This program initializes an autoincrement INDEX located at memory address 21 to 001777_8. It next utilizes the pointer at 35 wit 5206_8. This is done by copying the values from location CNT and CNT+1 into ACO and then from ACO into 21 and 35, respectively.

The loop is executed by indirectly loading a word via 21 into ACO. The word is then stored indirectly via 35. The value of the pointer at location 21 is incremented _before_ being used, and the value at 35 is decremented _after_ being used.

The loop is exited when location CNT+2 has been decreased to zero by the decrement-skip-if-zero instruction DSZ. The JMP 0,3 instruction performs a return to the address saved in AC3.

Clearly, this architecture is weak in terms of its E-switch potential. Each time an E-switch occurs, the E-intersection must be saved. The locations to be saved include part of main memory since the auto step INDEX words reside in main memory.

In addition, this architecture has limited subroutine capability because only one return address register AC3 is provided. Thus, the nested E-switch potential is limited as well.

The LC architecture sacrifices E-switch potential in exchange for addressability. Without the INDEX set and restricted subroutine return address register, addressability would be severely limited.

V. AN ADDRESSABLE MINI ARCHITECTURE (6)

The architectures of Figures 2 and 3 provide the greatest addressability of any architecture discussed in section II. The next minicomputer discussed incorporates both addressing mechanisms illustrated in Figures 2 and 3, see Figure 10.

The addressable mini architecture, AM of Figure 10 consists of n=6 GP registers, a stack pointer, SP, used for recursive subroutine calls, and a PC, PS register pair.

The instruction format of AM allows GP index addressing, SP operand addressing, and TA two-address addressing.

A program to move 30 words from 2000_8 to 5206 in reverse order is again used to demonstrate the AM architecture. In the program below, % indicates that an operand is a register, # indicates an immediate operand, and ( ) indicates that the register is being used as a pointer instead of an operand.

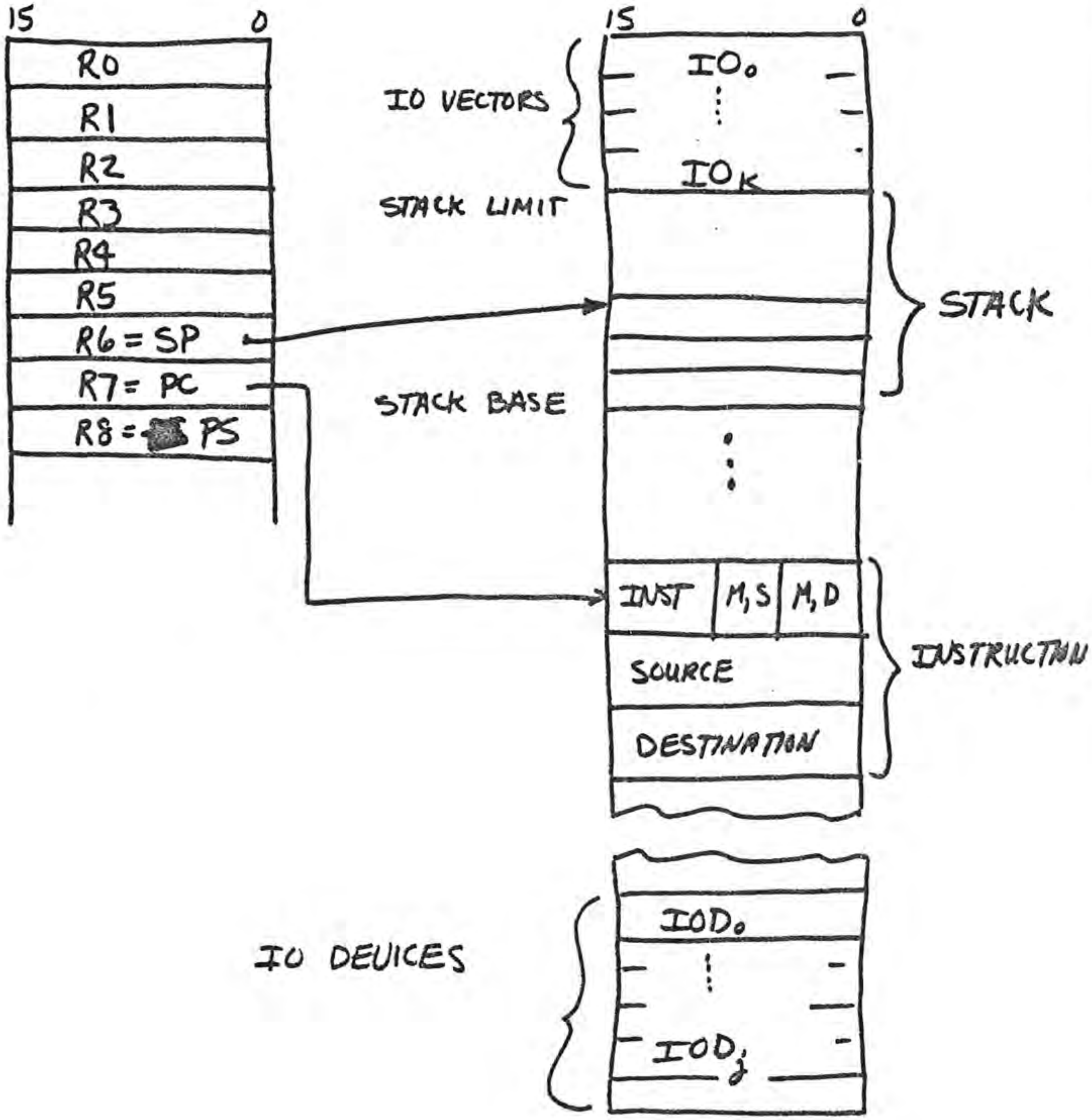


FIGURE 10. AN ADDRESSABLE MINI ARCHITECTURE

```
COPY:      MOV     #2000,%1     ; initialize pointer
           MOV     #5206,%2     ; initialize pointer
           MOV     #36,%0       ; initialize counter

LOOP:      MOV     (%1)+,-(%2); copy and auto step
           DEC      %0          ; count down
           BNE     LOOP         ; repeat
           RTN      %7          ; return, recursively
```

The E-intersection of the AM architecture is very large. Notice that the registers, stack, IO Vectors, and IO devices are all shared resources. Therefore the E-intersection contains these resources.

E-intersection (AM) $\{ RO \ldots R8,\ IO_o \ldots IO_k,\ STACK,\ IOD_o \ldots IOD_j \}$

The $IO_o \ldots IO_k$ vectors are useful for rapidly selecting a proper IO service routine and executing it to handle IO requests. The STACK assists in subroutining and the IO/Devices are treated the same as memory locations. This simplifies IO programming.

Actually, byte IO, when performed through working registers instead of special locations, reduces the E-intersection. Also, DMA (direct memory access) IO can reduce the E-intersection if the device is protected from interfering processes (an operating system function).

The AM architecture is a step forward for addressability, but still restricts the use of a minicomputer in a shared fashion because of its large E-intersection. (Obviously there are ways to minimize the harmful effects of the E-intersection. We will not discuss them here, but merely point out their problems).

VI AN E-SWITCH MINI ARCHITECTURE (7).

The E-switch architecture of Figure 7 is implemented in a varietyof minicomputers designed to support high level implementation languages. The languages supported by nested E-switch machines are block-structured. Therefore, the high level language environments created to implement systems in these architectures conform with machine environments established by MARKs.

Since processes are possible that are not nested within other processes, there must also be a mechanism for saving E-intersection resources when switching to non-nested environments. The E-machine of Figure 7 maintains a separate process stack for this purpose. Furthermore, the E-switch of the E-switch mini runs in a privileged mode to protect it from other (user) processes.

A sample of E-machine implementation language is illustrated with a program that solves the problem of moving 30 words of memory from location SRC to location DEST in reverse order.

The high level language is translated into stack architecture instructions that manipulate reverse expressions.

```
COPY :       PROCEDURE (SRC, DEST) ;
             DECLARE (SRC (29), DEST (29)) Word;
             DECLARE (I.J) Word;
             DO  I = 0 to 29;
                 J = 29 - I;
                 DEST (J) = SRC (I);
             END;
             RETURN;
END COPY
```

This routine, when compiled and executed on the stack architecture of Figure 7 creates an environment. The environment consists of arrays SRC and DEST, the code for COPY, the variables I, J, and a MARK in addition to the pointer registers referencing the stack.

Figure 11 illustrates the configuration of an E-switch minicomputer based on the SA architecture during execution of the COPY code. The MAIN program that called COPY passes a pointer to the environment containing arrays SRC and DEST. COPY is able to access these values because their addresses have been forwarded into the environment of COPY. Thus, the values of SRC and DEST have become a part of the COPY environment.

A Dynamic Link between the COPY MARK and the MAIN MARK provides a return path to the outer environment.

Calculations for executing the DO loop and arithmetic assignment statements are done by pushing/ popping values on the stack at location SP. The values are loaded onto the stack by copying them from <u>local</u> addresses (I,J) or from non-local addresses (SRC, DEST).

