SEL-74-015

Computer System Performance Measurement: Instruction Set Processor Level and Microcode Level

by

Liba Svobodova

June 1974

Technical Report No. 66

This work was supported by the Joint Services Electronics Program: U.S. Army, U.S. Navy, and U.S. Air Force under contract N-0001 4-67-A-0112-0044

DIGITAL SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES

STANFORD UNIVERSITY • STANFORD, CALIFORNIA

SEL 74-015

COMPUTER SYSTEM PERFORMANCE MEASUREMENT:

INSTRUCTION SET PROCESSOR LEVEL AND MICROCODE LEVEL

by

Liba Svobodova

Technical Report No. 66

June 1974

DIGITAL SYSTEMS LABORATORY

Department of Electrical Engineering Department of Computer Science

Stanford University

Stanford, California

This work was supported by the Joint Services Electronics Program: U. S. Army, U. S. Navy, and U. S. Air Force under contract N-00014-67-A-0112-0044.

ABSTRACT

Techniques based on hardware monitoring were developed to measure computer system performance on the instruction set processor level and the microcode level.

Knowledge of system behavior and system utilization at these two levels is extremely valuable for design of new processors. The reasons why such information is needed are discussed and applicable measurement techniques for obtaining necessary data are reviewed. A hardware monitor is a preferable measurement tool since it can trace most of the significant events attributed to these two levels without introducing any artifact.

Described hardware monitoring techniques were implemented on the S/370 Model 145 at Stanford University. Measurements performed on the instruction set processor level were concerned with determining execution frequencies on individual instructions under normal system workload. The microcode level measurements measured the number and the type of S/370 Model 145 microwords executed in the process of interpretation of an individual S/370 instruction and the average execution time of each such instruction.

Implementation of each technique is described and the results based on the outcome of performed measurements are presented. Finally, effectiveness and ease of use of the discussed techniques are considered.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

TABLES

FIGURES

I. INTRODUCTION

Machine instructions provide a means of control a user can exercise over his machine. The instruction repertoire is an attribute of a computer family, but its implementation may differ widely among models of one family. How individual instructions from an available repertoire are used is a function of the processing environment of a computer installation. We say that each computer installation processes its own instruction mix where the instruction mix reflects the particular environment. Both the instruction set composition (instruction repertoire) and the technique of implementation affect the suitability of a particular computer to various processing environments. This suitability can be evaluated in terms of programming effectiveness and efficiency of execution, the latter being one of the aspects for evaluating performance of computer hardware.

One of the first measures of computer performance used for the purpose of selection of the best among several machines (selection evaluation) was the execution time of a single instruction, usually ADD instruction [LUCA 71]. Such a method was soon recognized as inadequate and was replaced by a method based on an instruction mix. An instruction mix was a list of instructions weighted according to their frequency of execution. The time needed to execute a particular instruction mix was then taken to be the direct measure of performance or it was combined with other computer system attributes (I/O speed, data path width) to calculate a 'Figure of Merit' [KNIG 66, DRUM 69]. Optimally, the instruction mix used in evaluation should represent the actual instruction utilization under the load the selected computer will have to handle.

However, statistics about instruction usage were rarely available and instruction mixes for performance evaluation were constructed according to what was thought to be a typical business or a typical scientific mix. Even so it should not be disregarded altogether, the instruction mix as a performance evaluation tool has a number of drawbacks and is today rated as entirely insufficient. This fact by no means implies that statistics on instruction usage have no application. There are fields of computer engineering where such statistics are extremely valuable. Some of the candidate areas are explained below.

Design of new processors. A broad instruction repertoire increases the processor's generality and applicability to several different processing environments, but the cost of implementation grows also. Some or all instructions can be implemented entirely in hardware, microcoded, or interpreted in software. The implementation technique has a direct impact on the cost of the processor's hardware and the execution speed. Hardware/firmware/software tradeoffs can be resolved effectively only when enough facts are known about the frequency of usage of individual instructions.

Emulator selection. The process of upgrading to a computer system with a different instruction repertoire than that of a system being replaced can be significantly simplified if the application software can be emulated rather than reprogrammed. The efficiency of an emulator may depend greatly on the instruction mix the emulator has to process.

Processor selection. Knowledge of the instruction mix may help in the first-run selection of computers before starting detailed (and costly) performance analysis. For example, an installation that is

known to use floating-point arithmetic in less than 1% of the total mix most probably will not be interested in purchasing an optional floating-point hardware unit.

Compiler evaluation. Compilers generally utilize only a subset of available instructions. A good compiler should take advantage of the flexibility and power of some special instructions. To what extent such attempts are successful can be found out from the object code instruction mix.

Generation of 'typical' instruction mixes. Instruction mixes classified according to what application they represent can be used as standards for comparing processing environments of different installations.

- Gaining more insight about the computer system environment and operations. Instruction utilization data reflect some characteristics of programs run on a particular computer. Some unexpected discoveries may emerge. Perhaps decimal arithmetic is used too heavily where conversion to binary numbers would be more appropriate, or some special and efficient instructions are never used merely because programmers have not been sufficiently trained to appreciate such extra features.

In this study we examine the degree of applicability of instruction usage statistics to design and implementation of processors and emulators. Attention is then focused on methods of acquiring such statistics.

The next section, Section 2, establishes basic terminology for the rest of the paper. Section 3 discusses various aspects and tradeoffs inherent in the design of processors and emulators. Section 4 summarizes techniques of collecting instruction usage data. Section 5 then describes

two ways a simple hardware monitor can be employed to count occurrences of S/360-370 instructions. Section 6 is concerned with analysis of S/370 Model 145 microprograms. Finally, Section 7 summarizes the work presented in this paper and suggests the direction of further research.

II. BASIC NOTIONS

The term instruction mix is well known and widely used, but not well defined. Many instruction mixes do not include every individual instruction implemented on a machine being evaluated, but concentrate on a subset of general instructions that are likely to be contained in instruction sets of machines of different families. Some mixes are constructed out of groups of instructions that perform similar functions. For example, an instruction mix for IBM S/360 could specify the weight for a fixed-point add function without distinguishing between RR and RX types of operation.

Let s be the number of different instructions (opcodes) implemented on a machine, where each opcode is assigned a unique integer k such that $1 \leq k \leq s$. Let N_k be the count of occurrences of k-th opcode in N instructions. Then $f_k = \frac{N_k}{N}$ is the frequency of occurrence of the k-th opcode. There are many possible mappings of the given set of s opcodes onto the set of first s positive integers. We will consider only two of them. The first mapping assigns integers to individual opcodes in such a way that $c_k < c_{k+1}$ where c_k is the value of the k-th opcode. Since some opcodes are not assigned to any instruction, the value of the k-th opcode is not always equal to k. The function f_k defined on a set of opcodes under this particular mapping will be called the Instruction Utilization Function (IUF). The second mapping orders opcodes by their frequencies of occurrence, such that $f_k \geq f_{k+1}$ for all $k=1,2,\ldots,s$. Then let the Instruction Frequency Distribution (IFD) be the function $F(q) = \sum_{k=1}^{q} f_k$ defined on the set of s opcodes under this second mapping. It should be noted that while the first mapping is unique for all machines

using the same instruction set (IUF may be therefore used for a comparison study), the second one may vary from one installation to another and from one application to another.

We have to distinguish between two cases of instruction utilization. Static utilization is determined by the structure of a program(s). Here f_k is the frequency of how often each opcode is written. Dynamic utilization represents how many times each instruction is executed. The static IUF for a specified program can be obtained by hand-analysis of its code. Dynamic instruction utilization is environment-dependent. It is a function of data operated upon by programs being analyzed and as such it cannot be computed exactly unless the values of all operands are explicitly known. This matter becomes even more complex when not only a single program (or a set of programs) but the entire system is studied. The only way to obtain the function of dynamic instruction utilization is to monitor the system under its normal operating conditions.

Another parameter of interest is the distance of two instructions I_i and I_j. It can be defined as either the average number of instructions or the average number of CPU cycles executed after an occurrence of I_i and before the first following occurrence Of I_j. Note that distance (I_i, I_j) is different from distance (I_j, I_i).

III. ROLE OF INSTRUCTION UTILIZATION STATISTICS IN THE DESIGN AND IMPLEMENTATION OF AN INSTRUCTION SET PROCESSOR

Conventional computers can be described as composed of three basic units: the Central Processor Unit (CPU), the main memory, and the Input/Output (I/O) facilities. A great part of the CPU elements are manipulated by externally supplied machine instructions, or for short instructions. The collection of instructions directly interpretable by the CPU is called the instruction repertoire or simply the instruction set. The portion of the CPU hardware that supports operations specified by machine instructions shall be termed the instruction set processor. Bell and Newell proposed the instruction set processor (ISP) description scheme for specifying a set of operations and rules of interpretations incorporated into the CPU architecture [BELC 711. We shall not be concerned with defining the exact data and control flow for an instruction set processor as it is done under the ISP descriptive scheme. Our interest lies in determining a set of operations to be included in a machine instruction set and the techniques of implementation of an instruction set processor.

3.1. Instruction Repertoire: The Structure and the Power

Since the beginning of the history of computers, the specification of a machine instruction set has usually been a major issue of a computer design project. Bell and Newell in [BELC 71] give an excellent overview of various dimensions of computer architecture. The subset of these dimensions that describes the structure of an instruction set is presented in Table 1 together with some of the many possible design alternatives.

To select a set of operations executable as a single machine instruction and to decide on the exact instruction format, a designer has

TABLE 1: Dimensions of an instruction set processor.

WORD SIZE	OPCODE FIELD SIZE	OPCODE INTERPRETATION RULES	BASE	ADDRESSES PER INSTRUCTION	PROCESSOR CONCURRENCY
8 bits	fixed		binary	0 address (stack)	serial by bit
12 bits	variable		decimal	1 address	parallel by word
16 bits			character	1+x (index) address	multiple instruction stream
24 bits				l+g (general register) address	multiple data stream
32 bits				2 address	1 instruction buffer
48 bits				3 address	n instruction buffer
64 bits				n+l address	look-aside memories
					pipeline processing

to evaluate a number of tradeoffs. The word size often represents a serious constraint. When the word size is small, the individual fields of an instruction have to be kept as small as possible. The operation code, or opcode, specifies the type of a function carried out by an instruction (load, store, add, etc.). The shorter the opcode field, the fewer distinctive opcodes are available; this limits the power of an instruction repertoire.

The power of an instruction set can be defined as the ease of implementing various programming tasks. We could possibly use the term effectiveness instead. It is determined by two factors. The first factor is what operations can be performed with a single instruction. For example, a multiply operation on some small machines such as the DEC PDP-8 has to be programmed explicitly using add and shift operations; n+l address instructions can branch in addition to executing a function specified by the opcode. Some machine instruction sets might include instructions for such complex functions as search or sort. The second factor stands for how many and how large the pieces of data are that can be moved between the processor and the main memory by a single instruction. Data that are longer than what can be handled by a single instruction have to be broken into n fields and the requested operation performed in n steps, while some control information is passed from one step to the next. For example, to perform a long add operation, the overflow from step k is the carry into step k+l. Multiple operand address instructions eliminate the necessity of having to move operands and results between local processor storage and main memory explicitly. An example is given below.

Goal: Add OPERAND1 and OPERAND2 and store the result in RESULT

One address per instruction:	L	OPERAND1
(IBM S/360)	A	OPERAND2
	ST	RESULT
Three addresses per instruction:	A	OPERAND1,OPERAND2,RESULT
(MIDAC [BELC 711)		

The problem of a short opcode field is quite common to minicomputer architecture. Some minicomputers such as the PDP 8 or HP 2116 use the address portion of an instruction word of those instructions that do not reference the main memory as additional opcode subfields. This approach greatly enhances the power of an instruction set while only a few bits are required for the opcode field. Another scheme, proposed but not yet implemented, is based on conditional interpretation [FOST 71A]. It utilizes the fact that each instruction has only a very few likely successors. As an example, 'Load accumulator' operation is not going to be followed by 'Clear accumulator.' Each instruction can be then encoded as the k-th successor of the instruction immediately preceding in execution. Under this scheme, the size of the opcode field could be greatly reduced, but only at the expense of some special decoding hardware.

3.2. Utilization of the Information in Opcodes

Extra power of an instruction repertoire can be provided only through increased cost of computer hardware together with extended design effort. When the instruction set is not fully utilized, both the design effort and the hardware investment are partially wasted. The maximum utilization of the information in opcodes is achieved if all instructions are equally likely. Two measures of opcode usage were proposed in [FOST 71B]. The first measure I is the average number of bits of information

contained in each opcode. It is obtained as $I = \sum_{i=1}^{S} p_i \log_2 p_i$ where p_i is the probability that the i-th instruction from the instruction set S of size s is used. The maximum is reached when $p_i = \frac{1}{s}$ for all $i=1,2,3,..s$, and $I_{max} = \log_2 s$, but such a situation is fairly improbable.

The process of coding a set of tasks to run on a particular machine maps tasks into, but not necessarily onto, the set of machine instructions. This means that some instructions are perhaps never used. Let the effective instruction set S_e of a specific program be defined as the set of all instructions used at least once in that program. The second measure then examines the effort necessary to recode such a program onto an instruction set $\bar{S}$ such that $\bar{S} \subset S_e$ and can be expressed as $g(\bar{s})=1-F(\bar{s})$, where $\bar{s}$ is the size of the set $\bar{S}$ and $F(\bar{s})$ is the IFD defined in Section 2.

The two measures were applied to the analysis of instruction usage in both handcoded programs and object coded programs on the CDC-3600 computer at the University of Massachusetts. Table 2 shows what percentage of program code would have to be rewritten if the original set of 142 CDC-3600 instructions were limited to include only the $\bar{s}$ most popular ones.

TABLE 2: Program portion that has to be rewritten if the original set of 142 instructions is limited to $\bar{s}$ most popular ones.

$\bar{s}$	Hand-coded programs	Compiler output (object code)
64	2%	0%
32	10 - 16%	0 - 3%

Statistics on static (written) usage of opcodes are not sufficient--some instructions are executed with much greater relative frequency than they appear in a program listing (branch instructions, for example). Not only the reprogramming effort, but a possible increase in execution time has to be analyzed when such reduction of an instruction set is considered.

3.3. Implementation Techniques

Thus far we have discussed the structure of machine instructions in the context of the instruction set power and the information contained in each instruction. Next we shall concentrate on how a specified instruction set can be implemented, or in other words, we shall discuss various instruction set processor organizations.

An instruction set processor is an assembly of hardware functional units (registers, adders, shifters) interconnected with data busses and control lines. Execution of a single instruction may require assistance of several such units, and some of these units may be used more than once. As an example, a multiply operation can be performed as a series of add and shift operations. Sequencing control can be made entirely of logic circuits, and such processors are then called hardwired, or it may be supervised by stored programs. Such programs are transparent to a programmer using the machine instruction set. They are written in a more primitive instruction set called microcode instruction set with each instruction termed a microinstruction or a microword. Processors implemented in this way are known as microprogrammed processors. Microprograms reside in control memory (control store) and are collectively referred to as microcode. Control store may be either read only or may provide write capacilities in addition to read capabilities (writeable control store).

Processors with writeable control store are called microprogrammable processors.

The third level of implementation is then software interpretation. Frequently used operations can be coded and catalogued as special subroutines, a popular term for which is macroinstructions. Macroinstructions eliminate the necessity of explicit coding and are thus a convenient tool for a programmer. They must not be mistaken for a subset of the machine instruction repertoire, but they can be viewed as its extension.

The three levels of instruction set processor implementation are illustrated in Figure 1. The main tradeoff here lies in design and implementation cost versus execution speed. Hardwired operations are fastest, but a completely hardwired processor might be too complex and too costly. Further, once logic circuits are assembled together, there is no way to modify the instruction set. Microinstructions perform very simple operations and their execution does not require an extensive hardware control. Simpler hardware is only a tradeoff for a reduction in the speed of execution, since several microinstructions often have to be executed per single machine instruction. But this factor is becoming less and less significant as the speed of control memories increases. Both approaches may be combined in the organization of a single machine. That is, the crucial instructions may be hardwired while the less frequently used ones are implemented as microroutines.

In recent years, there has been a growing trend toward implementing an instruction set processor in microcode. As a result, a great part of the effort associated with hardware design has been shifted to the area of microprogram development. This shift is even more enhanced by the fact that many of today's control memories are dynamically writeable; that is,

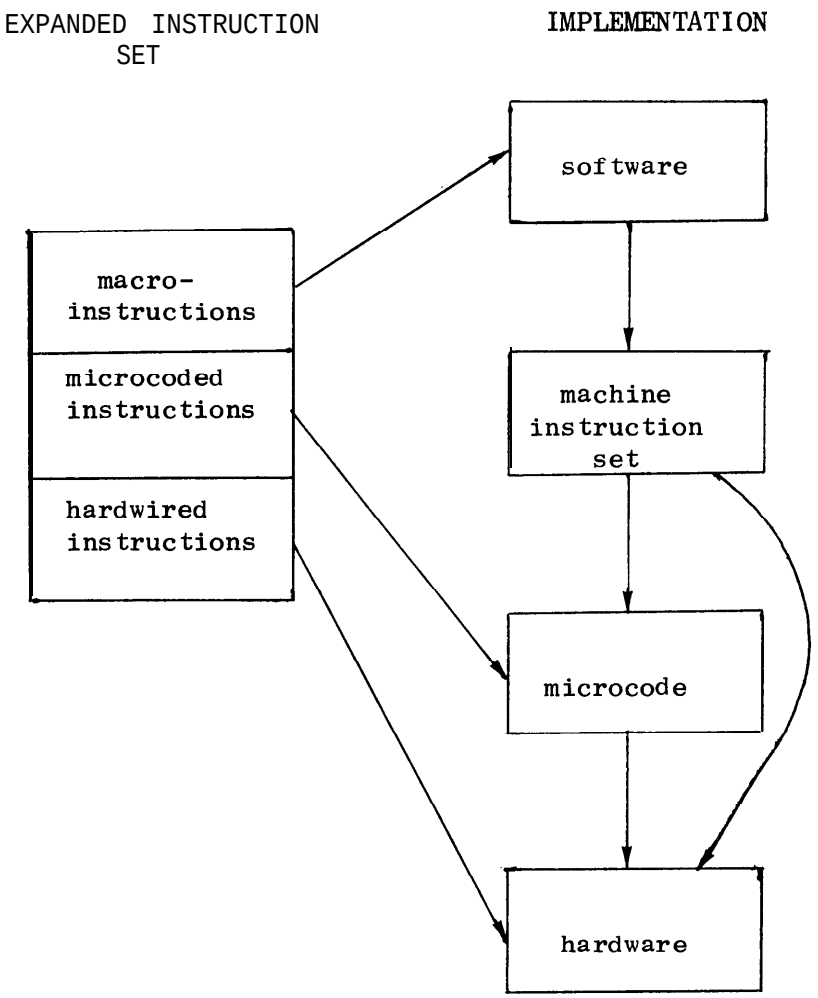


Figure 1: Levels of implementation of an instruction set processor.

their contents can be changed under control of a program. Processors with dynamically writeable control store are referred to as dynamically microprogrammable processors. Some dynamically microprogrammable processors can execute microprograms only if those reside in the control store [COOK 701. Microprograms that are not found in control memory have to be first swapped in (from main memory or some auxiliary storage). In other cases, a dynamically microprogrammable processor can execute microinstructions directly from main memory, at the expense of the execution speed [TUCK 711. In both cases, if a requested microprogram is not found in control storage, its execution requires more CPU time. A dynamically writeable control store facilitates the opportunity to change dynamically the entire instruction repertoire. The exact composition of the instruction set and the control store residency can then be optimized to fit the immediate needs of each individual installation. The optimization process has to be based on some reliable data about opcode utilization. Some of these problems are similar to those encountered on the operating system level. For example, the most frequently used OS/360 SVC routines are resident in the main memory; the rest are transient. If one or more transient routines arecalled too often, system overhead might become too large. To find out the real cause of such overhead, the system has to be monitored. The same is true when dealing with microroutines, but it is not always possible to use the same measurement methods.

3.4. Microcode Level

We have discussed microprogramming as one of the possible means of implementation of an instruction set processor. Indeed, the original purpose of microprogramming was to aid a computer designer to eliminate the

randomness of control logic in an instruction set processor [WILK 51]. But the concept of microprogramming has a much wider area of application. Frequently executed tasks that require a considerable amount of CPU time can be written directly in microcode to increase system throughput. System diagnostic routines [JOHA 711, operating system functions, special functions for performance monitoring [SAAL 72, ROBE 72] are all acknowledged candidates for microcode implementation. Computers can be also microprogrammed to execute higher level languages directly without the need for a compiler.

In the previous section we suggested that a microprogrammable computer can support more than one instruction set. This means in turn that such a computer can execute programs written for other, different machines. The process just described is known as emulation. Salisbury presented a broad overview of microprogramming in the context of emulation [SALI 73A]. A truly universal processor should have the capabilities to emulate a wide range of machines. The prime supposition here is a great flexibility of its microinstruction set.

Specification of a microinstruction repertoire represents a problem analogous to the one encountered on the instruction set processor level. In fact the microinstruction set could be considered the basic instruction set of a microcoded machine, and the microcoded processor, the interpreter of the machine instruction repertoire. Such a concept seems to provide a better description, especially for machines that do not have what can be called their 'own' instruction set. The term interpreter has indeed been used by designers of the B1700 [WILN 72] which is possibly the most flexible processor ever designed.

Table 3 summarizes instruction processing capabilities of various processor types. Emulators and interpreters actually belong to the group

of microprogrammable processors, but they are described separately because of special functions they perform. An emulator is also an interpreter, but its primary function is to execute programs using its own, basic instruction repertoire.

TABLE 3; Characteristics of various types of instruction set processors.

Hardwired processor	Instruction set is fixed
Microprogrammed processor (read only control storage)	Instruction set is fixed, but it can be customized prior to machine delivery
Microprogrammable processor (writeable control storage)	Instruction set can be customized and changed in accordance with user processing needs any time such needs arise
Dynamically microprogrammable processor (dynamically writeable control store)	Instruction set can be changed dynamically to fit various applications
Emulator	Besides having its own instruction set, it is capable of executing one or more instruction sets differing from the basic repertoire
Interpreter	Capable of executing several instruction sets that are either basic sets of some existing machines or are designed to support some special purpose application
High-level language interpreter	Executes either a conventional or a special purpose high-level language directly without using a compiler

Microinstructions are frequently classified either as vertical or horizontal [ROSI 69, IBM 711. Vertical microinstructions are short and perform usually only one simple operation. Horizontal microinstructions

are based on a very wide word and use only a minimum degree of encoding. Individual bits of horizontal microinstructions may be directly assigned to circuit control lines. This approach enables a single microinstruction to control simultaneous operations of many independent hardware elements. A combined approach then uses words of a medium length with control information encoded into several short fields. The more 'horizontal' microinstructions are, the broader repertoire they constitute, but the more difficult it is to use them. The more powerful a microinstruction repertoire, the most costly it is to implement it. A very good discussion of these aspects can again be found in [SALI 73].

How individual instructions are used depends on the application purpose of a processor (execution of basic instruction repertoire, emulation, implementation of special functions) and on the actual load of that processor. The latter implies that utilization of microinstructions is data dependent. The task of determining the dynamic utilization of microinstructions is even more complicated than analysis of machine instructions usage.

3.5. Summary

This section reviewed various aspects of instruction set processor design. We discussed the power of an instruction repertoire versus the instruction structure (format). The more powerful the instruction set, the more expensive the resulting instruction set processor, both in terms of hardware and manpower dedicated to the design project. Whether the power of the instruction set repertoire of a particular machine is fully utilized can be discovered only through analyzing instruction utilization statistics obtained under normal working conditions.

Implementation of an instruction set processor also has a number of tradeoffs; such tradeoffs can be resolved effectively only when based on the knowledge of actual instruction processing requirements. Flexibility inherent in microprogramming, especially the possibility of dynamic changes of the entire instruction repertoire, makes this technique the more and more frequently used one. To provide an adequate basis for designing microprogrammed processor, instruction utilization statistics have to be supplemented with statistics about microword utilization.

IV. TECHNIQUES FOR MEASURING DYNAMIC INSTRUCTION UTILIZATION

Before getting down to the problem of how to obtain statistics about dynamic instruction utilization, we shall review computer performance monitoring techniques in general. Performance monitoring is a process of extracting various information about the behavior of a computer system. The monitored system, frequently called the host system or the object system has to be first instrumented to make such information accessible. Instrumentation provides an interface between the host system and the monitor. The function of a monitor is to extract data and to transform it into a form suitable for recording and/or display. This is illustrated schematically in Figure 2. Monitors fall into one of four basic categories:

Hardware monitors. A hardware monitor is an external device attached to the monitored computer via a set of electronic probes. A simple hardware monitor has a set of probes, a set of counters, and some logic that allows it to combine two or more signals from probes prior to their display and/or recording. More advanced monitors include features such as sequencers, data comparators, random access memory (RAM), and associative memory. In addition, some hardware monitors are programmable--they are built around a mini-computer that dynamically selects signals for recording and may also be allowed to communicate with the host's software. Except for such communication, hardware monitors do not interfere with normal operations of the host system. Hardware monitors are capable of recording signals at high rates. They can monitor the state of any logic element, provided there is an external pin on the host's system mainframe assigned exclusively to such an element.

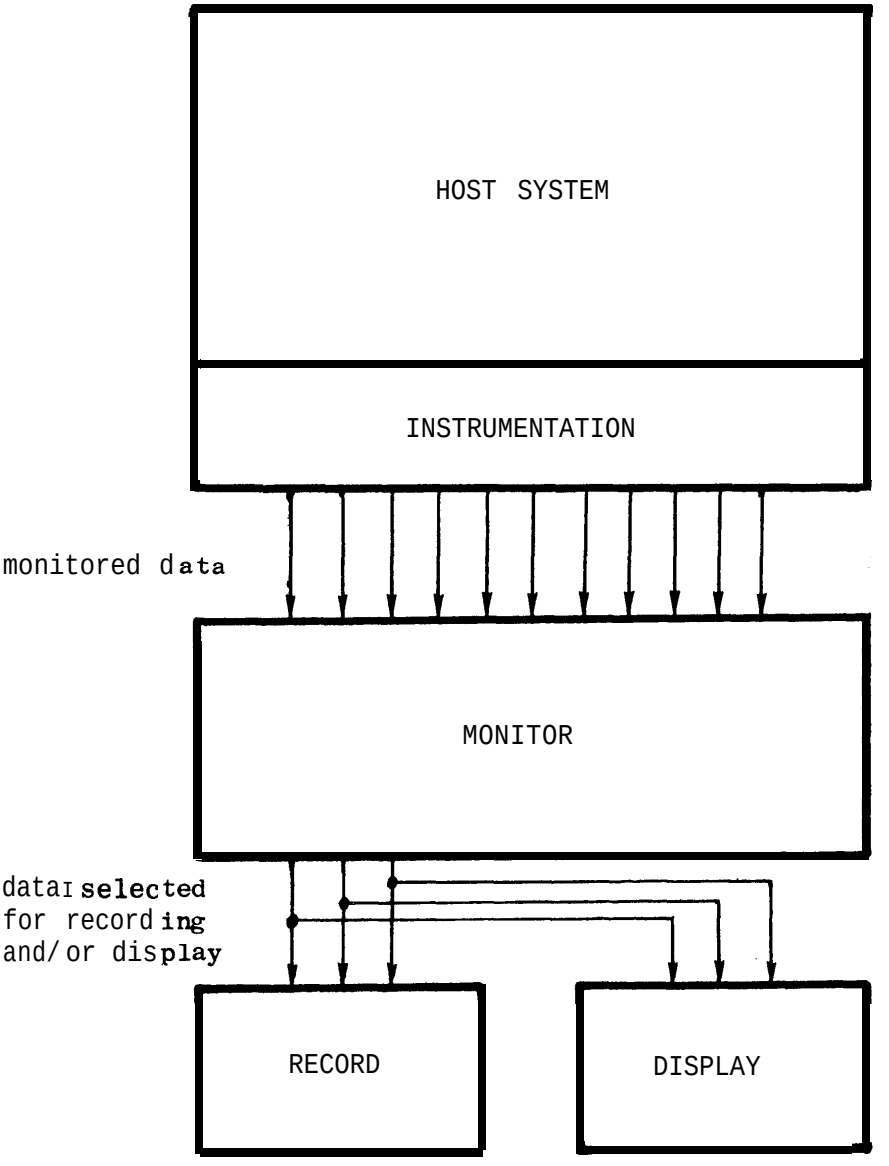


Figure 2: Simplified diagram of a system performance monitoring process.

- Software monitors. Special programs incorporated into the host's software can collect relevant data either by sampling contents of the system registers and control tables or can be invoked by the occurrence of selected events. Such programs, called software monitors, compete for system resources with jobs run on the host system. We say they generate system artifact. As the sampling or the event rate increases, the artifact may grow too large to be tolerable.

  Firmware monitors. Firmware monitors are analogous to software monitors, the only difference being that monitoring programs are written in microcode. Such a technique can greatly reduce the amount of CPU time necessary to support monitoring, but is of course applicable only to microprogrammed computers.

  Hybrid monitors. A monitor that uses both the host's facilities and an external device is called hybrid. The software part collects data stored in various memories of the monitored system and presents them in the appropriate form to a hardware monitor for further processing and recording.

Many different computer installations have reported measurements of instruction utilization. Collectively these measurements cover a broad range of monitoring techniques with representatives in each of the four classes described above. The Neurotron monitor [ASCH 71] is a hardware device developed at Argonne National Laboratory. It has a random access memory that can serve as an array of 256 counters. Each instruction is assigned one counter such that the value of the instruction opcode can be used directly as the address of the associated counter. Any time a counter is addressed, it is incremented by one. Counter update can be performed in

about 200 nsec. This device can be used on any computer provided that no instruction is executed in a time shorter than the time needed to update a counter. Bonner [BONN 69] describes what information a simple hardware monitor that does not have random access memory can extract from the operation code. A monitor of this kind, the Computer Synectics SUM, was used to determine usage of 64 operation codes on the CDC160A [ARND 72]. Several test runs had to be made since executions of only 15 different instructions could be counted during each run. A special hardware instrumentation designed for UNIVAC 1108 [BORD 711 can generate an instruction trace that is later processed to provide a report on instruction utilization. An instruction trace can also be implemented entirely in software [WIND 731. Bailey [BALY 71] discusses how instruction usage profiles can be obtained through statistical analysis of compiler-generated code. A method called self-simulation was used at UCLA [BUSS 701 to collect statistics for the S/360 Model 75 and the XDS Sigma-7. Instruction utilization statistics for the S/360 Model 91 at UCLA were produced using a software sampling method. A firmware monitor has been implemented on the SCC IC7000 at SLAC [SAAL 721. The instruction operation code of the IC7000 utilizes from 3 to 12 bits. The microcoded monitor recognizes short and long forms of operation codes and generates the address of the appropriate counter in the main storage.