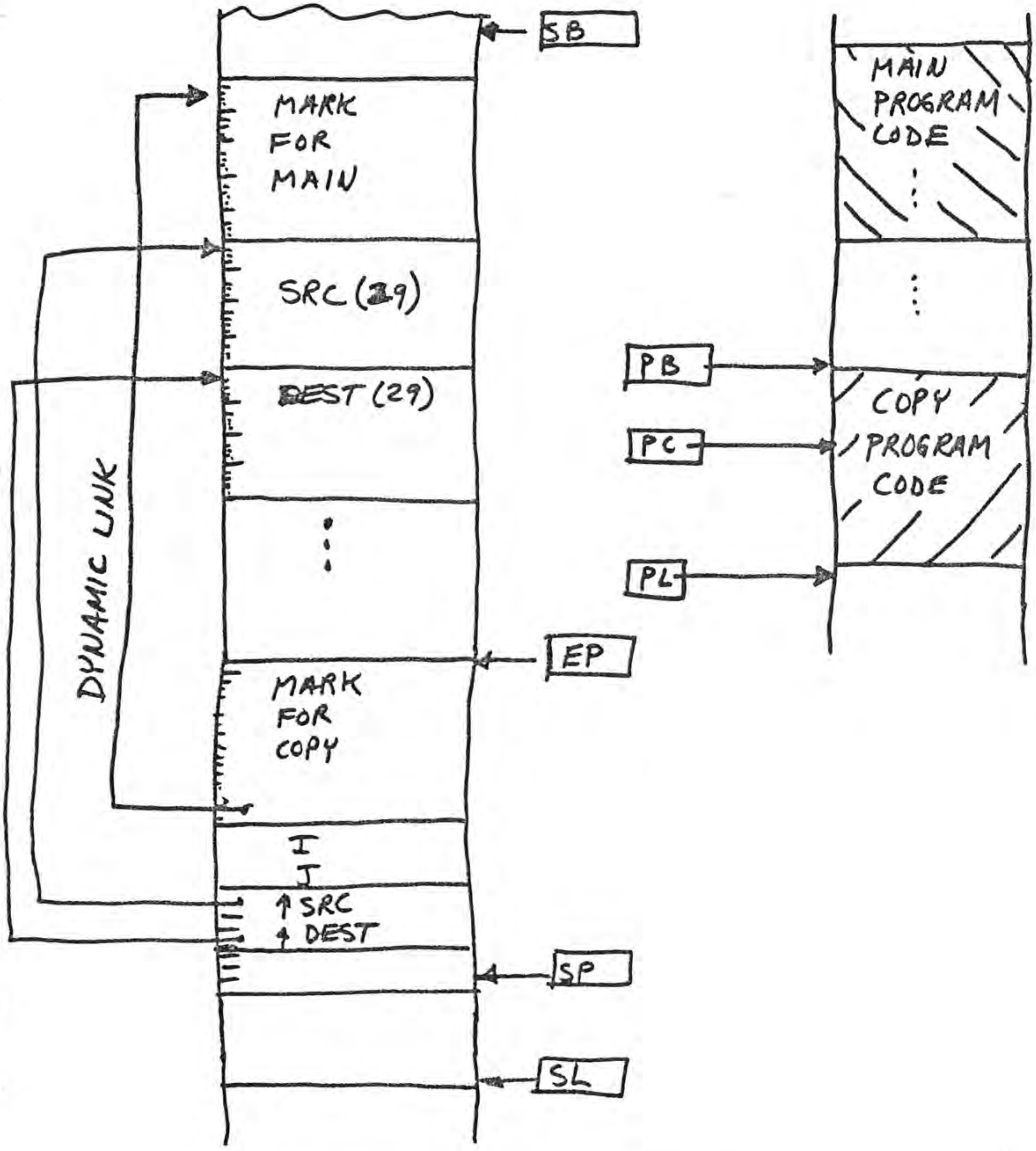


FIGURE 11. NESTED ENVIRONMENTS FOR SAMPLE PROGRAM

The stack architecture appears to be an efficient E-switch architecture. The disadvantages of this approach should be pointed out, also. The SA architecture's limited addressability results in a large number of PUSH and POP operations being performed. It is not unusual for 25-40% of the program code to consist of PUSH and POP instructions. This means that program space and execution time is being traded-off for E-switch capability.

In the following section, a very recent architecture is used to demonstrate a compromise between addressability and E-switch efficiency.

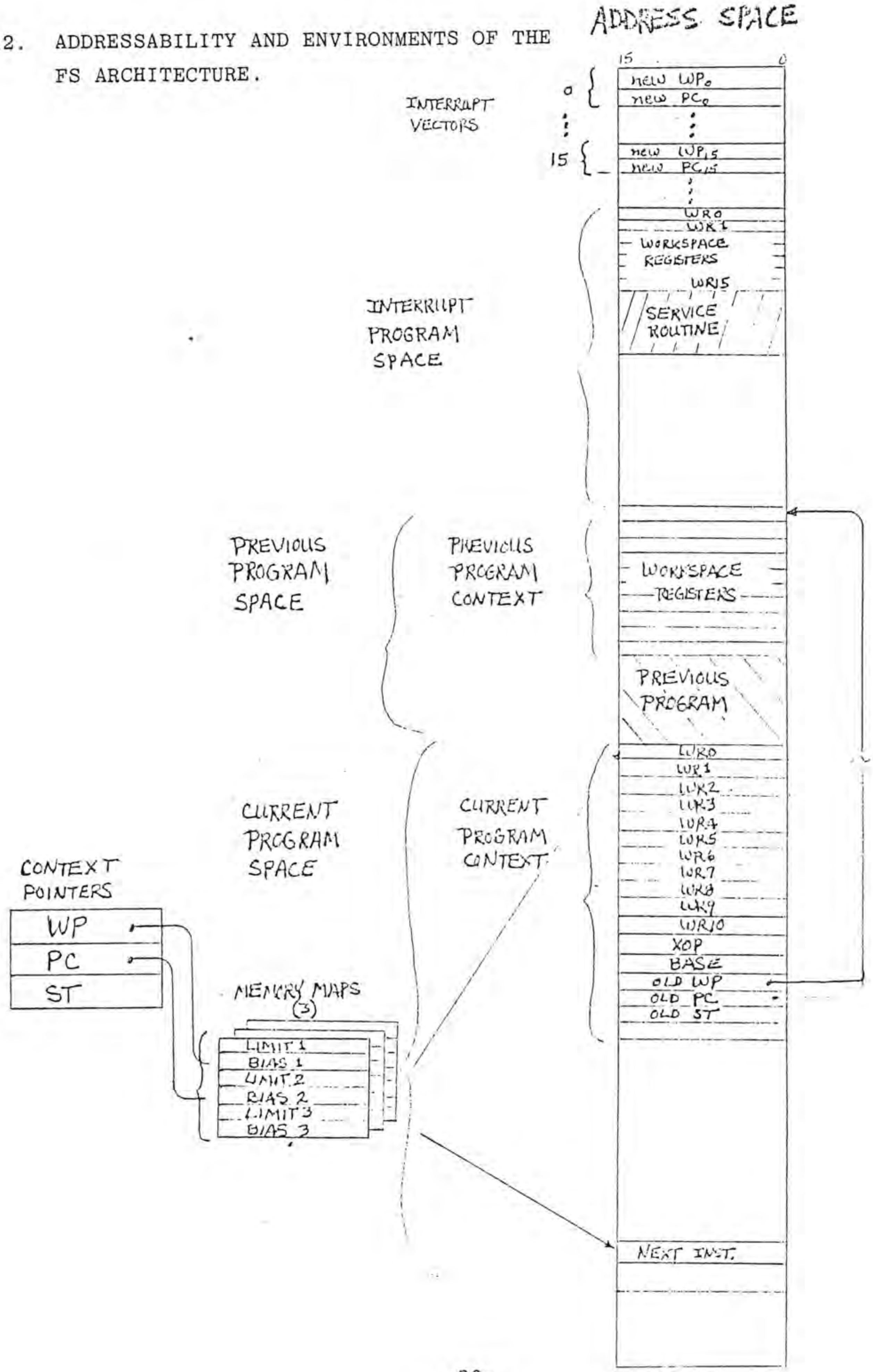
VII. A FUTURE MINI ARCHITECTURE (8)

Hardware advances have narrowed the gap between main memory and logic speeds. In addition, the cost of added cpu complexity of mini systems has decreased to the point where future computers can take advantage of architectures with large addressing capacity, E-switch potential, and relatively large instruction sets.

A future organization should incorporate advances in addressability, minimize the E-intersection of resources, and facilitate the implementation of software that meets the end-user requirements stated earlier.

The FS architecture of Figure 12 illustrates a recent advance in minicomputer organization. The working registers WR0--WR10, XOP--OLD ST actually reside in main memory rather than the cpu. There is a copy of these registers in each environment, thus reducing the E-intersection during E-switch. A dynamic link connects

FIGURE 12. ADDRESSABILITY AND ENVIRONMENTS OF THE FS ARCHITECTURE.



nested environments in a way that gives this architecture a stack-like capability for nesting and recursion.

The cpu actually maintains three registers, WP (work pointer), PC (program pointer), and the ST (status) register. In addition, three memory maps are also maintained in the processor. These maps are used to extend the addressability of the processor. When a memory reference takes place, the value of WP or PC is modified by one of three BIAS registers in the memory map. If the reference falls between LIMIT_i and LIMIT_{i+i} then BIAS_i is used in the following way. The BIAS_i register is shifted left 5 bit positions and added to either of the pointer registers (PC or WP). This yields a 20-bit memory address, hence the addressability of this machine is 2^{20}.

$$A_{FS} = 2^{20} \approx 1 \text{ MB}$$

The E-intersection of FS is seen to consist of the memory maps (up to 3 maps provide 3 process environments without saving), the three context pointers, and the dedicated Interrupt vectors in low memory.

If the "previous" program calls the "current" program either as a nested or non-nested environment, the working registers need not be copied or saved because each program carries its own copy of working registers with itself. If the current program is a procedure with parameters, then the dynamic link can be used to access the parameters. Thus, nesting is accomplished in a manner similar to the SA E-switch organization.

The WS (Work Space) registers, WR0-WR10 are used quite similar to the registers of the TA architecture of Figure 10. This allows a programmer to use the WS registers as pointer or operand. This feature is exploited in the following programming example. Again, the program moves 30 words from SRC to DEST in reverse order.

Notice the mnemonic symbols for hexadecimal constant > , indirect address @, and comment *.

```
*
*   Set-up  WP, PC, ST for operating system
*
 OS  DATA      WS, PC, > F       initialization
 WS  DATA      SRC               WRO points to SRC
     DATA      DEST              WR1 points to DEST
     DATA       > 1E             WR2 index and counter
     DATA       > 0              WR3 index
     BSS       24                WR4-WR10 unused
SRC  BSS       30                30 words
DEST BSS       30                30 words
*
*    Move from SRC to DEST
*LOOP MOV      @SRC (3),@DEST(2)  Copy in reverse
     INC       2                 step index
     INC       2                 step index
     DEC       3                 step counter
     DEC       3                 step counter
     JNE       LOOP              repeat
*
*    register 2 and 3 are used as index registers
*
```

The XOP register shown as part of the WS registers is used to extend the basic FS architecture. Unimplemented instructions (there are 16 such op-codes) cause a trap when encountered in a program. The pointer stored in XOP is then used to locate a micro-program, software program, or hardware module that performs a dedicated operation on the data. With this feature, the FS architecture is able to be extended beyond it's original design limitations.