The appropriateness of each technique is a function of many factors. If execution counts are the only information needed, a hardware monitor with a small RAM will be the best tool. Sometimes the execution counts alone are not sufficient, since the amount of time needed to execute a certain set of instructions depends also on the sequence in which these instructions are submitted for execution. One important factor is the distance of various instructions. Information about the distance of

successful branch instructions can decide how large the stack should be that holds prefetched instructions [BORD 71]. Distances of other instructions may be critical to a design of processors with pipelined circuits or with several independent functional units that can execute instructions in parallel. The S/360 Model 91 processor employs both concepts. Its floating-point add unit and the floating-point multiply unit may operate simultaneously, each on a different set of operands. In addition, each of these units is pipelined, meaning that a new operation can be started on a functional unit while that unit is still processing a request initiated some time earlier [ANDE 67 1. Assume that each of the requests for a particular unit can be completed before another one for the same unit is issued. In such cases the piepline concept is completely useless. Similarly, if the intervals during which the individual functional units are busy are mutually exclusive, same hardware could be shared to perform all functions distributed among such units.

The best source of information about instruction sequencing is an instruction trace. An instruction trace is a detailed record of all or selected (branch) instructions in the order they were executed. Software methods generating an instruction trace are extremely time consuming. They might slow down the computer as much as several hundred times. This is usually due to the fact that the speed of a supporting I/O device is not sufficient and the monitoring process has to be frequently interrupted (wait loop) until the I/O device completes its operation. If not the I/O speed but the CPU speed is the limiting factor and the CPU happens to be microprogrammed, it is advantageous to implement trace routines directly in microcode. CPU overhead is nevertheless significant. Since the processing speed of other system resources remains unchanged, interactions of

these resources and the CPU become time-skewed, and the trace does not give a correct picture of normal system operations. A programmable hardware monitor with fast RAM may be able to trace instructions while the monitored system is operating at full speed, but the amount of data collected is limited by the size of the monitor's memory. If such a hardware monitor has its own sufficiently fast secondary storage device, then RAM can be divided into two or more buffers, where one buffer can be copied into the secondary storage in parallel to another buffer being filled with monitored data. Both Models B and C of the CPM-X [RUUD 72] may operate in the described mode. A hardware monitor that has the ability to communicate with the monitored system can avoid losing data by creating an interrupt every time it falls behind operations of the monitored computer.

Some design areas require only a reduced amount of information about instruction sequencing. Such areas include design of conditional operation codes [FOST 71A] or development of an algorithm for microroutine swapping. The problem here is to obtain a list of the m most frequency successors for each of the n specified instructions. Such information can of course be easily extracted from an instruction trace, but there exist more efficient methods that do the same job. It is only necessary to have an nXn matrix C, such that the element c_{ij} is the count of how many times execution of the instruction i immediately preceded execution of the instruction j. Software implementation of this method would introduce almost as large an artifact as a software trace. A firmwave monitor can update c_{ij} counters in a much shorter time, but its overhead would still be significant [SAAL 721. The most efficient tool is definitely a hardware monitor where the matrix C is accumulated in the monitor random access memory under control of a set of sequencers.

The hardware device SLUR proposed in [MURP 691] can utilize its associative memory to detect specific sequences of events, and it could also be used to build the C matrix.

In summary we discussed several monitoring methods used as tools for gathering statistics about instruction usage. Mere instruction execution counts are not always sufficient; for certain design concepts instruction distance or successor relationship are more indicative parameters. Execution of each single instruction is an event that has to be processed and recorded. A software monitor that can accomplish the processing and recording task in M instructions slows down the execution process by the factor M:l. The artifact of a firmware monitor can be considerably less, but rarely falls below the level of significance. Hence, a hardware monitor is a preferred tool.

V. USING A HARDWARE MONITOR TO MEASURE INSTRUCTION UTILIZATION

This section discusses how a simple hardware monitor such as the Computer Synectics SUM described in Appendix A can serve to measure utilization of IBM System/360-370 instructions. Let x_i be the i-th bit of the instruction opcode field, where xl is the most significant bit. The eight-bit S/360-370 instruction opcode $x_1x_2x_3x_4x_5x_6x_7x_8$ can be written as two hexadecimal digits X_1X_2. Not all of the 256 possible combinations are valid opcodes. The S/360 instruction repertoire contains 150 instruction opcodes [IBM 681. The S/370 instruction set is basically the S/360 instruction set augmented with fourteen special instructions [IBM 701. The matrix in Table 4 shows the opcode value assignment for the S/370 instructions. The maximum number of distinct events the SUM monitor can record at any time is sixteen. It is therefore not possible to monitor every single instruction of the S/360-370 repertoire, or at least not in one measurement run only. Next we shall discuss two possible solutions to this problem.

(1) The two most significant bits of an S/360-370 instruction specify the instruction type (RR,RX,RS/SI or SS). The whole digit X_1 describes in most cases the type of the operand (fixed-point RR, fixed-point RX, floating-point RX, immediate, etc.). But such a division is too general. A set of instructions with the same value of Xl may be composed of several subsets of instructions, each subset performing entirely different operations. As an example, the set associated with X_1=4 consists of halfword operation instructions, branch instructions of the RX type, instructions for decimal/binary conversion, character handling instructions of the RX type, Load Address instruction, and

TABLE 4: S/360-370 instruction set--opcode value assignment.

X_1 \ X_2	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0					SPM	BALR	BCTR	BCR	SSK	ISK	SVC				*MVCL	CLCL
1	LPR	LNR	LTR	LCR	NR	CLR	OR	XR	LR	CR	AR	SR	MR	DR	ALR	SLR
2	LPDR	LNDR	LTDR	LCDR	HDR	LRDR	MXR	MXDR	LDR	CDR	ADR	SDR	MDR	DDR	AWR	SWR
3	LPER	LNER	LTER	LCER	HER	LRER	AXR		SXR	LER	CER	AER	SER	MER	DER	AUR	SUR
4	STH	LA	STC	IC	EX	BAL	BCT	BC	LH	CH	AH	SH	MH		CVD	CVB
5	ST				N	CL	O	X	L	C	A	S	M	D	AL	SL
6	STD							MXD	LD	CD	AD	SD	MD	DD	AW	SW
7	STE								LE	CE	AE	SE	ME	DE	AU	SU
8	SSM		LPSW	Diagn	WRD	RDD	BXH	BXLE	SRL	SLL	SRA	SLA	SRDL	SLDL	SRDA	SLDA
9	STM	TM	MVI	TS	NI	CLI	OI	XI	LM				*SIO SIOF	TIO	HIO HDV	TCH
A																*MC
B			*@				*STCTL	*LCTL						*CLM	*STCM	*ICM
C																
D		MVN	MVC	MVZ	NC	CLC	OC	XC					TR	TRT	ED	EDMK
E																
F	*SRP	MVO	PACK	UNPK					ZAP	CP	AP	SP	MP	DP		

*S/370 instructions only

@B202 - STIDP
B203 - STIDC
B204 - SCK
B205 - STCK

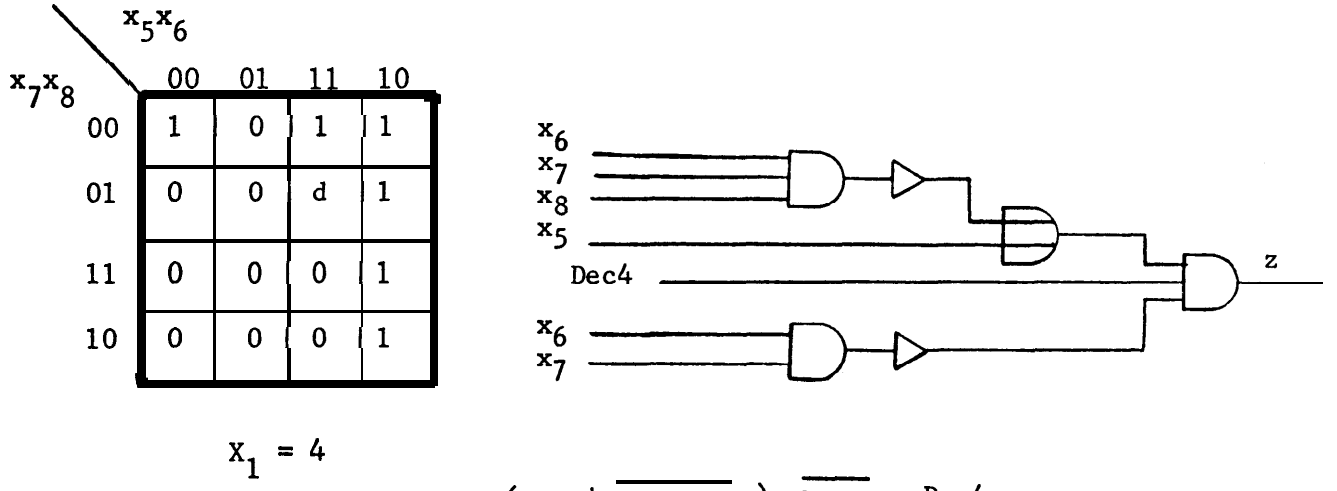
Execute instruction. Such subsets are often difficult to extract. Bits of the X_2 field have to help in identifying individual subsets.

The instruction set can be divided into m groups, $m \leq 16$, where each group g_r is assigned one SUM counter. The input CNT_r of the r-th counter is a function of X_1 and individual bits of the X_2 field, $CNT_r = h_r(X_1, x_4, x_5, x_6, x_7)$. Xl is obtained as the output of the SUM decoder; the decoder input are the four most significant opcode bits. Since the total number of gates on the SUM logic patch-panel is very limited, methods known from switching theory [MCLU 651 should be employed to minimize the number of gates needed. Figure 3 shows the circuits that decode halfword operation instructions and branch instructions.

The second opcode digit X2 determines often the type of an operation carried out by the instruction (load, store, add, etc.). But again there may be several subsets of instructions that perform different operations but have the same value of X_2. Circuits that separate such subsets can be acquired by a method analogous to the one described above where the roles of X_1 and X_2 are interchanged.

This first method provides general information about characteristics of the load processed by the measured system; the resultant data may be sufficient for the purpose of computer selection in the sense defined in Section 1, or for the purpose of determining the 'typical' instruction mix of a particular installation. For all other areas listed in Section 1 we have to measure the IUF function.

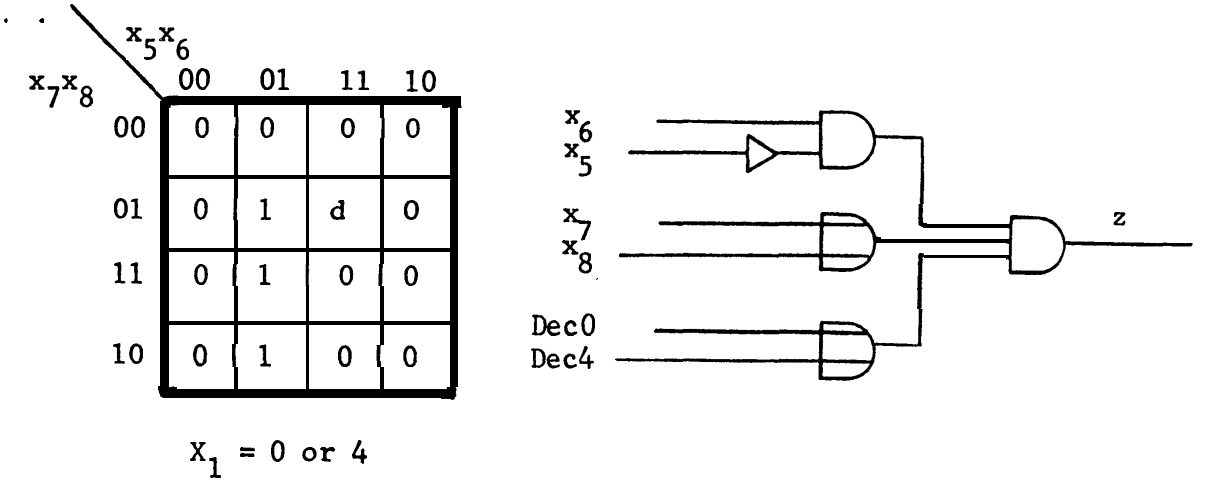
(2) The hardware monitor SUM cannot monitor more than fifteen instructions simultaneously; the sixteenth counter has to be dedicated to counting how many instructions are executed in total. Sets of fifteen instructions can be formed in many ways. The simplest way is to group



$X_1 = 4$

$$z = (x_5 + \overline{x_6 \cdot x_7 \cdot x_8}) \cdot \overline{x_6 \cdot x_7} \cdot \text{Dec4}$$

(a) Halfword operation



$X_1 = 0$ or 4

$$z = \bar{x}_5 \cdot x_6 \cdot (x_7 + x_8) \cdot (\text{Dec0} + \text{Dec4})$$

(b) Branch operation

Figure 3: Decode circuits for halfword and branch operations

instructions according to the value of either the X_1 or the x_2 digit. For X_1=1,2, or 3, all sixteen values of x_2 represent a valid opcode (Table 4); for every value of x_2 there is at least one value c_j of X_1 such that $c_j X_2$ is not a valid opcode. It means that no more than fifteen counters are needed to count usage of opcodes $x_1 x_2$ for each selected value of X_2.

Each input of the SUM sensor concentrator can be inverted prior to routing it to the logic plugboard. The sensor concentrator can therefore serve as part of the decoder for the digit X_2. Let the output of the sensor concentrator line corresponding to x_j be false if the bit x_j has the desired value. X_2 is then decoded as illustrated in Figure 4. X_1 is decoded in the SUM decoder that is enabled only when X_2 has the selected value. It is very convenient to use the sensor concentrator this way since the logic plugboard does not have to be rewired between individual runs; the digit X_2 is selected by setting the sensor switches.

It is important to make monitored intervals sufficiently long to eliminate the effect of workload changes during various times of day. Data collected by this method can be verified in two ways. If f_k is the frequency of occurrence of the k-th opcode, then $\sum_{k=1}^{s} f_k$, where s is the number of valid opcodes to equal 1. The second way to verify the results of this method is to calculate the sums $\sum_{k \in g_r} f_k$* where g_r are the instruction groups established under the first method and to compare these sums against measured utilization of each group g_r.

The two methods were implemented on the S/370 Model 145 at Stanford. Details about this particular implementation are presented in Appendix B

---

*$k \in g_r$ means the instruction assigned the integer k belongs to the group g_r.

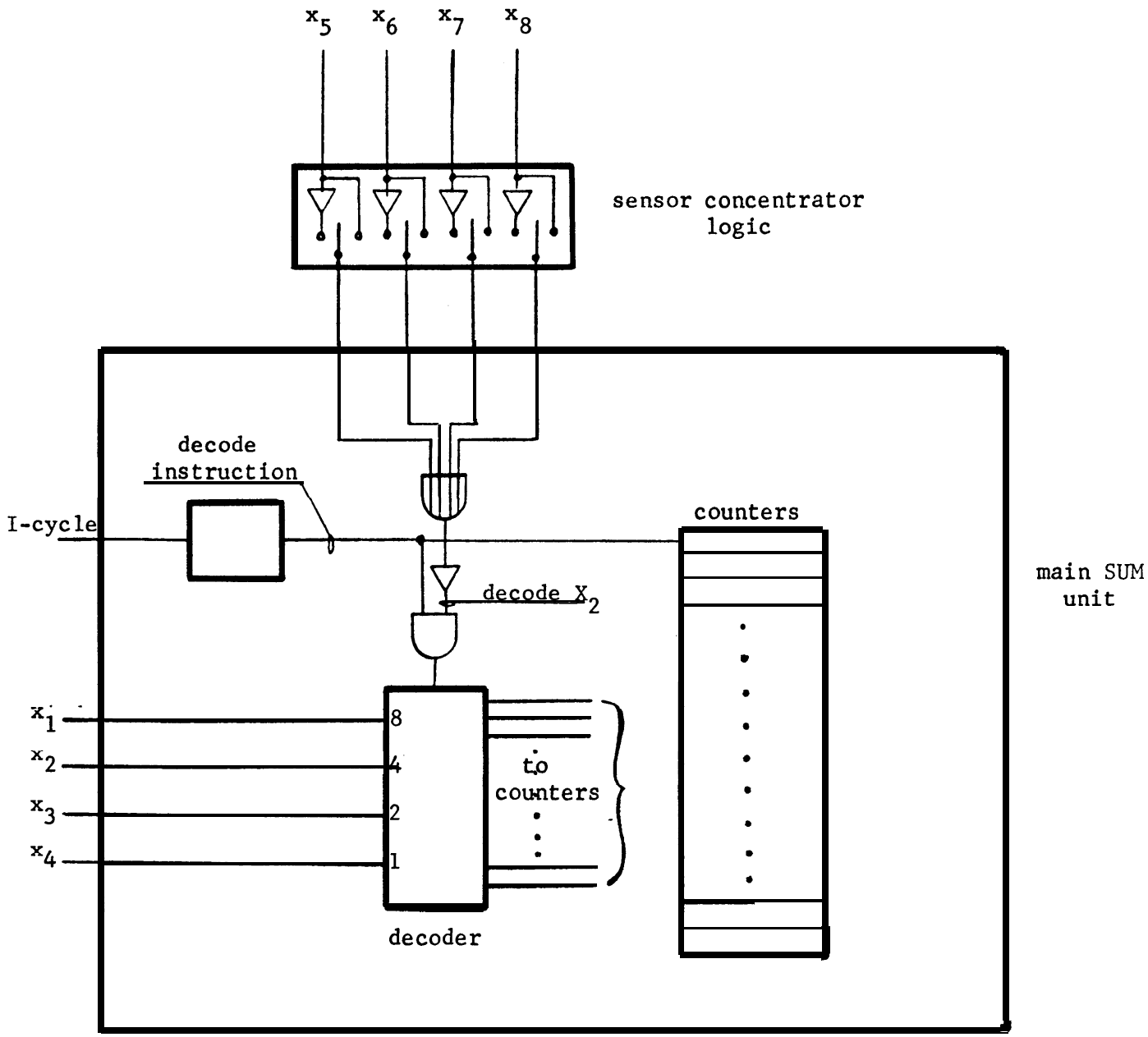


Figure 4: Hardware monitor SUM connections for monitoring the IUF

together with the measurement outcome. With slight modifications, the described methods are applicable to machines other than those using the S/360-370 instruction repertoire.

VI. ANALYSIS OF MICROCODE LEVEL FUNCTIONS

In general, execution of an instruction can be decomposed into a series of more primitive operations or steps (instruction fetch, instruction decode, operand fetch, arithmetic or logical operation, etc.); the same type of operation may occur in more than one step. In a microprogrammed computer, execution of a single step may be accomplished by one microinstruction, or may require several microinstructions, formed usually into a microroutine. Individual bits and subfields of a microword have then direct control over circuits performing actual data transfer and data transformation. These levels of the instruction execution process are illustrated in Figure 5.

The number and types of operations each instruction is comprised of is not determined merely by the instruction opcode. For some instructions this number varies only as a result of whether such instruction has been prefetched or not. For other instructions, it is a function of the operand length (S/360-370 Multiply instruction). In addition, the number of microwords required to carry out a particular type of an operation may be operand dependent. As a result, certain assumptions about the operand value distribution and the operand length distribution have to be made; only then we can estimate the numbers of microwords of various types executed when interpreting a particular instruction.

Let o_i be the primitive operation of the type i and n_{ki} the average number of such operations needed to interpret the instruction I_k. Let m_{ij} be the average number of microwords c_j of the type j executed per each operation o_i.

Let the M_k-profile of the instruction I_k be defined as p-tuple

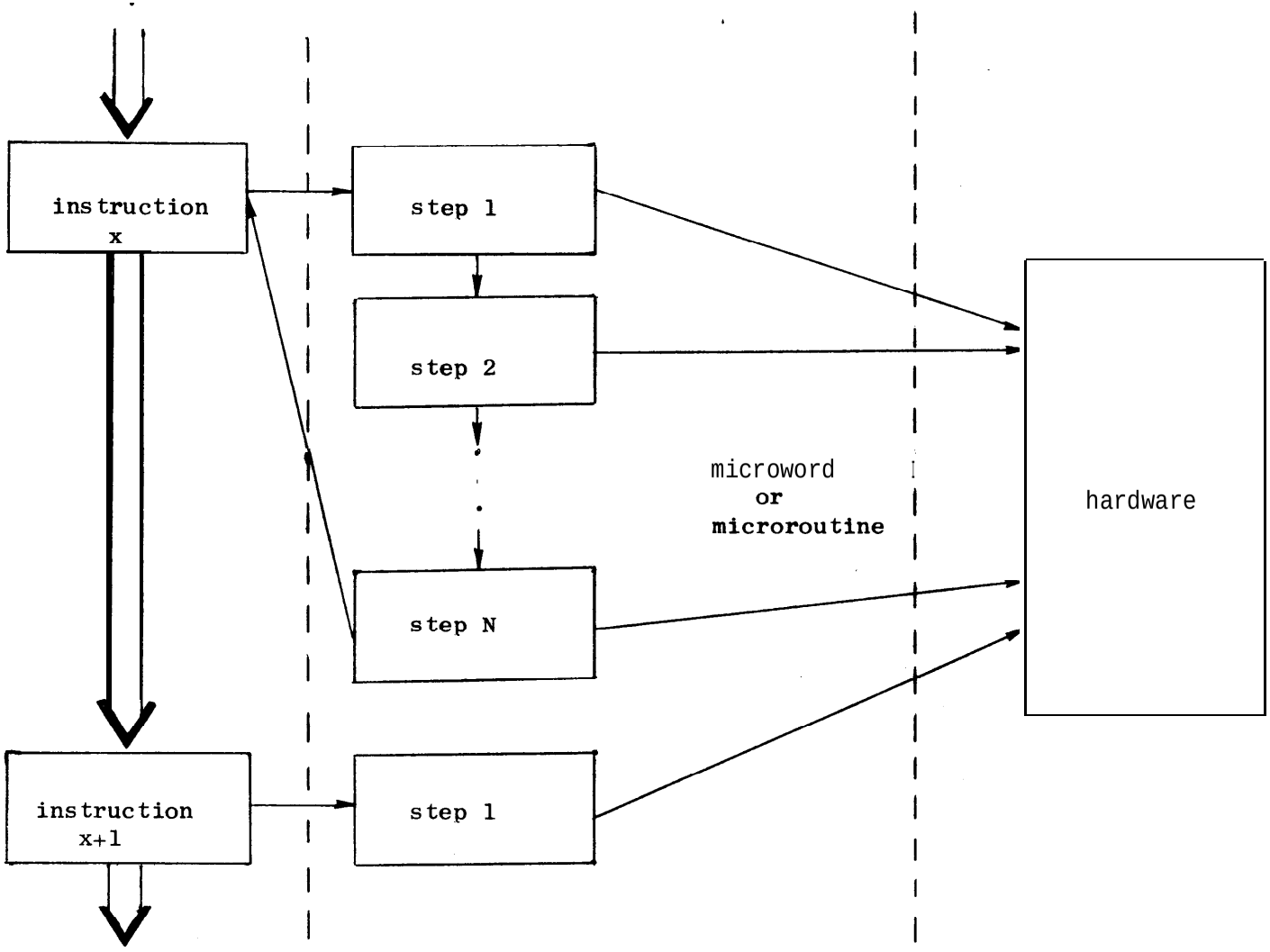


Figure 5: Levels of the instruction execution process in a microprogrammed processor.

$[M_{k1}\ M_{k2}\ \ldots\ M_{kp}]$ where M_{kj} is the average number of microwords of the type j executed per such instruction and p is the number of different microword types. Let L_{kj} be the minimum number of j-type microwords that must be executed and U_{kj} the maximum number of j-type microwords that can ever be executed in the process of executing the instruction I_k. Then the p-tuple $[L_{k1}\ L_{k2}\ L_{k3}\ \ldots\ L_{kp}]$ is the lower boundary of the M_k-profile; the p-tuple $[U_{k1}\ U_{k2}\ U_{k3}\ \ldots\ U_{kp}]$ is the upper boundary of the M_k-profile. The M_k-profile can be obtained as:

$$[n_{k1}\ n_{k2}\ n_{k3}\ \ldots\ n_{kr}]_{1 \times} \begin{vmatrix} m_{11} & m_{12} & m_{13} & \ldots & m_{1p} \\ m_{21} & m_{22} & m_{23} & \ldots & m_{2p} \\ & & & & \\ m_{r1} & m_{r2} & m_{r3} & \ldots & m_{rp} \end{vmatrix} = [M_{k1}\ M_{k2}\ M_{k3}\ \ldots\ M_{kp}]$$

Knowing the M_k-profile, the average execution time T_k of the instruction I_k can be calculated as $T_k = \sum_{j=1}^{P} M_{kj} * t_j$ where t_j is the time needed to execute a microword of the type j.

Microinstruction utilization statistics are perhaps the only means for evaluation and selection of emulators. Salisbury defines emulator power P as the number of instructions emulated per second. Emulator power can be calculated from a corresponding frequency micromix, which is the frequency of execution of each microinstruction for a fixed number and particular mix of emulated instructions [SALI 731. The application of a microinstruction mix to the task of emulator selection is very similar to using an instruction mix for the purpose of computer selection. The main drawback of the instruction mix method is that it does not evaluate effects of the system software. That part of microcode that performs the actual interpretation of the instruction repertoire, the true emulator,

can be viewed as a collection of application programs written in the processor's lowest-level language and the problem of software (operating system, language processors) evaluation thus does not arise here.

Knowing the instruction utilization function IUF and the M_k-profile of each instruction I_k, $k=1,2,3,...,s$, the relative frequency e_j of each microword type j can be calculated as $e_j = \sum_k f_k * M_{kj} \Big/ \sum_k \sum_j f_k * M_{kj}$. Earlier we showed that the exact values of M_{kj} parameters cannot be calculated; they have to be measured. Software measurement techniques are'inapplicable to the microcode level. Let us explore the possibility of using a firmware monitor to gather statistics about microinstruction utilization. Since the microinstruction set is far more primitive than the machine instruction set, the number of microinstructions n that have to be executed each time a firmware monitor is called to process and to record an event is much greater than the number of instructions executed by a software monitor to accomplish a similar task. Let t be the average time it takes to execute a single microword and T the average instruction execution time. Then t*n is the average execution time of the firmware monitor and T*N is the average execution time of the software monitor. The microinstruction execution rate is T/t times greater than the instruction execution rate. The ratio of overhead of a firmware implemented microinstruction trace to that of a software instruction trace can be calculated as $(t*n)/(T*N)*(T/t) = n/N$. We just showed that the overhead of a firmware monitor measuring microcode-level functions can be even more severe than the overhead of a software monitor performing analogous operations on the machine instruction level.

The process of execution of a microinstruction is very closely related to operations of various hardware circuits. Most of the parameters

of interest, such as the microinstruction opcode, the instruction opcode, the interrupt status, the processor status, are at some instant available as output signals of discrete hardware elements. These parameters can be therefore easily measured by a hardware monitor, providing that the probes of such a monitor are sensitive enough to recognize high-frequency changes of states of microcode-level elements. From this discussion it follows that a suitable (high-frequency) hardware monitor makes the most appropriate tool for microcode-level measurements.

6.1 S 370 Model 145 Microcode

S/370 Model 145 microinstructions incorporate both vertical and horizontal microprogramming characteristics. The type and the amount of elementary operations performed by a single S/370 Model 145 microinstruction or control word varies. The type of a control work is encoded in the four most significant bits of each word. The six basic types of control words are listed in Table 5 together with their representative binary code. Their interpretation and implementation is discussed in Appendix C. With the exception of a storage word, each control word is executed in one CPU cycle. However, the CPU cycle length is variable. Microwords that belong to the same group under the division presented in Table 5 do not always execute in the same time. For example, 0001 type microwords are usually executed in a 202.5 nsec cycle. But sometimes a control word of this type switches modules also, and in such cases it requires a 247.5 nsec cycle. A different division had to be therefore accepted. Table 6 describes control word types under this new division together with their CPU cycle length. Each type is assigned a decimal digit for the purpose of easier reference.

TABLE 5: S/370 Model 145 control word types.

CONTROL WORD TYPE	BINARY REPRESENTATION
Branch and module switch	0000
Branch	0001
Branch and link or return	0010
Word Move	0011
Storage word	0100 - 0111
Arithmetic word	1000 - 1111

TABLE 6: S/370 Model 145 control word types--division based on the control word function and the CPU cycle length.

j	CONTROL WORD TYPE c_j	CPU CYCLE LENGTH t_j (nsec)
1	Branch and module switch	247.5
2	Branch	202.5
3	Branch and link	202.5
4	Return	247.5
5	Word move	202.5
6	Storage word read	540.0
7	Storage word store	607.5
8	Arithmetic word fullword operation	247.5
9	Arithmetic word byte operation	202.5

Microcode utilization measurements were performed in two stages. The first stage provided statistics about execution of individual microword types as a result of overall system operations. During the time the CPU is busy, the state of microword utilization is a direct consequence of what instructions are executed, or in other words, microword utilization is a function of instruction utilization. The second stage of microword-level measurements is concerned with how individual instructions contribute to this overall microword utilization.

From the programmer's point of view, the instruction sets of all S/360-370 models are identical, but their implementation differs radically. S/360 Models 75, 91 and 195 are hardwired. Each of the other models has its own private set of microinstructions. While S/360-370 programs can be easily transfered from one model to another, the models are categorically incompatible at the microprogram level. Because of the differences in microinstruction sets, it is not possible to compare microword utilization statistics across different S/360-370 models as was done for machine instructions. But the techniques developed here for the S/370 Model 145 can be applied to other microprogrammed S/360-370 models; in fact, they can be applied to any microprogrammed computer.

6.2 Microword Utilization Measurements

The C-register* bits were monitored by the hardware monitor SLJM probes and decoded in the SUM decoder. The decoder operation was controlled by the signals applied to the decoder strobe input. When the strobe input is active, one and only one of the decoder outputs must be active. Selection of the strobe signal is one of the most critical tasks in microcode measurement experiments. The time interval during which the C-register content is

*For explanation see Appendix C.

valid is a function of the CPU cycle length. But the validity condition always holds between the start of a G-time delay and a l-time delay pulse.* The O-time delay signal was therefore chosen as the indicator of the C-register content validity. The actual decoder strobe is the AND function of this signal and the signal representing the global condition G, which selects the period when microwords are to be decoded. The measurements were repeated for three different interpretations of the condition G:

Case 1: G stands for CPU busy. In this case, the output is the microword frequency distribution for routines handling normal instruction processing and microprogram-controlled I/O operations (channel traps, shared cycles).