The FS architecture represents one approach to the ultimate in minicomputer evolution. There still remain difficulties with this organization that have not been discussed here (multiple precision arithmetic is difficult to perform). But within the goals of addressability and E-switch potential, the FS architecture is at the apex of minicomputer evolution.

IIX SUMMARY AND CONCLUSIONS

The obvious goals of end-user support remain a problem for minicomputer systems. It is not easy to determine if a new architecture is able to solve problems leading to better end user support. Therefore, it is mere speculation to claim that the evolution of minicomputer architectures is improving end-user support. In fact, it is difficult to measure "success" or "failure" of a given architecture in terms of the applications supported.

What can be said from the analysis of contemporary and new architectures is that they either facilitate addressing and E-switching as demonstrated. The previous discussion appears to support claims of an "improving" collection of organizations.

The importance of addressability and E-switching is recognized and need not be justified. The impact of these two features of minicomputer systems is not recognized; however, nor have they been measured and evaluated. We can only speculate, once again, as to their impact.

It was noted that the necessity for sharing hardware is being questioned. If we remove cpu time-multiplexing from the list of requirements, then E-switching may have little impact on future systems.

Currently, sharing is applied to the most expensive subcomponent of a system. In the mini-computer world, this means that printers and mass storage should be shared, as opposed to the cpu.

Transaction computing is a form of interactive computing where small bursts of data is processed in a very short period of time. Typically transaction computing requires access to large storage units. As an example, updating a person's account with a bank is a form of transaction computing.

A transaction requires very unsophisticated computing, and yet access to a large data base is necessary. If minis are to be used for transaction computing, then the goals of future architectures must be modified to meet this new requirement.

Perhaps mini architectures should evolve toward supportof virtual databases, or perhaps to support virtual peripherals. Or perhaps the future mini-computer will support communications operations, word processing operations, or new operations not yet conceived. If so, the current architectures must evolve in new directions.

Acknowledgements: The author appreciates the many thought provoking discussions with Bruce Shriver during the development of this paper.

REFERENCES

(1) Lewis, T.G. Minicomputers: An Attitude, Software Engineering Handbook, National Bureau of Standards, Computer Science Section, Ed. Gordon Lyon, Washington, D.C. 20234.

(2) Withington, F.G., Beyond 1984: A Technology Forecast, DATAMATION, Vol. 21, No. 1, Jan. 1975, 54-73.

(3) Horn, B.K.P., and Winston, P.H. Personal Computers (who needs timesharing?), DATAMATION, Vol. 21, No. 5, May 1975, 111.

(4) Lewis, T.G., How Large Should A Computer Be? SIGMINI NEWS, ACM, Vol. 2, No. 1 (1976).

(5) Technical Publications, Data General Corp., Southboro, Mass. 01772, (a) How to Use The NOVA Computers; (b) Introduction to RDOS (093-000083-00)

(6) Software Distribution Center, Digital Equipment Corp., 146 Main Street, Maynard, Mass. 01754
(a) PDP-11 Processor Handbook (1973);
(b) PDP-11 Handbook (1969)

(7) Burns, R. and Savitt, D., Microprogramming, Stack Architecture Ease Minicomputer Programmer's Burden, Electronics, Feb. 15, 1973, 95-101.

(8) TMS 990 Computer Family Systems Handbook (945250-9701) Texas Instruments, Inc., Austin, Tx. 78767.

(9) Shriver, B. D., Anderson, J. W., Wagespack, and Bambet, R., A Virtual Machine Monitor For Mini-Computers, Proc ACM-76, October 1976. Houstin, Tx.

A NEW LOOP STRUCTURE FOR DISTRIBUTED MICROCOMPUTING SYSTEMS

H. Jafari
Electrical Engineering Department

T. Lewis
Computer Science Department
Oregon State University
Corvallis, Oregon 97331

ABSTRACT

This paper presents a new distributed computer network structure appropriate for a network of microprocessors. The new network structure combines advantages of a ring structure; simplicity, high line utilization, concurrent service, distributed control information, minimum delay for minimum cost, and high reliability. This is accomplished using two loops. The "inner" loop is for data transfer. It is partitioned into N buses interconnecting N microprocessors. The "outer" loop is for control information to pass along under the guidance of a bus controller. Results for simulations of contemporary proposals (Pierce, Newhall, and Reames et al.) and the new network proposed in this paper show that the new structure substantially improves throughput when compared to the other structures.

INTRODUCTION:

Researchers have proposed distribution of low-cost computing processors throughout a network as an alternative to expensive and highly centralized computer systems (SPAN 76). The results have shown that completely distributed systems lead to a great deal of inefficiency due to increased hardware and software overhead and often fail to deliver acceptable throughput as expected. In addition a computer network introduces other complexities concerning deadlocks, network reliability, traffic regulation, and scheduling.

This paper introduces a new network topology with high throughput rate for distributed computer systems. The network has an improved response time, greater throughput, and is more reliable than the Pierce, Newhall, or Reames - Liu loop network topologies.

I. DESIGN PHILOSOPHY

A distributed computer system interconnects several heterogeneous or homogeneous nodes which communicate with each other through network media. A heterogeneous network is a collection of architecturally different nodes while a homogeneous network is a collection of architecturally similar processor nodes.

Farber (FARB 72) lists the motivations to develop a distributed computer system as any or all of the following:

1) Modular Growth
2) System Reliability
3) Incremental Upgrading of Processor Nodes
4) Dynamic Restructing
5) Decreased Design Time
6) Ease of System Validation

In addition we include:

7) Tailored Design to the Users Needs
8) Better Throughput (Speed)
9) Less Cost

With these motivations in mind, several people have proposed and implemented a variety of network topologies in hopes of efficiently managing distributed computer systems.

The topology of the interconnections in a network is of great concern since it has a major effect on the performance of the distributed system. The most highly connected network is to connect every computer to every other directly. This involves $N(N-1)/2$ interconnections for N nodes and is very costly unless N is very small. A less costly topology requiring N interconnections and an additional central control processor is the star configuration. The central control computer provides node-to-node interconnection by switching from one node interconnection pattern to another upon demand. Furthermore, each distributed star computer system can be connected to another star computer system by connecting the two central control computers together, and with appropriate control algorithms, this will allow any node in either subnetwork to communicate with any others.

A problem with star network computer distribution is reliability of the system, for as soon as the central computer exhibits faulty functions, the whole system breaks down. In addition, the central control processor is an overhead cost added to the whole system. If the number of nodes around the central processor is small, then the advantage of this system is its speed, also because the links between computers are bidirectional, the system has a very good throughput. We will not include the star network in the work reported here because of its poor reliability (STRE 76).

Another philosophy is to connect all the processor nodes in a loop or ring configuration. This is called a loosely coupled connection since each node is connected to others by only two links, an input link which comes to the node and an output link that goes away from the node. Loop systems are attractive for mini-micro computer networks due to their possible high line utilization and because they are simple. This last philosophy has attracted the attention of many researchers who have designed a variety of network systems based on the simplicity of a loop. The new loop structure will allow more parallel communications between nodes, while taking advantage of loop simplicity.

II. PREVIOUS LOOP CONTROL ALGORITHM

The first loop structure system was suggested by NEWHALL (FARM 69). In the NEWHALL loop a round-robin control passing mechanism circulates around the loop and allows only one node at a time to transmit one or more messages through the loop. Therefore, the rest of the nodes have to wait and this causes a queuing time in sending the messages which limits the achievable loop utilization.

A version of a loop discipline similar to the NEWHALL discipline is allowed with IBM's SDLC (DONN 74) (or with the largely equivalent HDLC (DAVI 73)). In this discipline a central controller originally sends a poll command around the loop. The first attached device wishing to transmit is thereby enabled to transmit. This device then ends its transmission by passing the poll on, so that control passes around the loop in a manner similar to the behavior of a NEWHALL loop. This variation is not explicitly studied here because of its similarity to the NEWHALL loop. On the other hand, Pierce (PIER 72) introduced a new mechanism that improves network utilization by time multiplexing the loop. That is, the information sent around the loop is divided into fixed-size packets and to send a message, each node checks for an empty packet before transferring all or part of its message. If a message is smaller than the fixed-size packet, the excess space is wasted. If the message is too large to fit the packet, then the message is broken into two or more packet-sized messages. When a processor node transmits a message, it must first check whether the next packet or time slot passing by it is empty. If it is, control will pass to the processor nodes transmitter to see if there is any information to be transmitted. In case the packet is not empty, the processor node checks to see if the destination address in the packet matches the node address. If so, the processor node transfers the packet information into its buffer. If the packet address does not match the processor node address, then the processor simply passes this packet to the next node. The transmission mechanism is as simple as waiting for the beginning of an empty slot and filling it with a packet, but disadvantages of this system include:

a) problem of dividing messages into packets
b) problem of packet reassembly which occurs when messages are divided into packets and then sent separately, so a sorting problem arises.
c) messages do not always fit into a fixed number of packets, so there are some partially empty packets with corresponding waste of network capacity.

Therefore, neither Newhall or Pierce loops make very efficient use of loop topology. Reames and Liu (REAM 75) introduced a new message transmission mechanism called DLCN (Distributed Loop Computer Network) which allows multiple messages in the loop as the Pierce loop does and messages of variable length as the Newhall loop permits.

DLCN incorporates a variable length shift register before each node's transmitter, see Figure 1. A message can be transmitted through the loop whenever no other message transmission is already in progress, or no other messages have started passing that node. In this case, the variable shift register provides a delay in the incoming message equal in size to at least the size of the message to be inserted. Once an incoming message has been delayed in this manner, it is transmitted ahead of any incoming messages which are in turn delayed during the time needed to transmit. The contents of the variable length shift register will gradually decrease in length and finally be eliminated if there is not enough traffic. DLCN actually combines Newhall and Pierce loop advantages by allowing simultaneous message arrival with message transmission, and also provides automatic traffic regulation based on observed system load, but DLCN favors infrequent requests while delaying more frequent requests for network service.

A disadvantage of the DLCN is the complexity of interface mechanism and, therefore, the cost to build such an interface. Secondly, inserting a variable shift register at each node lowers the reliability of the overall loop since it adds one new possible failure mode. Also, when the number of nodes in the loop increases, eventually the queuing time will increase drastically. This limits the number of nodes inserted in a loop.

Potvin (POTV 71) introduced a generalized distributed computer system called the star ring system. It combines the control feature of a loop network with the message transmission features of a star network. It is somewhat similar to Newhall's technique for passing control along its loop and in its method of time multiplexing message transmission. The system is restricted by the number of nodes on the loop because the central star ring is common to all the nodes and, therefore, not more than two nodes can talk to each other at any time. This slows the throughput of the system by a great amount. Potvin considers only a very small number of nodes in the network.

All the above communication loops suffer from the following common shortcomings in addition to the problems discussed above.

1) The stream of data is in one direction and therefore, sometimes the transmission of data from one node to its neighbor node takes place through the rest of the nodes causing more delay and less reliability than necessary.
2) If a node starts sending a stream of messages to another node it will block out all other transmissions and network performance will decrease by a great amount. Thus, the networks mentioned above are sensitive to local demands that affect the performance of all nodes.
3) If there are errors in the address fields of the message and/or a node fails to function properly, messages will saturate the loop, in all the above systems. Several different techniques have been used to recover from errors, but this eventually slows down loop communication.

4) If there is a failure in the loop, the whole network will fail to operate.

A new experimental loop is proposed that will enable the whole network to recover from the above shortcomings. The concept is to distribute data and control into two different loops (a data loop and a controller loop). The data loop is actually a segmented loop consisting of a single segment connecting nodes. Each node is interfaced to the loops by a switch that may be turned "on" or "off". The control loop operates by a simple arbiter, which accepts requests for communication, decides the minimum route, and sets up the data paths between nodes by turning appropriate switches "on" and non-appropriate switches "off".

III. DESCRIPTION OF NEW LOOP NETWORK

We suggest a modified loop network in which control messages and data messages are transferred through two different communication lines. This adds flexibility to the network for very little increase in cost. The loop network system is configured from four different components:

1) control line loop
2) data line loop
3) processor nodes
4) a special processor node dedicated to line control.

The control line loop employs a polling technique to start and stop the transfer of messages from a source node to a destination node. Transmission is accomplished through a "double handshake" where a request to send is followed by an acknowledgement that the message has been received. In particular, there are two different possible types of messages, SYN/ACK and Relay Control which can be sent over the control line.

SYN/ACK: When a node desires to communicate with another node (SYN), or respond to end of communication (ACK), then it will send a message to the controller containing the address of the source node and the address of the destination node along with the command (either SYN or ACK) to be performed by the controller. Messages of this type have the format shown in Figure 2(A).

Relay Control: Messages sent from the controller to a source or destination to inform the node that a message is being sent to it (destination), or that a message has been received by the destination node (source), or directing other nodes to position their data switches to bypass the data and allow it to continue along the data loop until reaching its intended destination. The messages of this type are shown in Figure 2(B).

The data line loop transfers all the data messages from any source node to any destination node through a minimum route which has already been set up by the controller as explained above. The data line loop illustrated in Figure (3) is interfaced to each node through a three-way switch at each node which enables the node to connect segments of the data line together and either bypass the node or connect the node to the data loop so that the node can receive or send data. The controller sets the three-way switches before each data transmission is allowed. For example, if Node 1 of Figure (4A) is to send data messages to Node 3, then the switches and data segments are connected in one of the configurations shown in Figure (4). Observe that the connection of segments of the data loop permit partial use of the entire data loop network Figure (4). Remaining segments of the data loop are available for concurrent data transmission to other nodes in the system. Therefore, simultaneous transfer over non-interfering segments of the network is quite possible. The combined effect of redundant alternate paths and concurrent transmission over non-interfering segments of the loop adds to the network reliability and throughput.

The partitionable loop structure described above is a general structure. In addition to the loop topology studied here, there is also the potential for other configurations. The topology of a specific network may require high-speed transmission between two or more nodes, depending upon the needs of these two processor nodes. In such a special case, it may be expedient to include additional "express" buses to supplement the basic loop. This can be done, for example, as shown in Figure (5), by merely increasing the capability of control line switches at these nodes. In the examples of Figure (5), supplemental data buses may be used to establish high bandwidth communication between Node 1 and Node 4. Alternatively, the response time of communication between Nodes 4 and 2 may justify an additional data line as shown in Figure (5B).

Processor nodes are configured from four elements:

A. A node control mechanism to perform data loop and control loop functions.
B. Control switches to switch the data lines.
C. Transmitter and receiver.
D. Terminal processor which may be a simple I/O device, a microcomputer, or an interface to another network.

Figure (6) illustrates these four elements. Each node control mechanism provides timing control, message detection, decoding and encoding of messages, controlling the data switches, transmitter and receiver control, and communication with its node terminal.

The control switch is a modular unit easily extendable through hardware changes; for instance, a control switch can control two data segments along with the receiver and transmitter. If the number of data segments interfaced to the node increases, the complexity of the switches will increase in a modular manner.

A simple transmitter-receiver can be time multiplexed or separated from each other by using separate channels which adds to complexity to the control switches. Figure (7) shows both a simple and more complex transmitter receiver section.

The loop network interface is designed as an "intelligent interface" so that no assumption about the processor terminal is needed. Any device may be plugged into the loop network regardless of its sophistication. All the control needed for any terminal to talk to the receiver-transmitter section is provided by the node control, thus allowing terminals to be of any type. The intelligence of the node controller is easily provided by a low-cost microprocessor and PROM.

The loop controller functions are as follows:

A. Sends and receives control messages to and from control line.
B. Schedules node communications.
C. Finds the minimum path between the nodes which are to communicate.
D. Provides a timing mechanism.

The control messages have the formats of Figure 2(A) or Figure 2(B). The controller decodes or encodes them by managing the right timing. Scheduling of nodal communication may be by any scheduling algorithm as LIFO, FIFO, round robin, or shortest-messages-first. For the routing algorithm, any method can be considered, but since all the needed information is within the controller, routing can be tailored to special applications of the network. The timing mechanism can be part of the controller's function to synchronize all the nodes or it can be varied in each individual node. Therefore, nodes can work synchronously or asynchronously. The function of the controller is flowcharted in Figure (8). The functions of the network controller are very straightforward and can be performed by any node in the network. We will assume a special control node microprocessor is used to perform the control functions for the entire network. In the comparisons to follow, we will include this special-purpose control node as an overhead cost, but it should be pointed out that the control functions required by the proposed loop can be carried out by any node. In terms of reliability, this means that failure of the control node does not imply failure of the entire network, because control can be passed to another (working) node on the loop.

IV. SIMULATION RESULTS

We modeled our simulation study after the work of Reames and Liu (REAM 75). They simulated the DLCN (Distributed Loop Computer Network), Newhall Loop, and Pierce Network. The results obtained in our study will be compared with their results. Our results will extend their results to provide an evaluation of all four network topologies. In the DLCN simulation model, the length of the shift register interface to the loop was 512 characters. For the Pierce model, Reames and Liu selected a packet size of 36 characters. This is an optimal packet size obtained by minimizing the product of average number of packets times the packet size. In the Newhall network, they simulated passing the control token only when the queue of messages in that node is empty instead of passing one message at a time at each node. This produces a shorter total message transmit time for the Newhall network.

For all of the systems simulated by Liu and Reames, message length has a truncated negative exponential distribution with a mean of 50 characters, minimum of 10, and maximum of 512 characters of which the first nine characters are control characters. Message arrival time obeys the Poisson distribution, and the number of nodes is 6.

For the new experimental loop, the message length and message arrival statistics, and the number of nodes are the same as above. There is no need for control messages along with data in this new system. For reliability purposes, we used the same number of characters by including control characters with the data. The messages in this system can be of any length without hardware or software constraints.

The scheduling algorithm is simple FIFO and the routing algorithm is to simply find the minimum path between two nodes in either direction. If two paths have the same length, the clockwise direction is arbitrarily chosen. The new loop network improves throughput when employing these simple algorithms for scheduling and routing.

Table 1 shows the average interarrival rate, data line usage, waiting time for each message to be transmitted, transmission time total transmission time, and control line usage for the new experimental network, as well as for the other three networks. Figure (9) shows the variation of mean total message transmission time versus mean arrival rate for all four networks. Figure (10) shows the changes in line utilization versus changes in interarrival rate for all systems, which indicates the load of the system, and finally, Figure (11) is a graph of mean control line utilization versus the mean interarrival rate for new experimental systems, only.

From Table 1, we see the Pierce and Newhall loops and new experimental loop have almost a constant transmission time for any load on the system (46 time units per packet for Pierce loop, and 63 time units per message for Newhall loop, and 52 time units for the new experimental loop). This is due to a constant delay in the transmission lines for Pierce and Newhall systems. For the new experimental loop, there is no delay in transmission line. Transmission time is equal to the transfer time of the characters in a message. For DLCN, message transmission time is variable and as soon as the arrival rate increases (that is, the system load increases) then the shift register delay line time will increase leading to an increase in transmission time proportional to system load. On the other hand, the queuing time at each node will not increase as fast as transmission time since whenever a message is ready to go in the loop, the node will insert the variable delay shift register in the loop and then the message does not have to wait longer. This explains why DLCN is faster than the Pierce and Newhall loops. The superior performance of the new experimental loop is due to multiple concurrent transmission, variable message length without any additional hardware or software overhead, and the ability to select the shortest path from the bidirectional segments of the loop. As we see from Figure (9), total transmission time for Newhall, DLCN, and the new experimental loop is the same for very low system load. But as soon as the load on the system goes higher, the total transmission time for the new experimental loop shows improvement over the others. In the Pierce loop, a message always has a mean wait equal to one-half of the packet size and must then be transmitted in several packets. For this reason, the Pierce loop can not compete with the others for low systems loads. As soon as the system load goes higher, the Pierce loop exhibits concurrency (simultaneous packets on the loop) and its performance improves over the Newhall loop which shows its inherent serial nature leading to poorer performance.

In our new experimental loop there is a minimum queuing time for SYN/ACK and relay control messages. For low loading of the network this overhead shows up as a significant part of the overall delay, but since these two control messages cause a constant average delay they contribute a smaller proportion of the delay as the network load increases. Typically messages are queued before being transmitted and the delay due to control messages is overlapped with the fixed control message's queuing time.