Case 2: G is true when the CPU is busy but not assisting any I/O operation. The microword frequency distribution covers instruction processing only.

Case 3: G corresponds to CPU wait state.

-Results covering all three cases are presented in Appendix D.

6.3 M-Profile Measurements

The two experiments that will be discussed next use basically the same approach for obtaining microword counts as the measurements discussed above, but an additional effort was required to generate the condition G. Details of how the condition G was generated are covered in Appendix E. The first experiment measures M-profiles of the instruction fetch phase (I-phase, I-cycles). The second experiment covers both phases of instruction execution.

Instruction execution times and the time spent in I-phase (I-cycle time) were obtained in two different ways:

a) Measured: Measure the length of time the condition G is true. The internal clock of the hardware monitor SUM is too slow for this purpose. The execution time for the most of the Model 145

*For explanation see Appendix C.

instructions is on the order of 1 or 2 μsec. The SUM clock resolution is 1 μsec. The external timer based on a 9 MHz crystal controlled oscillator was used instead to clock the signal representing the condition G.

b) Computed: $T_k = \sum_{j=1}^{9} M_{kj} * t_j$ where M_{kj} are measured frequencies of occurrence of each microword type under the condition G.

6.3.1 I-Phase Measurements

A separate measurement run was performed for each of the four classes of the S/370 instructions (RR, RX, RS/SI, SS). The condition G had to be true during and only during the I-phase of instructions belonging to the class currently monitored; G was generated as shown in Figure 6a.

The I-cycle latch is set at the beginning of the I-phase and remains set until the first microword for the Execute phase is half executed. The hardware monitor SUM has no means for recognizing the actual start of the Execute phase. Table 7 shows the lower and the upper M-profile boundaries for the I-phase of each instruction class. Only three types of microinstructions may occur during any I-phase. The results of I-phase measurements (Table 8) show non-zero values in the entries other than those in Table 7. This is a consequence of the fact that the first word of an Execute phase is measured along with microwords executed during the preceding I-phase.

6.3.2 Measurements Covering Both Phases of Instruction Execution

The most frequently used instructions, identified during the instruction utilization measurements (Section 5), were selected to be analyzed individually. The circuit for generation of the condition G is shown in Figure 6b. Each selected instruction I_k was monitored for a period of 100 seconds. The lower and the upper boundaries L_k and U_k were obtained

TABLE 7: Lower and upper M-profile boundaries (I-phase only).

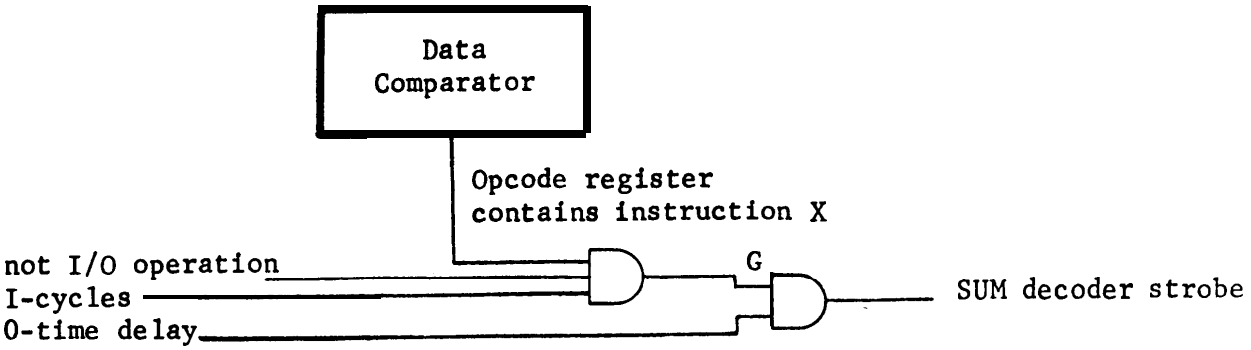
MICROWORD TYPE	BOUNDARY	INSTRUCTION TYPE			
		[illegible]	[illegible]	RS/S	SS
Word move	L	[illegible]	0	[illegible]	0
	U	4	2	2	2
Storage word read	L	0	0	0	0
	U	1	2	2	2
Arithmetic word fullword oper.	L	0	0	0	1
	U	0	1	2	2
Minimum number of CPU cycles		1	2	1	2
Maximum number of CPU cycles		6	6	5	6

by static analysis of corresponding microroutines. Figure 7 illustrates the flow of control during the execute phase of the Branch on Condition instruction (BC), the Store Multiple instruction (STM), and the group of shift instructions. Numbers inside the circles describe individual control word types as they were assigned in Table 6. Table 9 contains the measured M-profiles of the selected set of S/370 instructions. For easier comparison, the values of L_k and U_k are included in the same Table. The errors ($M_{kj} < L_{kj}$ or $M_{kj} > U_{kj}$) are due to the fact that the hardware monitor SUM is disabled during the time the counter contents are being written on magnetic tape and some data are therefore lost. The last column, labeled Instructions executed, is the number of instructions of the particular type I_k executed during the monitored interval, over which the average values

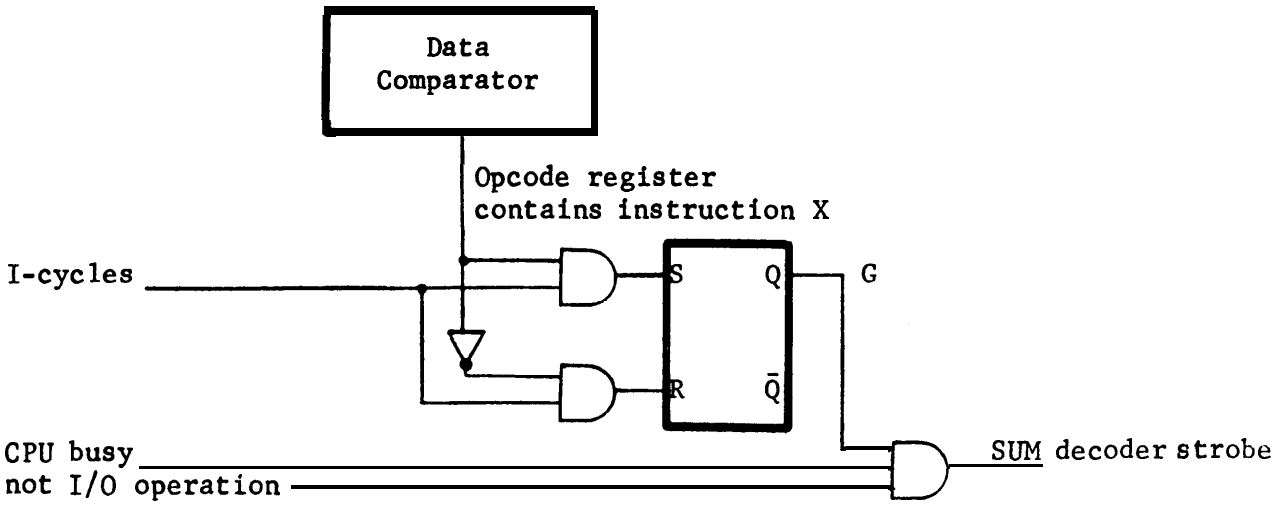
M_{kj} were calculated. In addition, instruction execution times obtained by direct measurement, by being computed, and those listed in the IBM manual [IBM 72] are compared in this table. The difference between the measured and the calculated values is on the order of the external clock resolution. The difference between the IBM values and the measured values is much greater. Since instruction execution times are a function of the processing environment, it would not be appropriate to say that the values supplied by IBM are incorrect.

TABLE 8: Measured M-profiles (I-phase only).

MICROWORD TYPE	INSTRUCTION TYPE			
	RR	Rx	RS/SI	SS
Branch and module switch	0.273	0.822	0.809	0.433
Branch	0	0.002	0.002	0
Branch and link	0	0	0	0
Return	0	0	0	0
Word move	1.601	1.293	0.850	1.098
Storage word read	0.630	0.832	0.658	1.022
Storage word store	0	0	0	0
Arithmetic word fullword operation	0.551	0.941	0.632	1.424
Arithmetic word byte operation	0.094	0.002	0.171	0.429
Average number of CPU cycle for I-phase	3.78	4.72	3.78	5.43
I-cycle time (sec) measured	0.876	1.130	0.896	1.299
I-cycle time (sec) computed	0.888	1.148	0.919	1.321



(a) I-phase monitored only



(b) Both phases of instruction execution monitored

Figure 6: Circuits that generate the condition G for M-profile measurements

TABLE 9: M-profiles of selected IBM S/370 instructions (Part 1).

Instruction		Microword type									Instruction execution time			Instruction executed
		Branch and module switch	Branch	Branch and link	Return	Word move	Storage word read	Storage word store	Arithmetic word fullword	Arithmetic word byte	computed [μsec]	measured [μsec]	IBM * timings [μsec]	
BALR	L	0	0	0	1	3	0	0	0	1	1.416	1.343	1.682 + .874	89204
		0.00	0.00	0.00	1.00	3.28	0.55	0.00	0.00	1.00				
	U	0	0	0	1	7	1	0	0	1				
BCTR	L	0	0	0	1	1	0	0	1	0	1.184	1.137	1.074 +1.078	38261
		0.08	0.00	0.00	1.00	1.58	0.64	0.00	1.00	0.00				
	U	1	0	0	1	5	1	0	1	0				
BCR	L	0	0	0	1	1	0	0	0	0	0.917	0.902	0.872 + .875	187162
		0.30	0.00	0.00	1.00	1.62	0.47	0.00	0.03	0.00				
	U	1	0	0	1	5	1	0	1	0				
SVC	L	13	6	0	1	7	3	2	4	13	13.551	13.167	13.497	12027
		13.05	5.99	0.00	1.01	8.13	4.01	2.00	4.00	14.00				
	U	14	8	0	1	11	4	2	4	17				
LPR LNR LTR LCR	L	1	0	0	1	1	0	0	1	1	1.752	1.689	1.373 1.676	26139
		1.90	0.00	0.00	1.00	1.09	0.48	0.00	1.41	1.00				
	U	3	1	0	1	5	1	0	2	1				
LR	L	0	0	0	1	2	0	0	0	0	1.100	1.061	0.923	154495
		0.00	0.00	0.00	1.00	2.36	0.77	0.00	0.00	0.01				
	U	0	0	0	1	5	1	0	0	0				
AR SR	L	2	0	0	1	1	0	0	1	1	1.868	1.761	1.373 1.575	47946
		2.00	0.00	0.00	1.00	1.10	0.84	0.00	1.00	1.00				
	U	2	0	0	1	4	1	0	1	1				
STD LD	L	1	0	0	1	1	2	0	0	0	3.962	3.991	3.386 2.633	42082
		2.57	0.00	0.00	1.00	3.91	2.44	1.00	1.50	0.00				
	U	4	0	0	1	5	3	2	3	0				
STH	L	1	0	0	1	0	0	1	0	0	1.695	1.671	1.498	232697
		1.00	0.00	0.00	1.00	0.51	0.52	1.00	0.81	0.00				
	U	1	0	0	1	2	2	1	2	0				
LA	L	1	0	0	1	2	0	0	0	1	1.717	1.712	1.452	3861807
		1.01	0.00	0.00	1.00	2.47	0.52	0.00	0.96	1.00				
	U	1	0	0	1	4	1	0	2	1				
STC	L	1	0	0	1	2	1	1	2	0	1.507	1.497	1.452	25188
		1.00	0.00	0.00	1.00	0.15	0.47	1.00	0.48	0.00				
	U	1	0	0	1	2	1	1	2	0				
IC	L	1	0	0	1	0	1	0	0	0	1.596	1.592	1.384	66101
		1.01	0.00	0.00	1.00	0.39	1.52	0.00	0.80	0.00				
	U	1	0	0	1	2	2	0	2	0				
BAL	L	1	0	0	1	3	0	0	0	1	2.194	2.165	2.399	96434
		1.00	0.00	0.00	1.00	4.03	1.05	0.00	0.43	1.00				
	U	1	0	0	1	5	2	0	2	1				
BCT	L	1	0	0	1	0	0	0	1	0	1.463	1.434	1.369 + .873	889798
		1.00	0.00	0.00	1.00	1.51	0.31	0.00	1.99	0.00				
	U	1	0	0	1	3	2	0	3	0				
BC	L	0	0	0	1	0	0	0	0	0	1.152	1.133	0.917 + .875	5890011
		0.54	0.00	0.00	1.00	1.50	0.43	0.00	0.94	0.00				
	U	1	0	0	1	3	2	0	2	0				
LM	L	1	0	0	1	3	1	0	0	0	2.730	2.678	2.295	232697
		1.78	0.00	0.00	1.00	3.77	1.86	0.00	1.11	0.00				
	U	2	0	0	1	5	2	0	2	0				
AH SH	L	3	0	0	1	2	1	0	1	1	3.001	2.924	2.749	347402
		3.20	0.00	0.00	1.00	2.12	1.55	0.00	1.01	1.00				
	U	4	0	0	1	4	2	0	3	1				

TABLE 9. M-profiles of selected IBM S/370 instructions (Part 2).

Instruction		Microword type										Instruction execution time			Instructions executed
		Branch and module switch	Branch	Branch and link	Return	Word move	Storage word read	Storage word store	Arithmetic word fullword	Arithmetic word byte	computed [μsec]	measured [μsec]	IBM * timings [μsec]	
ST	L	1	0	0	1	0	0	1	0	0	1.665	1.670	1.497	[illegible]253
	1.01	0.00	0.00	1.00	0.32	0.60	0.99	0.67	0.01				
	U	1	0	0	1	2	1	1	2	0				
NC OC XC	L	1	0	0	1	1	1	0	0	1	2.007	1.977	2.700 2.138	42781
	1.62	0.00	0.00	1.01	1.14	1.11	0.00	1.39	2.00				
	U	2	0	0	1	2	2	0	3	2				
L	L	0	0	0	1	2	1	0	0	0	1.862	1.851	1.688	64664
	0.55	0.00	0.00	1.00	2.59	1.53	0.00	0.51	0.00				
	U	1	0	0	1	4	2	0	2	0				
S	L	2	0	0	1	1	1	0	1	1	2.482	2.463	2.385 2.340	50479
	2.98	0.00	0.00	1.00	1.23	1.04	0.00	1.93	1.00				
	U	3	0	0	1	3	2	0	3	1				
M	L	2	0	0	1	1	1	0	3	0	16.596	17.112	20.077	2966
	6.21	0.11	0.00	1.00	5.36	1.25	0.00	32.23	25.07				
	U	28	1	0	1	6	2	0	50	55				
D	L	4	0	0	1	3	1	0	97	34	33.93	34.36	34.186	9860
	4.77	0.00	0.00	1.01	3.39	1.53	0.00	97.30	33.99				
	U	5	0	0	1	5	2	0	104	34				
Shift	L	5	1	0	1	0	0	0	0	3	4.763	4.760	2.900 to 13.700	1026862
	5.73	2.38	0.00	1.00	0.15	0.52	0.00	5.53	4.58					
	U	8	4	0	1	3	1	0	27	13				
STM	L	2	0	1	2	0	1	1	2	8	12.463	12.453	b.579 to 19.774	473467
	2.02	0.00	1.00	2.00	1.06	1.59	8.25	2.51	22.50					
	U	2	0	1	2	2	2	16	3	38				
TM	L	1	0	0	1	0	1	0	0	3	2.437	2.408	1.992	1661875
	1.39	0.00	0.00	1.00	0.36	1.75	0.00	0.87	3.01					
	U	2	0	0	1	2	2	0	1	3				
MVI	L	1	0	0	1	0	0	1	0	0	1,784	1.761	1.452	665099
	1.00	0.00	0.00	1.00	0.343	0.70	1.00	0.93	0.00					
	U	1	0	0	1	2	1	1	1	0				
NI	L	1	0	0	1	0	1	0	0	3	2.718	2.711	2.397	180126
	1.01	0.00	0.00	1.00	0.63	1.80	1.00	0.44	1.99					
	U	1	0	0	1	2	2	1	1	2				
CLI	L	2	0	0	1	0	1	0	0	2	2.263	2.219	1.992	1310510
	2.01	0.00	0.00	1.00	0.66	1.57	0.00	0.51	2.00					
	U	2	0	0	1	2	2	0	1	2				
OI XI	L	1	0	0	1	0	1	1	0	2	2.685	2.651	2.397	48967
	1.01	0.00	0.00	1.00	0.89	1.61	1.00	0.65	2.01					
	U	1	0	0	1	2	2	1	1	2				
LM	L	2	0	1	2	0	1	0	2	9	13.775	13.884	4.566 to 18.741	405998
	2.00	0.00	0.99	2.00	0.28	11.64	0.00	2.65	27.55					
	U	2	0	1	2	2	18	0	3	39				
MVC	L	1	0	0	1	1	3	1	5	4	8.622	8.598	6.406 to 92.492	1560671
	1.20	0.16	0.00	1.00	1.80	4.93	2.15	5.61	11.45					
	U	#	#	0	1	#	#	#	6	#				
CLC	L	2	0	0	1	0	2	0	1	2	4.616	4.584	3.899 to 47.492	1304483
	2.01	0.28	0.00	1.00	1.51	3.97	0.00	1.55	b.76					
	U	#	#	0	1	2	#	0	2	#				