The greatest advantage of the new network is that segments of the loop can be activated simultaneously. The added concurrency of the new loop explains its increased throughput when compared with the other networks. From Figure (10) we see the mean line utilization is very low for all the networks. As system load increases the line utilization for Newhall network levels off at about 50 percent. For Pierce and DLCN systems, line utilization increases as system load increases. However, when the loop is utilized up to its maximum, the waiting time will increase drastically. The proposed network requires nearly half of the line utilization of the other loops simulated. Figure (11) shows a linear relationship between the mean control line usage and system load. This is due to constant delay for SYN/ACK messages, however the relay control message is of variable length, (changes are within 7 percent).

CONCLUSION:

The main goal of this work was to improve the throughput of a microcomputer network using a flexible, simple, and reliable loop topology.

The results of our simulation have shown that completely decentralizing microcomputers leads to a decrease in throughput compared to the expected throughput of n processors. The loss in throughput resulting from networking multiple processors can be partially compensated for by careful design of the network and its interfaces. Reliability can be achieved by permitting any node to take over the controller's job.

The hardware implementations given for the interface and line controller show compatibility of this system with microprocessor technology. Because microprocessors are low cost, this type of network can be constructed inexpensively.

Future research in this area will be done using different scheduling algorithms for the controller, using a different number of nodes, and with different types of loop structures. Also an investigation of a mathematical model for such a loop structure, as has been done in the past for other loop structures is needed. (SPRA 72), (HAYE 74), (KONH 72), and (KAYE 72).

ACKNOWLEDGEMENT:

The authors wish to express their appreciation to J. Spragins for his advice and consultation during this research, and to C. Reames for supplying us with his GPSS simulation program. Also we are thankful for the participation of W. R. Adrion.

REFERENCES:

DAVI 73 - Davies, D. W., Barber, D. L. A., "Communication Networks for Computers", John Wiley, London, 1973, pp. 234,235.

DONN 74 - Donnan, R. A., Kersey, J. R., "Synchronous Data Link Control: A Perspective", IBM Systems Journal, 13, No. 2, 1974, pp. 140-162.

FARB 72 - Farber, D. J., Larson, K., "The Structure of a Distributed Computer System - The Communication System", Proc. Symp. on Computer Communications, Networks and Teletraffic, Polytechnic Institute of Brooklyn Press, 1972, pp. 21-27.

FARM 69 - Farmer, W. W., Newhall, E. E., "An Experimental Distributed Switching System to Handle Bursty Computer Traffic", Proc. ACM Symposium. "Problems in the Optimization of Data Communications System", Pine Mtn. Georgia, Oct. 1969.

HAYE 74 - Hayes, J. F., "Performance Models of an Experimental Computer Communication Network", BSTJ, Vol. 53, No. 2, 1974, pp. 225-259.

KAYE 72 - Kaye, A. R., "Analysis of a Distributed Control Loop for Data Transmission", Proc. Symp. on Computer Communication Networks and Teletraffic, Polytechnic Institute of Brooklyn Press, 1972, pp. 47-58.

KONH 72 - Konheim, A. L., Meister, B., "Service in a Loop System", Journal ACM, Vol. 19, No. 1, 1972, pp. 92-108.

PIER 72 - Pierce, J. R., "Network for Block Switching of Data", BSTJ, Vol. 51, No. 6, 1972, pp. 1133-1145.

POTV 71 - Potvin, J. N. T., "The Star-Ring System of Loosely Coupled Digital Devices", University of Toronto, Computer Systems Research Group Report No. 7, 1971.

REAM 75 - Reames, C. C., Liu, M. T., "Design and Simulation of the Distributed Loop Computer Network (DLCN)", in Proc. 3rd Annual Symposium on Computer Architecture, Clearwater, Florida, January 1975, pp. 7-12.

SPAN 76 - Spang, III, H. A., "Distributed Computer Systems for Control", General Electric Technical Information Series Report No. 76CRD049, April 1976.

SPRA 72 - Spragins, J. D., "Loop Transmission Systems - Mean Value Analysis", I.E.E.E. Trans. Communications, Vol. COM-20, No. 3, 1972, pp. 592-602.

STRE 76 - Strevens, C. W., "Current Research in Computer Network", ACM Computer Communication Review, April 1976, Vol. 6, No. 2, pp. 13-40.

+ NEW LOOP
x DLCN
- PIERCE
* NEWHALL

TABLE 1

INTERARRIVAL RATE	DATA LINE USAGE	QUEUING TIME	TRANSMISSION TIME	TOTAL TRANSMISSION TIME	CONTROL LINE USAGE	
3600.	..025	19.42	51.36	70.78	0.020	+
1500.	.065	30.33	53.58	83.41	0,051	+
900.	.100	39.66	51.28	90.94	0.083	+
600.	.155	61.00	52.36	113.36	0.123	+
480.	.185	773.25	50.25	123.50	0.154	+
420.	.218	107.79	51.86	159.80	0.179	+
340.	.275	160.00	51.27	211.27	0.222	+
300.	.324	266.78	54.28	321.06	0.247	+
270.	.340	335.58	51.09	386.67	0.276	+
240.	.387	596.43	51.51	648.95	0.307	+
220.	.432	1134.11	52.77	1186.88	0.342	+
3600.	.056	2.10	58.60	74.50	0.[illegible]	x
1500.	.138	6.40	61.30	86.70	0.[illegible]	x
900.	.235	12.20	67.80	103.90	0.[illegible]	x
600.	.365	19.40	79.60	136.10	0.---	x
480.	.474	30.10	102.10	175.20	---	x
420.	.543	39.90	115.90	210.20	---	x
342.	.677	64.20	150.80	297.70	---	x
300.	.759	101.60	210.30	404.00	---	x
270.	.844	181.50	332.70	648.40	---	x
240.	.937	303.10	648.90	900.60	---	x
2700.	.098	10.90	104.30	115.20	---	-
1800.	.147	18.70	105.70	124.40	---	-
1200.	.200	27.90	105.80	133.70	---	-
900.	.293	47.10	105.00	152.10	---	-
720.	.367	69.10	105.00	174.10	---	-
600.	.430	74.90	106.00	180.90	---	-
540.	.479	119.10	106.20	215.30	---	-
480.	.513	148.40	103.40	251.80	---	-
420.	.633	215.60	110.50	326.10	---	-
360.	.717	257.70	107.60	365.30	---	-
330.	.762	360.90	102.80	463.70	---	-
300.	.801	587.20	103.50	690.70	---	-
270.	.935	1412.00	99.00	1511.00	---	-
2100.	.153	15.30	62.60	77.80	---	*
1500.	.183	21.10	73.30	84.40	---	*
900.	.242	38.60	62.40	101.00	---	*
600.	.328	75.50	62.20	137.70	---	*
480.	.378	135.20	63.30	198.50	---	*
420.	.424	283.60	62.90	346.50	---	*
360.	.487	611.60	63.80	675.40	---	*
330.	.518	3210.00	59.00	3269.00	---	*
300.	.511	6564.00	68.00	6632.00	---	*

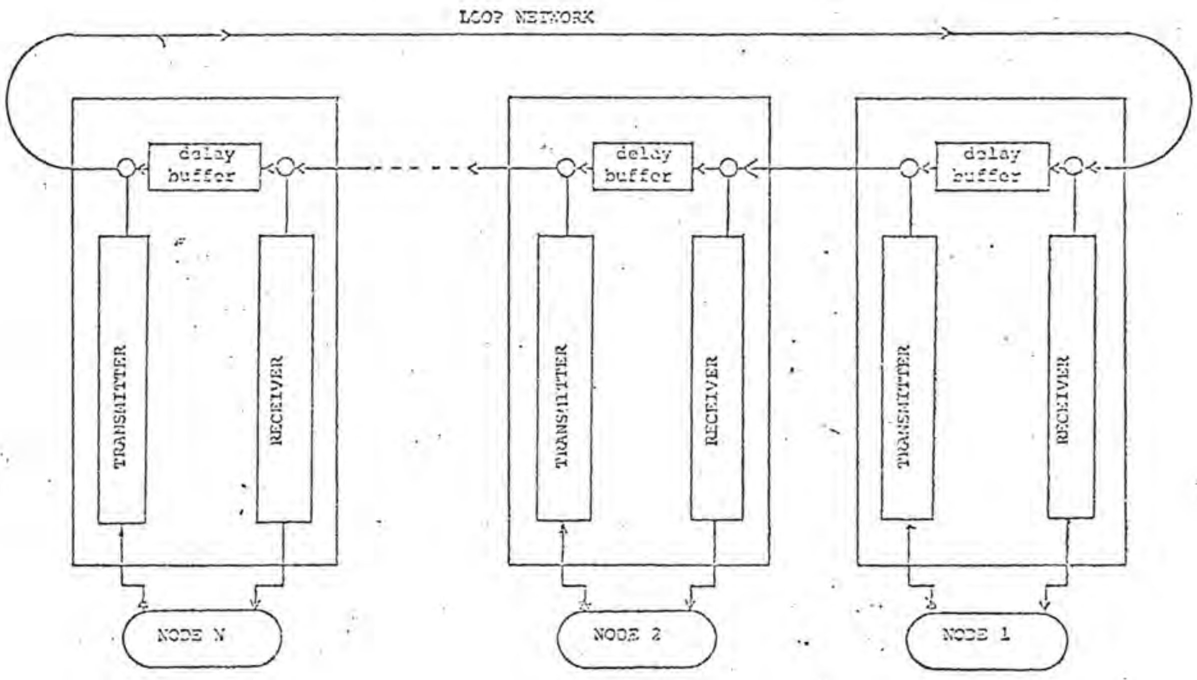


FIGURE 1

polling information	source node	dest. node	command

FIG. (2a)

polling information	node address	comm-and	node address	comm-and

FIG. (2b)

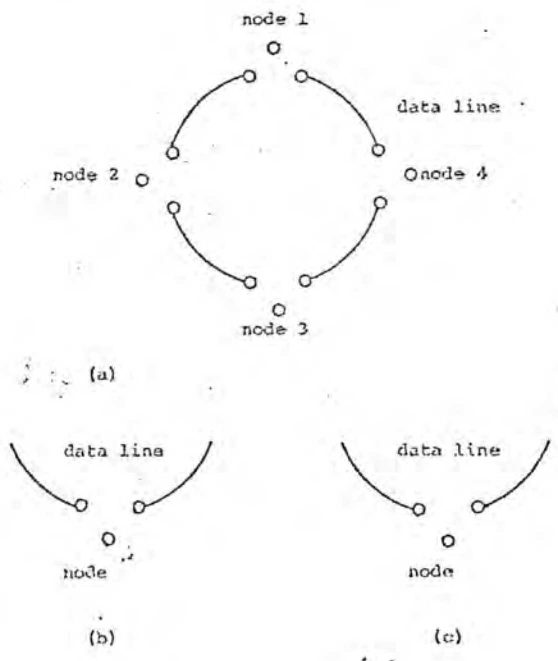


FIGURE 3

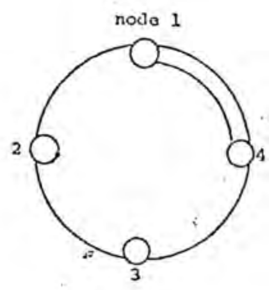


FIG. (5a)

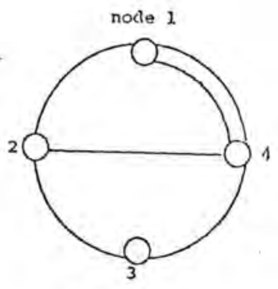


FIG. (5b)

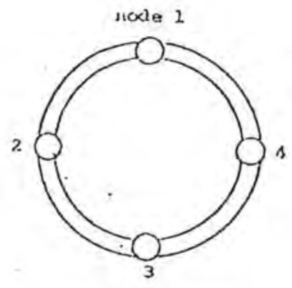


FIG. (5c)

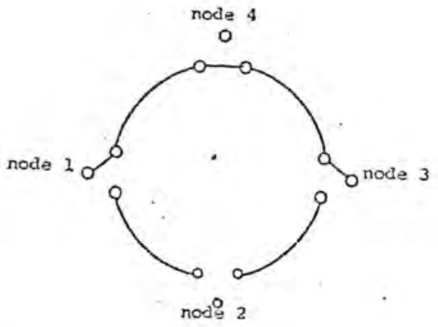


FIG. (4a)

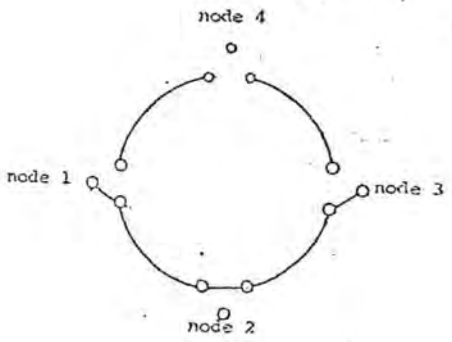


FIG. (4b)

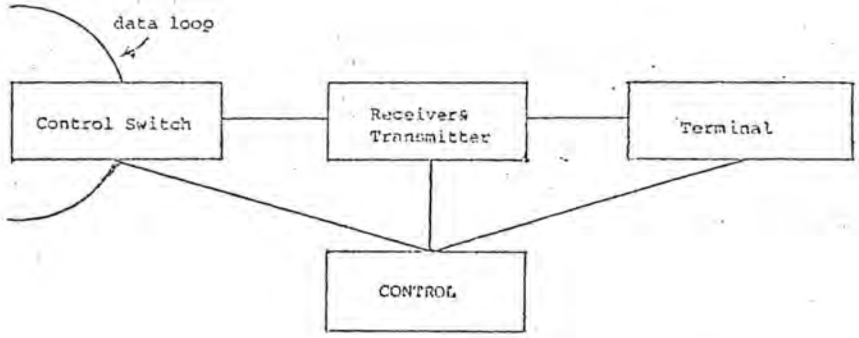
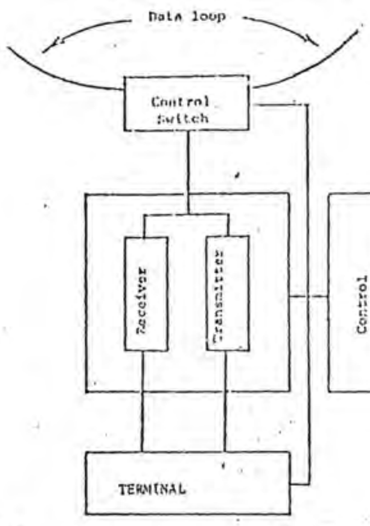


FIGURE 6



FIG_(7a)

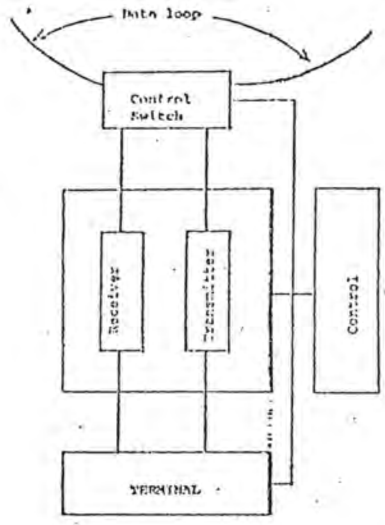


FIG. (7b)

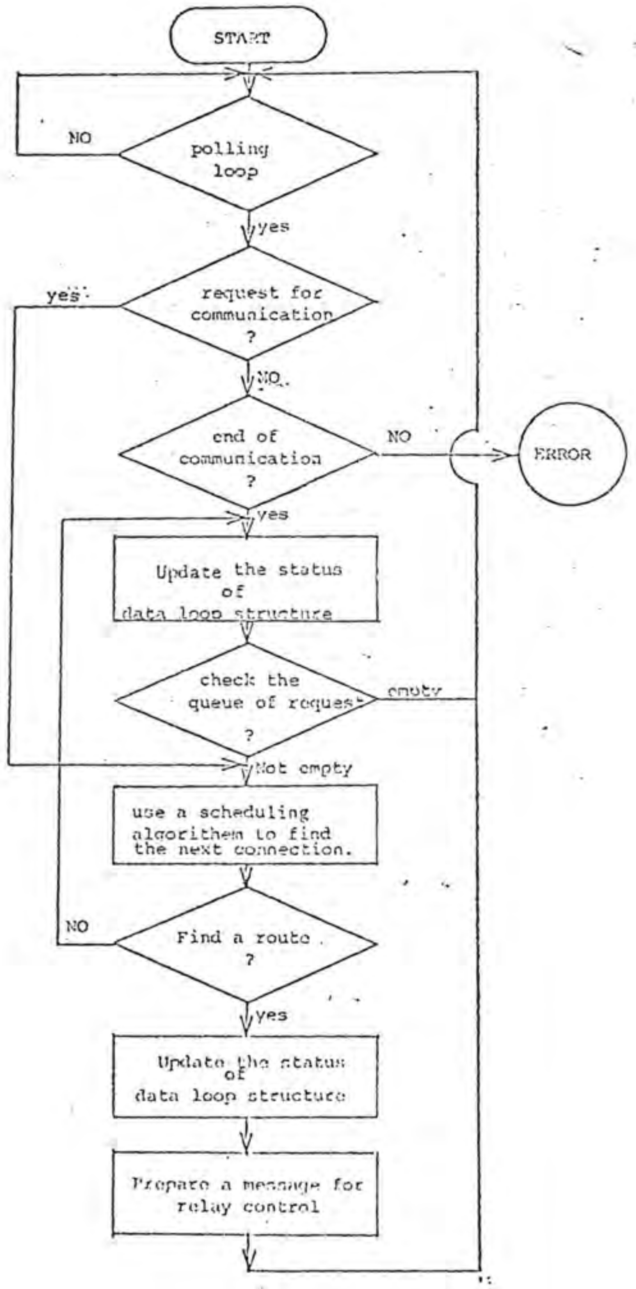
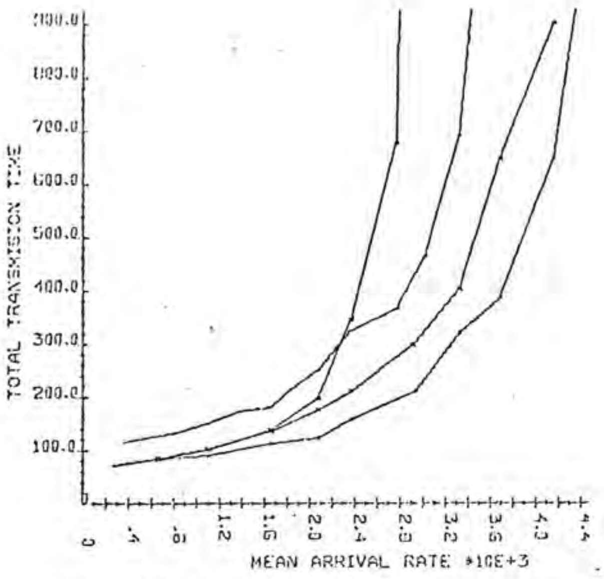
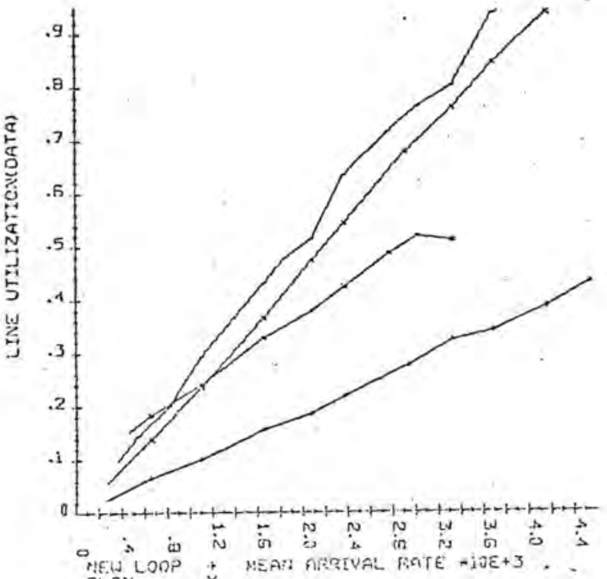

START
NO
polling loop
yes
yes
request for communication ?
NO
end of communication ?
NO
ERROR
yes
Update the status of data loop structure
check the queue of request ?
empty
Not empty
use a scheduling algorithem to find the next connection.
NO
Find a route ?
yes
Update the status of data loop structure
Prepare a message for relay control