* - Instruction timings from IBM S/370 Model 145 Functional Characteristics [IBM 72]

\# - values are functions of the operand length

L - lower boundaries

U - upper boundaries

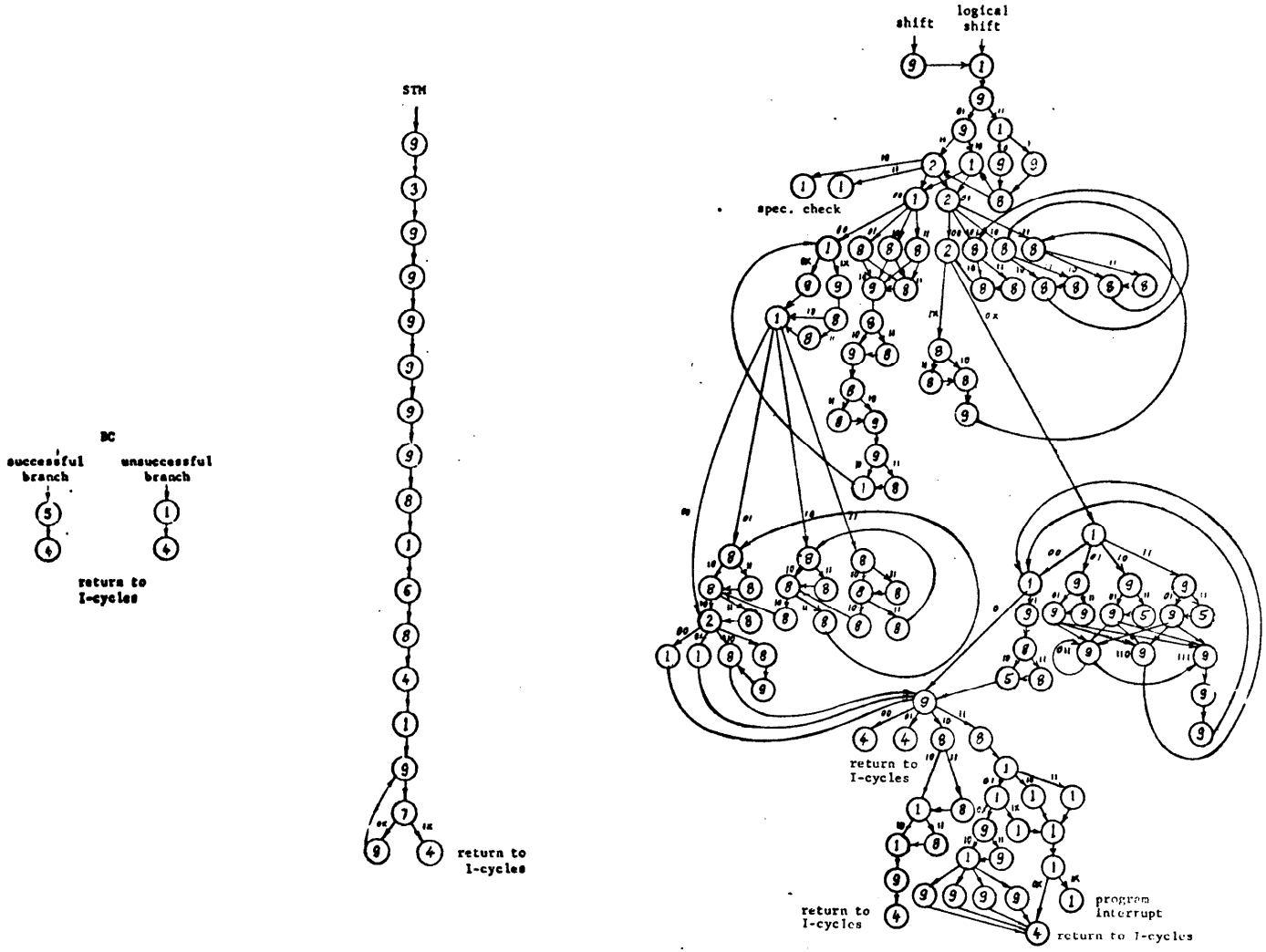


Figure 7: Flowcharts for E-phase microroutines (BC, STM and shift instructions).

M-profiles can be used to study certain aspects of the processing environment. Two examples are given below:

(1) During execution of a BC instruction, the Word "move microword" is executed only when the branch is not successful. Thus 46% of BC instructions in the measured instruction mix do branch.

(2) The average number of registers stored by a STM instruction can be obtained as the number of Storage word "store microwords" executed for this instruction. In the measured instruction mix, a STM instruction 'stores 8.25 registers on the average.

6.4. APPLICATION OF MEASUREMENT RESULTS

Data acquired with the described measurement techniques have many potential uses; they were already discussed in Sections 1 and 3. Two particular applications of the outcome of monitoring the S/370 Model 145 will be discussed next.

6.4.1 Enhancement of the Instruction Repertoire

High incidence of the LA (Load Address) instruction was closely examined. Static analysis of the most frequently used programs showed that this instruction is used predominantly to increment a counter (tally, pointer). Most frequently, such a counter is maintained in the main memory. The LA instruction is thus used accompanied by load and store operations, and the execution time of such a sequence is:

L	REG,COUNTER	1.851*
LA	REG,N(,REG)	1.712*
ST	REG,COUNTER	1.670*
		5.233 microseconds

Since general purpose registers are a scarce resource, a register used for incrementing a counter quite often has to be saved and restored:

* Instruction execution times measured using the hardware monitor (from Table 9).

```
ST      REG,SAVE            1.670*
L       REG,COUNTER
LA      REG,N(,REG)         5.233
ST      REG,COUNTER
L       REG,SAVE            1.851*
                            8.754 microseconds
```

An even worse situation arises when the code has to be reentrant and any variable that has to be saved must be put on a stack:

```
BCT     POINTER,OVERFLOW        1.434*
ST      REG,STACK(POINTER)      1.670*
L       REG,COUNTER
LA      REG,N(,REG)             5.233
ST      REG,COUNTER
L       REG,STACK(POINTER)      1.851*
LA      POINTER,1(,POINTER)     1.712*
                                11.900 microseconds
```

Using the S/370 Model 145 microcode, we can define a new SI-type instruction, INC d1(b1),I, that increments the contents of the memory word specified by bl and dl by the value of the immediate operand I. The INC instruction is assigned one of the invalid opcodes of the S/370 instruction set. The invalid opcode indicator in the Model 145 microcode has to be replaced by a branch into the E-phase microroutine for the INC instruction. Microcode implementation of the INC instruction can have the following form:

---

*Instruction execution times measured using the hardware monitor (from Table 9).

SI I-phase	0.896*	
Hardware-forced branch on opcode		
Storage word read	0.5400	Read main memory word
Arithmetic word fullword operation	0.2475	Increment by I
Storage word store	0.6075	Store back
Arithmetic word byte operation	0.2025	Set condition code
Return	0.2475	Return to I-cycles
	2.7410	microseconds

Thus any of the three machine instruction sequences presented above could be accomplished in only 2.7410 microseconds.

Decrementing a counter is an operation analogical to counter incrementation. When the decrement is 1, such operation is usually accomplished by using the BCTR instruction (BCTR REG,0). The same discussion about register saving and restoring applies to this case. Even though decrementing a counter is not as frequent an operation as incrementing a counter, a new instruction, DEC dl(bl),I, would be a useful addition to the S/370 instruction repertoire.

6.4.2 Application of Data on Microcode Level Operations

Many companies that operate a computer installation have large investments in application software. When the present system becomes saturated, selection of a new central processor may be severely restricted by the requirement that the application software would not have to be modified. This particular problem is referred to as software portability. In many instances, only models of the same family qualify as candidates for

*From Table 8.

replacement of the old hardware.

Emulation is one of the possible solutions to the problem of software portability. The instruction repertoire of the selected machine (preferably more powerful and efficient than the old one) can be used to write new software, that will run in the machine basic mode, while the old programs can be processed without changes in the emulation mode. But even in this mode, the new machine should give better performance than the old one; i.e., we expect the running time of the original programs to be less on the new machine. Salisbury [SALI 73B] developed a method for emulator evaluation. This method is based on synthetic kernels. The emulator power is estimated using the micromix corresponding to the expected system workload and execution times of emulator microinstructions. The results of the microcode level measurements described in this study were used to verify the validity of such an approach.

VII. SUMMARY AND CONCLUSIONS

Knowledge of how various instructions are used on existing machines provides a valuable basis for design of future machines. This study has focused mainly on methods for measuring instruction (and microinstruction) utilization.

We discussed the problems and trends of instruction processor and emulator design. Many shortcomings in an instruction processor are due to a lack of reliable information about processing requirements. Some parts of an instruction processor may be overdesigned, some underdesigned; both cases lead to an increase in the cost/efficiency ratio of the entire system. Processing requirements, or workload of an instruction processor, can be expressed in terms of execution frequencies of individual instructions from the available instruction repertoire. Incidence of individual instructions has to be measured. Since software monitor overhead is very high at this level, a hardware monitor is a preferable tool.

Techniques employing a simple hardware monitor were developed to measure instruction utilization and microword utilization on the S/370 Model 145. Results acquired by different techniques were compared and only small differences were detected. Microcode level monitoring is an extremely sensitive matter. The good fit of measured values M_{kj} and boundary values L_{kj}, U_{kj}, obtained by static analysis of S/370 Model 145 microroutines, is encouraging. This shows that despite the problems with traceability and timing of certain signals in the monitored processor, a hardware monitor can measure instruction set processor functions and microcode functions with reasonable accuracy.

Execution frequencies of individual instructions indicate which of

the available operations are critical and should be optimized, and which are used only rarely and could therefore be implemented using different, less costly media. But the information contained in the instruction utilization function is not sufficient to determine what other operations should be included in the instruction repertoire. Frequently occurring sequences of instructions could be merged into one instruction. A more sophisticated monitor than the hardware monitor SLIM would be needed to detect such sequences, especially if their exact composition is not known beforehand. Still there might exist other operations that would be used frequently if implemented as machine instructions, but measurements of existing programs cannot identify them. If they can be identified, it would have to be done only through analysis of processing requirements before such requirements are translated into programs. Defining a set of primitives providing the most efficient support of a particular processing environment is an area that deserves further research.

Hardware measurements of system performance on the instruction processor level and the levels below are often complicated by the fact that some events are not externally accessible. Even if all necessary events (signals) can be sensed by monitor probes, processing such events may turn out to be very difficult because of timing constraints and inconsistencies resulting from special cases. Great demand for computer performance data will hopefully lead into a situation when measurability will become one of the important aspects of hardware design. This is a closely related area of computer performance measurements that unfortunately has not yet received enough attention.

APPENDIX A. SYSTEM UTILIZATION MONITOR

The System Utilization Monitor (SUM) is a hardware monitor manufactured by Computer Synectics, Inc. [SUM 70]. Performance monitoring is accomplished by connecting the SUM sensors (probes) to externally accessible points of the monitored device. Sensors are connected via a cable to a Sensor Concentrator. The SUM has two Sensor Concentrators, and each Sensor Concentrator can accommodate up to '10 sensors. Each Sensor Concentrator input may be inverted prior to routing it to hubs on the program patch panel in the SUM. The patch panel can be wired to form various Boolean functions on monitored signals and to assign these signals and their functions to the SUM counters. Each counter may operate either in the timing mode, measuring the duration of a certain event, or in the counting mode, when it counts the number of occurrences of an event. The contents of any counter can be displayed on the SUM control panel. In addition, all counter contents can be periodically written on magnetic tape. The magnetic tape written under control of the SUM can later be processed by the SUM data reduction program SUMDAP.

Technical Specifications

Counters: 16 electronic counters
two modes of operation:
- -timing mode - minimum resolution time 1 μsec
- -counting mode - maximum repetition rate 10 MHz

mode selection accomplished via patch panel wiring
counter output:
- -visual display - single counter only selected by the pushbutton
- -magnetic tape - all 16 counters plus header information recorded in each magnetic tape record

Patch Panel: 300 hubs arranged in a 20X15 matrix
removable
signal levels: active: from 0 to +0.2 V dc
inactive: from +3 to +5 V dc
functions: 20 sensor outputs
6 2-way AND functions
4 3-way AND functions
8 4-way OR functions
4 set-reset latches
2 binary flip-flops
10 inverters
3 divide-by-l0 functions
6 fanouts
32 counter inputs (16 counting, 16 timing)
1 4x16 way decoder
function execution time: inverters - 7 nsec
decoder - 28 nsec
all other functions - 14 nsec

Sensor: responds to signals of 50 nsec or longer
maximum repetition rate 10 MHz

Basic SUM components and their connections are illustrated in Figure A.l.

DATA COMPARATOR

An optional unit, the Data Comparator, can be attached to the SUM to provide comparison facility. Up to six 4-bit sensors can be attached to the Data Comparator via a Sensor Concentrator. Outputs of the Data Comparator probes may be inverted in the Sensor Concentrator. The Data Comparator strobe (Load Comparator signal) gates the output signals of the comparator probes into the register X. The contents of this register are then compared to the contents of the Data Comparator registers A and B. Values of the A and B registers are set manually via two sets of push-buttons. The Data Comparator generates a set of signals describing the results of a comparison. The possible outputs are: $X < A$, $X=A$, $X>A$, $X < B$, $X=B$, $X > B$, $A \geq X > B$; a latch is set on the occurrence $A \geq X > B$

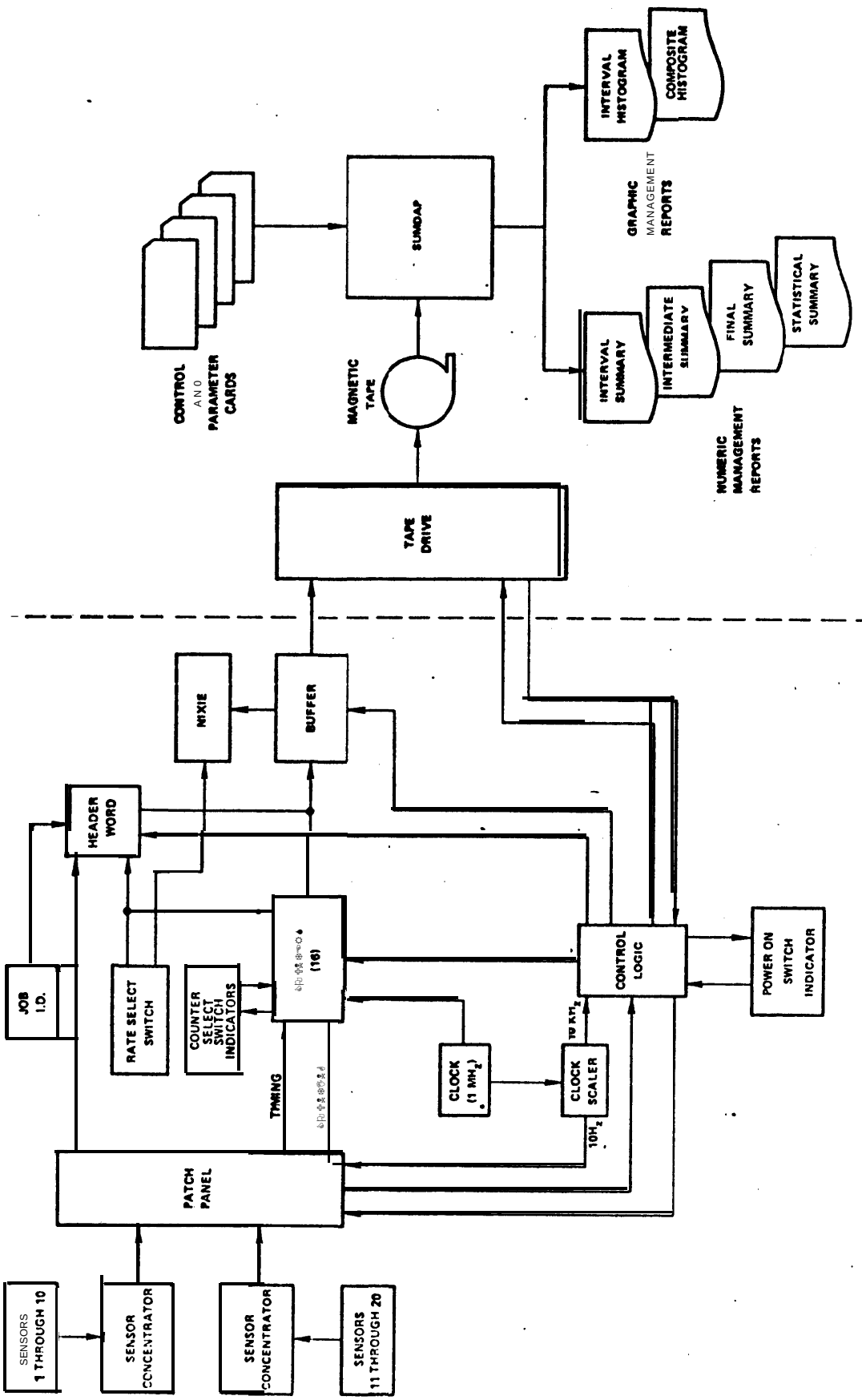


Figure A.1: SUM functional block diagram
(Courtesy of Tesdata Systems Corporation)

and reset when the condition is no longer true. Except for the latch output, comparison signals are in the form of 50-nsec pulses.

Technical Specifications

Control Panel:	2 dial switches to select the Load Comparator signal for a comparison against the A or B registers
	24 pushbutton switches for loading the A register
	24 pushbutton switches for loading the B register
	24 indicator lights displaying the contents of the A register
	24 indicator lights displaying the contents of the B register
	10 thumbwheel switches for routing the comparison results to the SUM
Sensors:	up to six 4-bit sensors for loading the X register
	up to two l-bit sensors providing the Load Comparator signal
	up to ten l-bit sensors connected to a different concentrator – can be used as Load Comparator signals or routed to the SUM for normal SUM inputs

Up to 20 Data Comparator units can be connected in tandem to the one set of sensors. Figure A.2 shows SUM connections using Data Comparators.

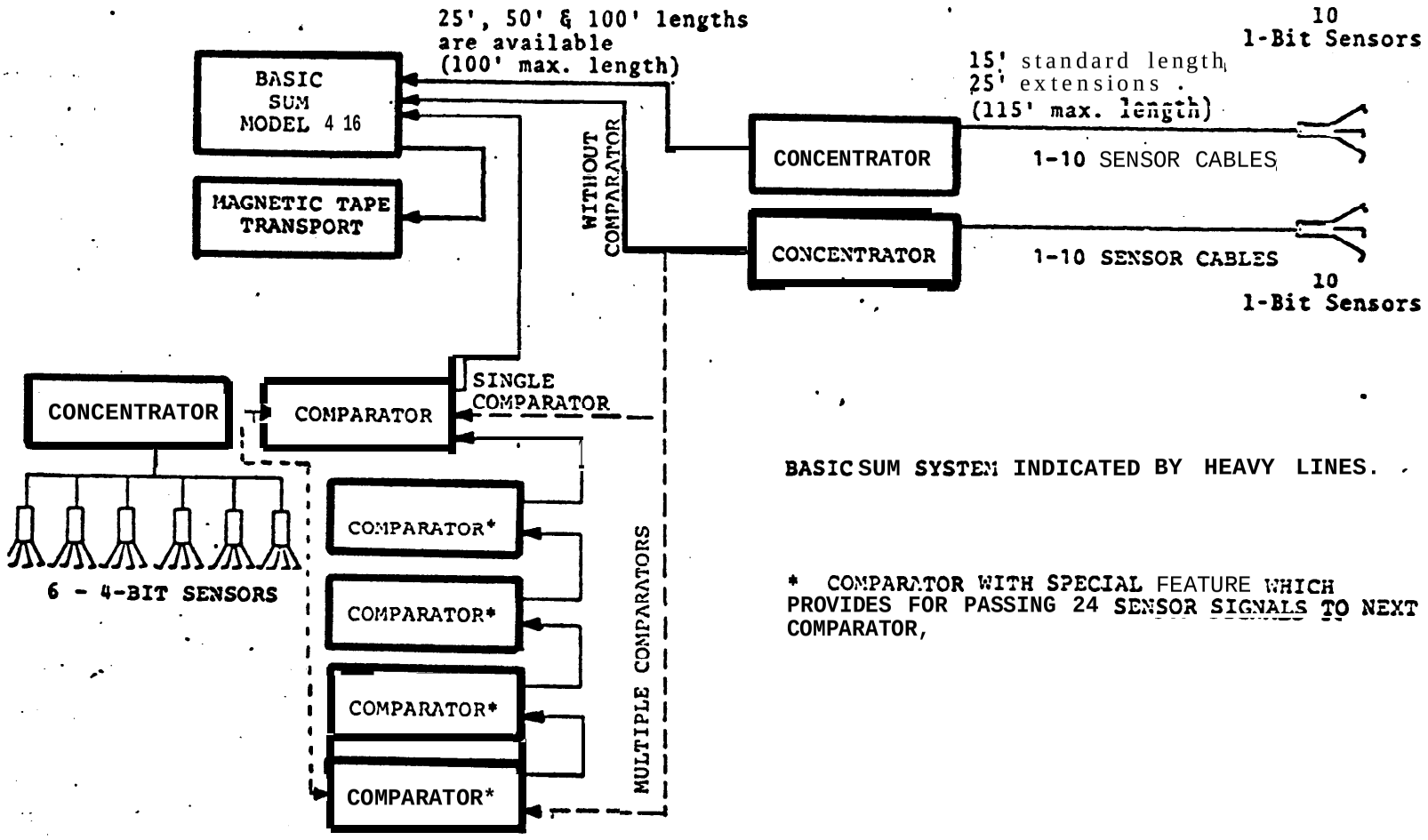


Figure A.2 : SUM connections using Data Comparators

(Courtesy of Tesdata Systems Corporation)

APPENDIX B. INSTRUCTION UTILIZATION MEASUREMENTS ON THE S/370 MODEL 145 AT STANFORD UNIVERSITY

Instruction utilization measurement methods discussed in Section 5 were implemented on the S/370 Model 145 at Stanford. This appendix presents results of these measurements and discusses some difficulties encountered during the instrumentation phase. Finally, data obtained through monitoring the Model 145 are compared to resuits of similar experiments performed at some other computer installations.

(1) The S/370 instructions were grouped in two different ways. The first division was based mainly on the operand type (X_1 digit); the second, on the function type (X2 digit). Each measurement experiment was run for a week in twelve-hour intervals (when writing one record every two seconds, the SUM tape can store data representing twelve hours of measurements). Divisions into groups g_r and frequencies of instructions in individual groups, $f(g_r)$, are presented in Tables B.l and B.2.

(2) The second method described in Section 5 was used to generate data necessary to determine the IUF and the IFD functions. Results are presented in Table B.3. The last column of this table, labeled k', specifies the permutation on the set of integers k assigned to individual opcodes. Under this permutation, $f_{k'} \geq f_{k'+1}$ for all $k'=1,2,3,\ldots$, and it therefore represents the IFD, illustrated graphically in Figure B.l. Instructions with frequencies less than .0001 were ignored when constructing the IFD; no numbers are assigned to them in the column k'. Frequencies of instructions I_k belonging to individual groups g_r from the method I were summed up and compared against frequencies $f(g_r)$ measured by the method I. These sums appear in the last columns of Tables B.1 and B.2.

TABLE B.l: Instruction group frequencies for the division based on the value of X_1.

r	Instruction group g_r	$f(g_r)$ measured	$\sum_{k \in g_r} f_k$
1	RR instructions (X_1=0)	.1134	.1081
2	Fixed-point RR instructions (X_1=1)	.1095	.0923
3	Floating-point RR instructions (Xl=2 or X_1=3)	.0016	.0018
4	RX instructions (X_1=4)	.3771	.3811
5	Halfword instructions (X_1=4)	.0776	.0768
6	Fixed-point RX instructions (X_1=5)	.1328	. 1366
7	Floating-point RX instructions (X_1=6 or X_1=7)	.0036	.0032
8	Branch instructions (X_1=0 or X_1=4)	.3274	.3202
9	RS/SI instructions (X_1=8)	.0189	.0232
10	Shift instructions (X_1=8)	.0152	.0163
11	RS/SI instructions (X_1=9)	. 1639	.1710
12	I/O instructions (X_1=9)	.0033	.0026
13	SVC instruction (X_1=0)	.0045	.0040
14	SS instructions - character operations (X_1=D)	.0825	.0824
15	Decimal SS instructions (X_1=F)	.0006	.0008

TABLE B.2: Instruction group frequencies for the division based on the value of X_2.

r	Instruction group g_r	$f(g_r)$ measured	$\sum_{k \in g_r} f_k$
1	Load instructions (X_2=8)	. 1348	.1386
2	Store instructions (X_2=0)	.0614	.0668
3	X_2=1	. 1158	. 1132
4	Move instructions (X_2=2)	.0561	.0595
5	Logical operations (AND,OR,XOR) (X_2=4 or X_2=6 or X_2=7)	.0147	.0165
6	Compare instructions (X_2=5 or X_2=9)	. 1100	.1127
7	Branch instructions (X2=5 or X_2=6 or X_2=7)	.3050	.3202
8	Add or subtract instructions (X_2=A or X_2=B)	.0674	.0626
9	Multiply instructions (X_2=C)	.0026	.0025
10	Divide instructions (X_2=D)	.0041	.0038
11	Load register test or change (X_2=0 or X_2=1 or X_2=2 or X_2=3)	.0191	.0206

Table B.3: Instruction Utilization Function (IUF) for the S/370 Model 145 at Stanford (Part I)

k	opcode c_k (hex)	instruction	f_k	k'	k	opcode c_k (hex)	instruction	f_k	k'
1	04	SPM	.0000		42	30	LPER	.0000	
2	05	BALR	.0278	10	43	31	LNER	.0000	
3	06	BCTR	.0090	26	44	32	LTER	.0000	
4	07	BCR	.0657	2	45	33	LCER	.0000	
5	08	SSK	.0004	68	46	34	HER	.0000	
6	09	ISK	.0012	55	47	35	LRER	.0000	
7	0A	SVC	.0040	40	48	36	AXR	.0000	
8	0E	MVCL	.0000		49	37	SXR	.0000	
9	0F	CLCL	.0000		50	38	LER	.0000	
10	10	LPR	.0079	27	51	39	CER	.0000	
11	11	LNR	.0021	44	52	3A	AER	.0000	
12	12	LTR	.0098	25	53	3B	SER	.0000	
13	13	LCR	.0005	64	54	3C	MER	.0000	
14	14	NR	.0000		55	3D	DER	.0000	
15	15	CLR	.0015	53	56	3E	AUR	.0000	
16	16	OR	.0000		57	3F	SUR	. ▫▫▫▫	
17	17	XR	.0020	45	58	40	STH	.0214	15
18	18	LR	.0234	13	59	41	LA	.0498	5
19	19	CR	.0107	24	60	42	STC	.0072	31
20	1A	AR	.0156	21	61	43	IC	.0206	16
21	1B	SR	.0177	18	62	44	EX	.0076	28
22	1C	MR	.0000		63	45	BAL	.0240	12
23	1D	DR	.0000		64	46	BCT	.0187	17
24	1E	ALR	.0000		65	47	BC	.1750	1
25	1F	SLR	.0011	56	66	48	LH	.0257	11
26	20	LPDR	.0001	79	67	49	CH	.0052	38
27	21	LNDR	.0000		68	4A	AH	.0165	20
28	22	LTDR	.0002	73	69	4B	SH	.0075	29
29	23	LCDR	.0000		70	4C	MH	.0005	65
30	24	HDR	.0000		71	4E	CVD	.0013	54
31	25	LRDR	.0000		72	4F	CVB	.0001	81
32	26	MXR	.0000		73	50	ST	.0306	9
33	27	MXDR	.0002	74	74	54	N	.0058	35
34	28	LDR	.0010	57	75	55	CL	.0063	32
35	29	CDR	.0002	75	76	56	O	.0001	82
36	2A	ADR	. ▫▫▫▫		77	57	X	.0002	76
37	2B	SDR	.0001	80	78	58	L	.0648	3
38	2C	MDR	.0000		79	59	C	.0122	23
39	2D	DDR	.0000		80	5A	A	.0020	46
40	2E	AWR	.0000		81	5B	S	.0033	42
41	2F	SWR	.0000		82	5C	M	.0018	49