FIGURE 8



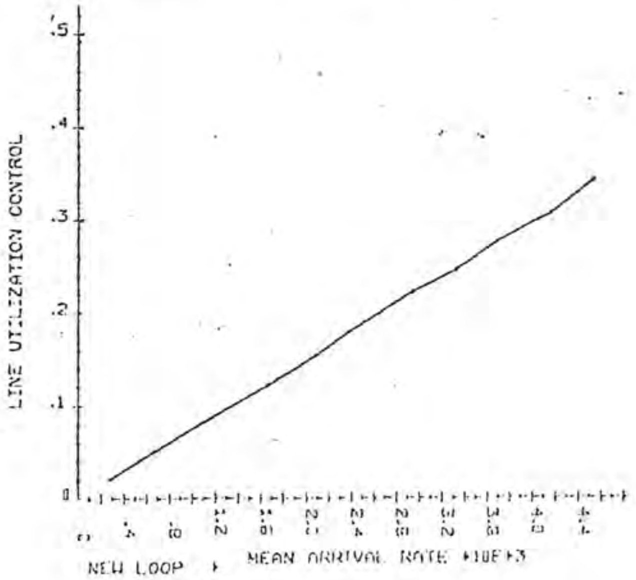
NEW LOOP +
DLCN X
PIERCE |
NEWHALL *

FIGURE 9



NEW LOOP +
DLCN X
PIERCE |
NEWHALL *

FIGURE 10



NEW LOOP +

FIGURE 11

Some Laws of Personal Computing

T. G. Lewis, Ph.D.
Computer Science Department
Oregon State University
Corvallis, Oregon 97331

(503) 754-3273

THE ORIGINS OF PERSONAL COMPUTING

In the beginning, man created pocket calculators to do rote arithmetic. The "four banger" solved a well known problem, e.g. addition, subtraction, multiplication, and division. Few people involved in the pocket calculator industry realized that pocket calculation was merely an initial thrust into the "computing for the millions" consumer market. Indeed, the millions of dollars made by this computer consumer product is paying for the development of more sophisticated devices we now call "personal computers".

Computing is the orphan of modern science, that is, computer science "ain't got no father". Isaac Newton, Albert Einstein, and others built the foundation of Physics. Freud gave birth to modern psychology, and biology has its origins with Darwin. But computing lacks a definite starting point. The works of Charles Babbage, and possibly Alan Turing, have little impact upon daily computing (some will argue that these two pioneers have everything to do with modern computing, but I speak of practical rather than theoretical computing). So where are the fundamental theorems of computing? Are there a set of "equations of motion" for programming?

This article contains ten empirical observations dubbed, "laws of personal computing". They are rules derived from personal experience with person computers in the real world of business. While many of the rules are controversial, I believe most can be proven to be true.

The first law of personal computing is of the form "action equals reaction". The law is derived by historical observation.

The first electronic computers were personal computers. That is, only a few programmers had access to the ENIAC, WHIRLWIND, and ATLAS. This one-on-one mode of interaction rapidly faded in favor of batch operation and multiprogrammed operating systems. Clearly, the shift away from one-on-one was the result of economic decisions. Large corporations poured large sums of money into data processing departments. For their investment, they demanded efficiency. Military installations required security and performance as their return on investment. Batch operation satisfied their demands.

Soon, however, users (programmers mostly) were able to show economics of scale and efficiency of operation by installing a limited form of interaction called remote-job-entry. RJE rapidly moved into timesharing with terminals because this increased the man-machine interaction. Finally, we have come full circle to dispersed, stand-alone, turnkey computers dedicated to a few users.

The key feature of the historical evolution of computing is "interactiveness". In fact, the more we are allowed to communicate with a computer system, the more we enjoy using the system (within limits), and the more "personal" computing becomes. This leads to the first law of personal computing.

[1] Personal Computing Equals Interactive Computing: The personalness of a computer system increases directly proportional to its interactiveness.

$$P = k I$$

where

P = personalness
k = constant of proportionality
I = interactiveness

Of course, we have not quantified (or defined) what P, k, and I really are in the formula.

In the following derivations, we determine at least a qualitative measure of several other variables in the laws of personal computing. In some cases we can represent the law with an equation, but in most cases this is not possible.

THE NEW ECONOMICS OF COMPUTING

Personal computing is governed by economics as much as by technology. Indeed, the directions taken by technology are governed by economics. Therefore, we must study economics in order to derive other laws of computing.

The concepts of programming, microprogramming, and chip design span the spectrum of software, firmware, and hardware. Why is it more suitable to microprogram the IBM 370/168 (model 370 hardware, model 168 firmware) and not microprogram the Intel 8080? Where is the trade-off between an "expensive" system and an "inexpensive" system when all features of such a system are considered?

A system designer can choose to build a cheap processor (like the 8080, say) and save money on production, design, and maintenance of the cheap processor. The same designer can elect to build a (expensive) sophisticated computer system and as a result increase the cost of hardware. Why would he choose to construct an expensive computer? The answer lies in looking at the total cost of a computer system. Lets take an example.

The Intel 8080 requires that the HL registers be loaded each time a memory reference is made. This feature is simple to implement and saves hardware dollars. However, every program that is written for the 8080 must pay the price of this simplicity. Typically, a macro called HL is used to relieve the programmer of this chore.

The Motorola 6800 includes a more sophisticated addressing mechanism using an index register for assisting in memory references. The addressability features of the 6800 often lead to 25% reduction in the number of instructions needed to perform the same function as performed by the 8080.

Both 6800 and 8080 architectures are more time consuming to program than the Texas Instruments 9900 chip due to the 9900's greater sophistication. Furthermore, the Microdata 32/S and Hewlett-Packard 3000 are stack machines supporting a high level language. Hence they are "easier" to program than any of the chips discussed above. But of course, the 32/S and 3000 are more expensive hardware machines than the chip machines.

Where is there a trade-off between complexity in hardware, complexity in firmware, and complexity in software? The trade-off is strictly economic, and leads to the second law of personal computing.

[2] Conservation Of Agony: The work expended to program a computer to solve a problem plus the work expended to construct the computer system remains constant for that problem.

$$W_S + W_H = C$$

where

W_S = software work,
W_H = hardware work,
C = constant for a specific problem.

Again, the numerical values for each of the quantities above are not easily determined. We suspect a curve similar to the one in Figure 1.

The second law of personal computing actually states that the problem solution remains at a constant level states no of complexity regardless of the system used to solve the problem.

The cost per unit of effort in building hardware may decrease (LSI chips), and the cost of programming may increase (due to unsophisticated microcomputers). The curve of Figure 1 traces the "best" point at which an economical blend of hardware and software meet. Therefore, in 1980, the most economical systems will be mainly firm-hardware (due to its low cost) and a small share in software (due to the conservation of agony).

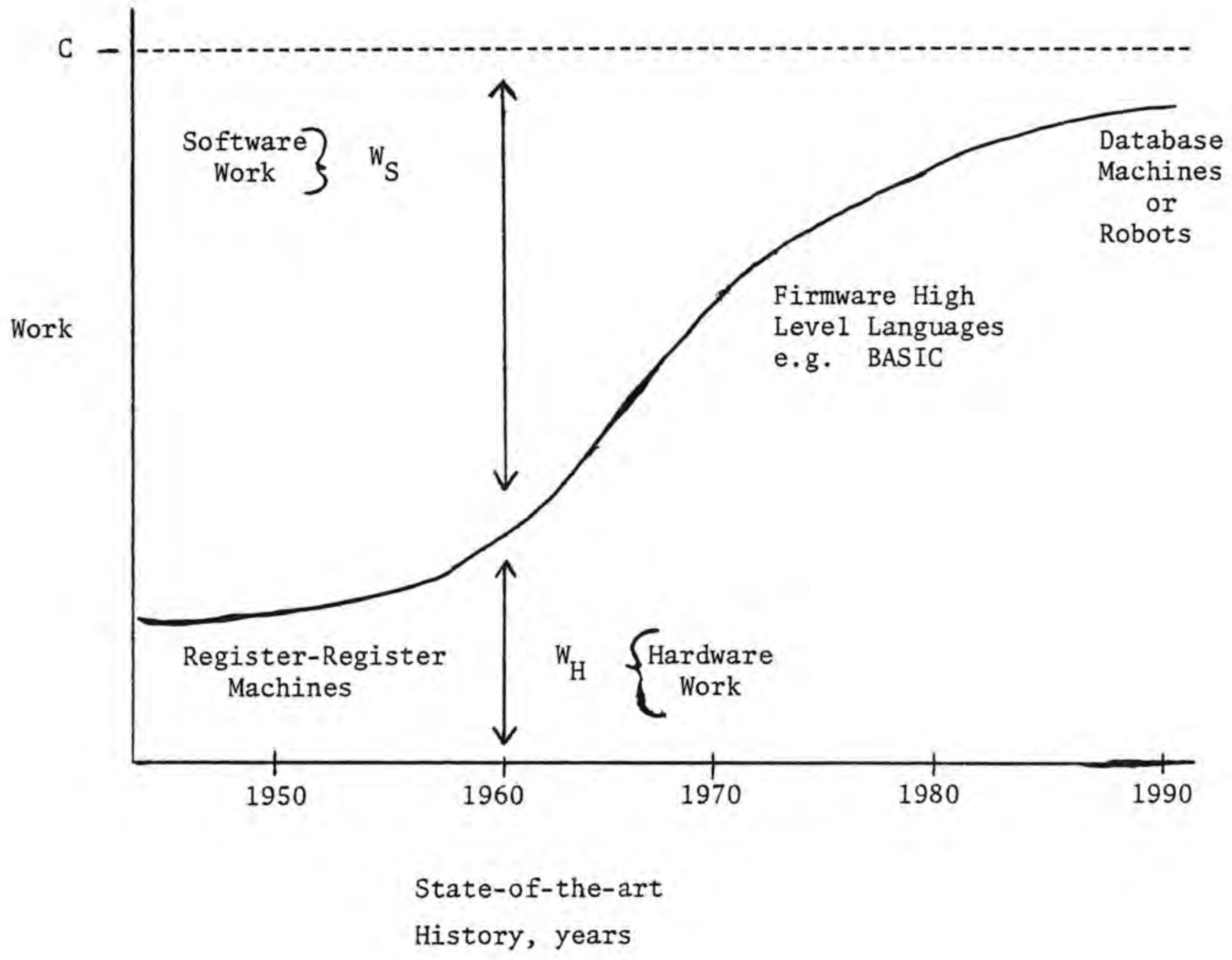


Figure 1. Point At Which W_H Equals W_S

The results of law two actually say something about the "power" of a computer system. We can state a macroscale formula for computer power as follows:

power = [(MIPS)(STORAGE CAPACITY)]/COST

This overly simple formula gives a broad measure of power in byte-cycles per dollar-second. Hence, increasing speed or storage capacity increases power. Conversely, decreasing cost increases power of a personal computer. For example, the Intel 4040 (four-bit pocket calculator chip) increased personal computing power because it was cheap even though it was slow and of low storage capacity.

Now, if we look at history once again, it is clear that an acceleration force is at work. Increasing capability in the past lead to increasing the number of applications in which a computer is useful. In turn, the increased use of a computer system in new applications results in increased sales. The sales stimulate mass production and further cost reductions. The end result is decreased unit cost of the computer system.

We can demonstrate this counter-intuitive notion as follows. In the mid sixties, processor speed increased dramatically. This increased capability motivated timesharing of the central processor. The support of many terminals reduced the cost per terminal, and in the final analysis, the cost of the unit of computation. Increased capability lead to reduced cost of computation.

In the seventies, the capacity of storage is increasing dramatically. We are witnessing a surge of activity in database applications with the corresponding decrease in cost of storage. In short, we are witnessing the third law of personal computing in action.

[3] As The Power Of A Personal Computer Increases, Its Price Decreases.

The equation for the exact form of diminishing cost expounded in law three is highly complex. To derive the equation would require a model of the economy, a predictive model of advances in technology, and a psychological study of people's acceptance of computerization of sensitive applications (making medical diagnosis, for example).

The third law deals only with hardware capability. Earlier, we stated that hardware capability plays a decreasingly important role in personal computing. Indeed, the effects of the third law of personal computing are rapidly diminishing due to the fourth law.

[4] Software Is Hard; Hardware Is Soft: It is economically more feasible to build a computer than to program it.

It is economically easier to design, implement, and mass produce a machine like the Intel 8080 or IBM 360 than it is to design and implement an operating system, compiler, or sophisticated application program. The cost of a chip may run to $250,000 when design and initial production is totaled. The cost of firmware BASIC may not exceed $100,000 (many often do, however), but the auxilliary cost of documents, service, training, and marketing may exceed one million dollars.

A company contemplating a new hardware architecture is gravely penalized for making radical changes to the instruction set of their existing computer. Is it not to be expected that the IBM 370 is only an evolutionary departure from the IBM 360? Why is the Z80 processor nearly as successful in the market place as the 8080?

The high cost of programming as opposed to the cost of a chip is reversing the traditional roles of software and hardware. In the future, more emphasis will be placed on the software and less emphasis will be placed on the machine architecture. Indeed, much of the current software will become "hard", by placing it in firmware and distributing it in hardware ROMs.

One result of law four is the following corollary. Corollary A states the rule that governs pocket calculators, today.

[A] Programs and data should be shared; but hardware should be replicated.

The only item in a computer system that must be shared, from a technological standpoint, is data. Common access to information stored in a database may be logically justified by an application. Whether the access is done via timesharing or dispersed processors is immaterial. Also, whether the data is copied for transmission, or the program that intends to process the data is copied for transmission to the database machine, is again immaterial.

The computer business has been over enthusiastic about timesharing in the past. We must recall that timesharing was invented to lower the cost of hardware. Now that hardware is no longer the major cost item in a system, timesharing is not justifiable in most cases. In fact, timesharing failed.

Timesharing failed because people couldn't understand it. Only computer experts are able to use MULTICS, VM/370, and other extremely capable timesharing systems. The average person will not tolerate JCL, telephone lines, computer jargon, and unreliable central computers that loose their files. In short, time-shared computers are useless due to their prerequisite of knowledge.

The computer utility concept of the late sixties failed because of the lack of expertise on the part of the users. The high level of sophistication needed to use a utility doomed it to failure. It also put a bad name on personal computers.

In effect, the "guilt-by-association" syndrome plagues personal computing, today. Myths (its too complicated), training (what is a byte?), and service (how do I get statements printed?) are three of the remnants of the computer utility that have turned people away from computing.

We can now state a conclusion called the fifth law of personal computing.

[5] Knowledge Costs More Than Software And Hardware: The usefulness of personal computers increases inversely proportional to how much people must know in order to use them.

The lesson is clear: any consumer product that is successful, must be simple. The pocket calculators that solve known problems (arithmetic) are successful. The pocket calculators that solve unknown or unrecognized problems are failures (the HP-85 for financial analysts solves an unknown or unrecognized problem).

The facts of life are even more severe for computers sold to the consumer market. The final economic law succinctly summarizes the fickle buyer's attitude.

[6] The Color, Shape, And Size Of A Personal Computer Is More Important To A Buyer Than What Is Inside Of It.

Once the personal computer system overcomes all other economic obstacles it must be packaged and maintained by a reputable service organization.

Packaging - eliminate buttons, switches, and knobs. The manuals must reduce jargon, and the software must be tailored to a particular industry.

The SOL-20 system from Processor Technology and the NOVAL from Gremlin Industries are vivid examples of packaging in the personal computer hobby market. Datapoint, Wangco, and Basic-Four demonstrate the law with tailored software packages for small businesses.

Service - fills the gap between the user's knowledge, and the personal computer's lack of capability. Service rescues the user when the personal computer cannot repair itself. It is service that counts when the manuals do a poor job of explaining a feature of the system. Finally, service is performed by humans, and so far, humans understand other humans vastly better than they understand a machine.

We can now turn to some interesting examples that lead to the final laws of personal computing. In particular, these laws impact directly on the majority of computer experts engaged in applications implementation.

IMPLICATIONS OF INTERACTIVE-NESS

The first law of personal computing equates "interactiveness" with "personalness". This means that in order to achieve a high degree of interactive computing, the personal computers of the future must be oriented toward languages and systems with a high degree of interpretation. Compiler languages, for example, have been shown to require from three to ten times as much effort to implement a given program as required to implement the same program using an interpreter.

It is little wonder that BASIC has achieved the title, "language of the masses". It is a simple interpretive language easy to implement on a modest processor. Unfortunately, it is extremely inappropriate for major applications requiring typical business data processing.

[7] BASIC Is To Personal Computing, As Sign Language Is To English.

BASIC is the "pig latin" of programming languages. BASIC programs are easy to write, naturally, but like pig latin, they are difficult to understand, and provide inadequate control of a personal computer system. Few dialects of BASIC permit indentation, structuring, comments (without memory penalty) or error control and recovery. Here are a few objections to BASIC as a serious, professional implementation language.

a) poor error recovery facilities - e.g. the application program must be capable of detecting file access errors, etc. and then calling an exception handling routine.

b) no dynamic overlapping or memory mapping of programs too large to fit in main memory.

c) restricted data structures - e.g. linked lists, trees, dynamic memory allocation for data, mixed data types.

d) limited user prompting - e.g. forms handling, menus, cursor control, scrolling, audio signals.

e) inadequate software security and protection - e.g. file security locks, interlock mechanisms for shared files, inadequate source code shielding.

f) slow execution due to poor interpretation.

g) inadequate primitives for standard data processing - e.g. no sorting, file access constructs, forms handling for report generation, or communications access constructs.

In short, BASIC is useful for beginners developing small programs for an unsophisticated application, or for programs that will be thrown away rather than modified.

The area of system control is no better off than the system languages area of personal computing. At least BASIC is partially standardized and widely known. Operating systems, on the other hand, have no consistent basis to begin with. Indeed, we question the utility of an operating system in interactive computing. This is pointed out in the eighth law of personal computing.

[8] An Operating System Is A Feeble Attempt To Include What Was Overlooked In The Design Of A Programming Language.

This heritical notion is fully obvious in systems employing interpretive BASIC to the hilt. The Wangco, Tektronix 4051, and similar small scale interpretive BASIC systems have no visible operating system. All commands normally associated in traditional operating systems are put into extended BASIC in these personal computers. In general, interpretive systems (and thus interactive systems) have no need for an operating system.

In future personal computers it is likely that a network of loosely coupled processors will communicate data and programs to one another. In such a network, concurrent processes will be allowed, and often compete for limited resources. In this situation, the synchronizing primitives of today's operating systems will be migrated to hardware (or firmware) and not be of concern to the language interpreter.

THE ULTIMATE LAWS

We have covered the econo-technical motivations for personal computing and stated eight laws along the way. In the final analysis we can derive two ultimate laws of computing used (knowingly or otherwise) by computer manufacturers.

[9] The Ultimate Personal Computer Is A Robot: The goal of personal computing is to reduce the differences between humans and computers.

In effect we are striving to make personal computers do what people can do, only faster, more accurately, and cheaper. We seek a partnership with personal computers akin to the symbiosis between humans and household pets.

A faster personal computer allows us to process census information in 2-3 years instead of 15 years. Speed is essential in a lunar landing, and so is accuracy.

An air traffic control computer is much more accurate than a human operator. The result is safer air transportation for people.

A computer that can do your job faster, more accurately, and cheaper than you can do your job is a threat to you. In fact, a cheaper computer is threatening jobs everywhere today. This aspect of computing is being ignored by computer scientists because it represents an undesirable aspect of computing. None the less, we must face this problem before the ultimate law is enacted.

[10] Knowledge Is Power: Information is the fabric of knowledge, and he who controls it, wields power.

Good versus evil. While personal computers are fast, accurate, and cheap, they also cause high speed propagation of errors, speed-of-light crime, and loss of life when they fail.

Politicians are able to push a button and disseminate campaign propaganda to the millions. Factories can replace entire vocations by automating production. Financial institutions are at the mercy of their data processing centers.

Is it possible that personal computing will lead to a caste society? When all menial tasks, management decisions, and control of production has been turned over to computers, what will mankind do? Will the elite of the future be those who can create, invent, entertain, and be humane, while everyone else is relegated to welfare?

The laws of personal computing are not only important to computer scientists, but also to society as a whole. Perhaps there is a place today for the futurologist, that is, a philosopher of computer science. I wait.