Table B.3: Instruction Utilization Function (IUF) for the S/370 Model 145 at Stanford (Part II)

k	opcode c_k (hex)	instruction	f_k	k'	k	opcode c_k (hex)	instruction	f_k	k'
83	5D	D	.0038	41	124	94	NI	.0062	33
84	5E	AL	.0057	37	125	95	CLI	.0424	6
85	5F	SL	.0000		126	96	OI	.0060	34
86	60	STD	.0016	51	127	97	XI	.0002	78
87	67	MXD	.0000		128	98	LM	.0220	14
88	68	LD	.0016	52	129	9C	SIO	.0020	47
89	69	CD	.0000		130	9D	TIO	.0000	
90	6A	AD	.0000		131	9E	HIO	.0000	
91	6B	SD	.0000		132	9F	TCH	.0006	62
92	6C	MD	.0000		133	AF	MC	.0000	
93	6D	DD	.0000		134	B2	STIDP		
94	6E	AW	.0000		135		STIDC		
95	6F	SW	.0000		136		SCK	.0000	
96	70	STE	.0000		137		STCK		
97	78	LE	.0000		138	B6	STCTL	.0000	
98	79	CE	.0000		139	B7	LCTL	.0000	
99	7A	AE	.0000		140	BD	CLM	.0000	
100	7B	SE	.0000		141	BE	STCM	.0000	
101	7C	ME	.0000		142	BF	ICM	.0000	
102	7D	DE	.0000		143	D1	MVN	.0003	70
103	7E	AU	.0000		144	D2	MVC	.0418	7
104	7F	SU	.0000		145	D3	MVZ	.0020	48
105	80	SSM	.0005	66	146	D4	NC	.0000	
106	82	LPSW	.0057	36	147	D5	CLC	.0342	8
107	83	Diagnose	.0000		148	D6	OC	.0006	63
108	84	WRD	.0000		149	D7	XC	.0008	60
109	85	RDD	.0000		150	DC	TR	.0010	59
110	86	BXH	.0002	77	151	DD	TRT	.0017	50
111	87	BXLE	.0005	67	152	DE	ED	.0000	
112	88	SRL	.0048	39	153	DF	EDMK	.0000	
113	89	SLL	.0074	30	154	F0	SRP	.0000	
114	8A	SRA	.0010	58	155	F1	MVO	.0000	
115	8B	SLA	.0022	43	156	F2	PACK	.0003	71
116	8C	SRDL	.0006	61	157	F3	UNPK	.0001	83
117	8D	SLDL	.0003	69	158	F8	ZAP	.0003	72
118	8E	SRDA	.0000		159	F9	CP	.0001	84
119	8F	SLDA	.0000		160	FA	AP	.0000	
120	90	STM	.0132	22	161	FB	SP	.0000	
121	91	TM	.0610	4	162	FC	MP	.0000	
122	92	MVI	.0174	19	163	FD	DP	.0000	
123	93	TS	.0000						

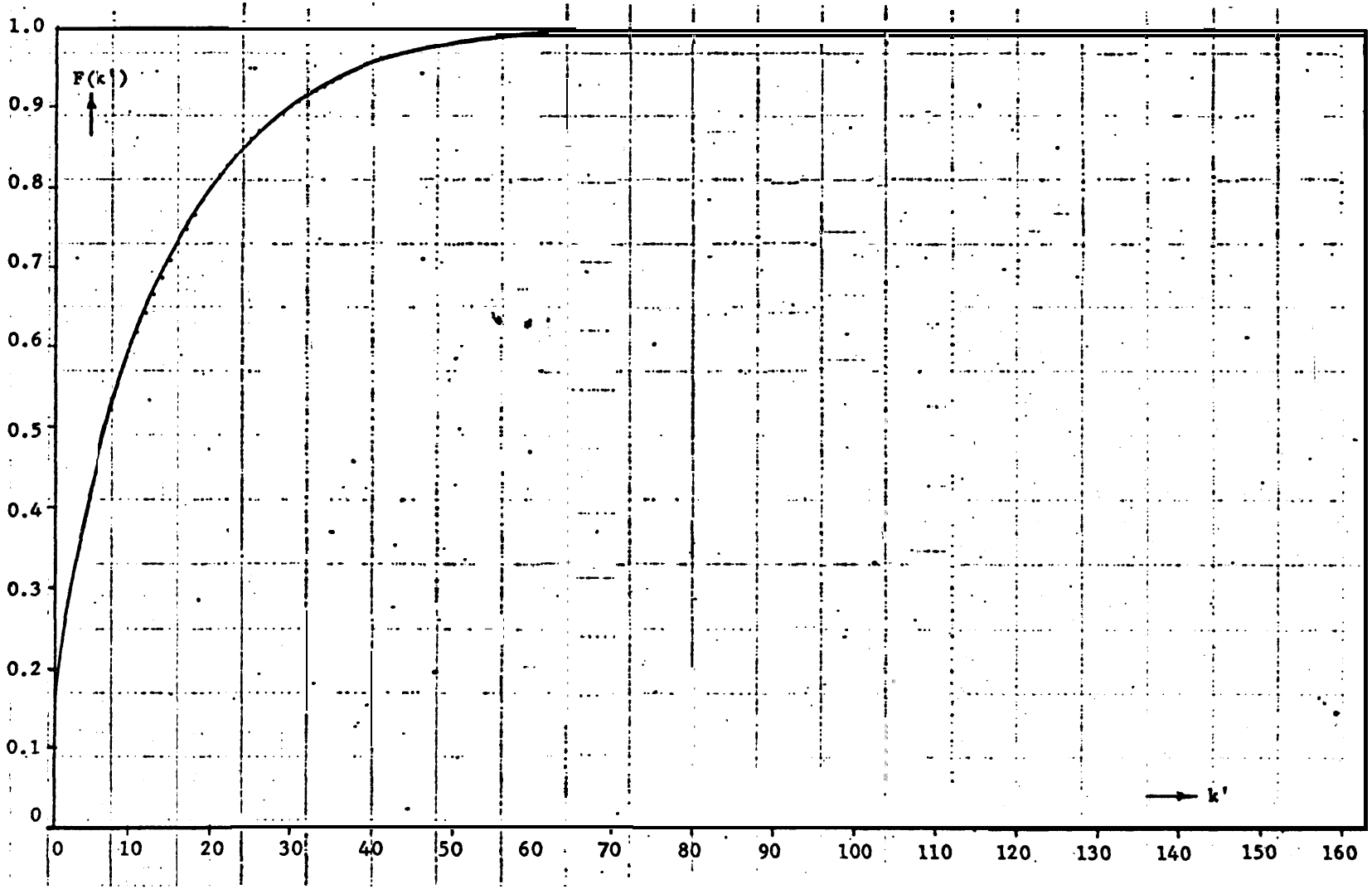


Figure B.1: Instruction Frequency Distribution (IFD)
for the S/370 Model 145 at Stanford

Difficulties Encountered During Hardware Instrumentation Implementation

The low number of various logical elements on the SUM logic plug-board was found to be quite limiting; several compromises had to be made in dividing the instruction set such that implementation of Method I would be feasible. But the main problem lay elsewhere. The opcode bits cannot be monitored at the output of the Opcode register since the Model 145 has no external attachment points provided for this register. Opcode bit values are routed to several units in the CPU. The points suitable for hardware monitor probe attachment were found among inputs of these units. Due to varying component and line delays, the values of opcode bits do not change simultaneously at all monitored points. This effect was found to cause an extremely large error. More than one SUM decoder output would often be activated during execution of a single instruction. Explanation: Since some opcode bits change earlier than others, the SUM decoder decodes one or more wrong values before the correct value is reached. This is a well known problem from switching theory, called critical races [MCLU 651. To eliminate errors of this kind, the SUM decoder must not be enabled except for a period during which its inputs are static. The selection of the SUM decoder strobe required a careful analysis of Model 145 circuit operations. The 'Set-Reset Opcode register' signal is activated several times during execution of a single instruction, and the number of activations varies from one instruction to another. This signal is not therefore suitable as the decoder strobe. In addition, execution of instructions immediately following a jump out of the prefetched stream of instructions proceeds in a special way. These special cases eliminate the possibility of assigning any of the processor internal signals directly

to the function of the decoder strobe. Strobe pulses had to be generated artificially. The pulse generator, built out of the SUM components, was triggered by the 'I-cycle' signal (beginning of a fetch phase) and generated pulses were delayed properly to ensure that the decoder inputs had correct values.

Comparison of S 360-370 Instruction Mixes From Different Installations

Table B.4 compares utilization of various subsets of instructions at different S/360-370 installations. The RCA Series 70/45 was included because its instruction format is identical to the IBM 360 instruction format. A dash in a table entry means that no information about the associated group was found in the corresponding reference. The Model 75 [ASCH 711 is oriented more toward arithmetic computation than the Model 145 at Stanford. Otherwise their instruction mixes compare quite closely. The Model 91 at UCLA handles a completely different load, based heavily on floating-point arithmetic, which is the strongest feature of this model. The RCA instruction mix was obtained from the instruction trace of a set of selected programs [WIND 731. Relatively high utilization of decimal instructions is probably due to heavy bias toward COBOL programs in the monitored set of programs.

TABLE B.4: Instruction mixes at different installations using the S/360-370 instruction repertoire.

Operation type	Installation/Model			
	Stanford University 370/145	Argonne National Laboratory 360/75	UCLA 360/91	RCA Laboratories 70/45
Integer load, store, arithmetic	26.51%	50.85%	25.21%	25.7%
Floating-point load, store, arithmetic	0.52%	2.82%	28.62%	
Decimal	0.06%			5.0%
Branch	32.74%	26.04%	18.30%	34.2%
Logical, compare, move	18.97%	17.15%	13.41%	17.1%
Control, I/O	0.54%	0.37%		

APPENDIX C. S/370 MODEL 145 MICROCODE PRINCIPLES

The S/370 Model 145 Central Processing Unit, IBM 3145, illustrated schematically in Figure C.l, is a microprogrammable processor. All CPU and channel operations of the S/370 Model 145 are controlled by micro-programs resident in reloadable control store (RCS). Microprograms are entered into the control store from the console disk file. Such operation is called the Initial Microprogram Loading **(IMPL).** Microcode may be altered during **IMPL** only. Control storage is divided into 64-word modules; 32-bit microwords, called control words, are composed of several highly encoded control fields. Decoding of these fields takes place in the C-register that directly activates circuit control lines. Each control word specifies explicitly the next control word address (each control word has the capability to branch), but most of the control words can branch to a word in the same module only. The branch can be either to a single specified address or to a branch set. A branch set is composed of 2 to 16 control words stored sequentially. Individual control words in a branch set are identified by a branch leg. The most significant four bits of a control word describe the control word type.

S/370 Model 145 Control Words

Branch and module switch word: Allows for up to four-way branching to any word in control storage.

Branch word: Provides for up to sixteen-way branch in control storage, may set or reset specified local storage or external word bits.

Branch and link or return word: Saves and restores status in connection with subroutine execution.

Word move: Moves a full word or selected bytes of a word between local storage or external locations.

Storage word: Moves data between the main or control storage and some working area in the CPU.

Arithmetic word: Performs arithmetic and logical functions on data from local or external storage.

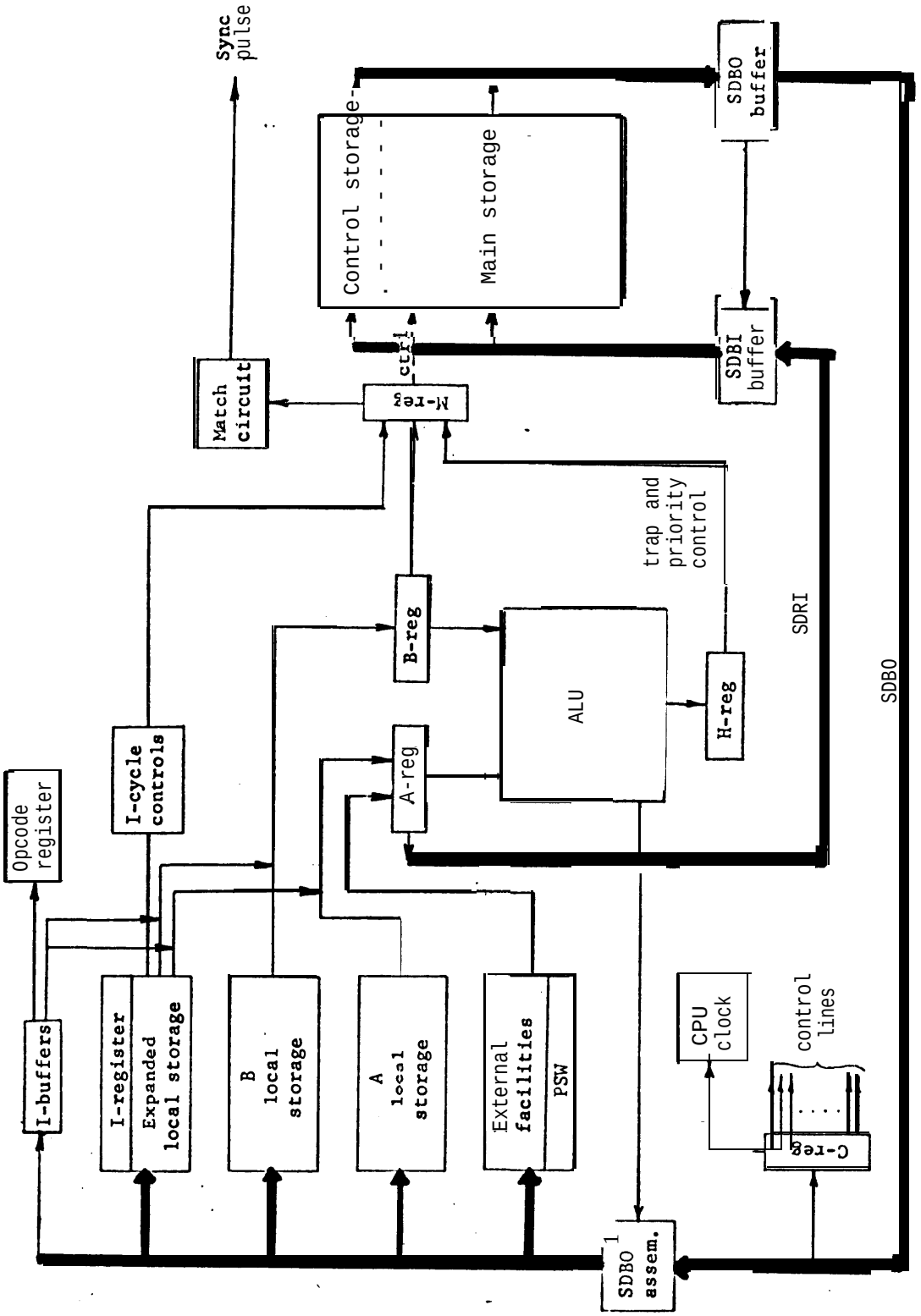


Figure C.1: Simplified S/370 Model 145 CPU data flow
(Courtesy of IBM)

Table C.l shows the control word formats for the six control word types.

CPU Timing

Each control word is executed in one CPU cycle, except for the ones that access data in the main or control storage; such control words require two CPU cycles. The CPU cycle time is variable; its length is determined by the control word type. The possible lengths of the CPU cycle are 202.5 nsec, 247.5 nsec, 292.5 nsec and 315 nsec. Timing of the IBM 3145 processor is derived from a 22.222 MHz crystal oscillator that drives a variable-cycle clock. Six basic signals are developed by this clock:

O-time	O-time delay
l-time	l-time delay
2-time	2-time delay

The n-time delay pulse is delayed 45 nsec after the n-time pulse, and the two pulses have the same length, which is either 90 or 112.5 nsec.

Instruction Execution Phases

Execution of a S/370 (machine) instruction is performed in two phases: the Instruction phase (I-phase) and the Execute phase (E-phase). The I-phase is speeded up by special hardware for automatic instruction prefetch. The I-buffers are used to hold the current instruction plus the next instruction doubleword. If at the beginning of the I-phase the instruction scheduled for execution is not fully contained in the I-buffers, this phase will start with instruction fetch. Such a situation occurs usually after execution of an Execute instruction or a successful branch. Automatic prefetch is initiated during the execute phase if it is detected that the next instruction is not fully contained in the I-buffers. The exact processing during the I-phase is further determined by the instruction type, as shown in Figure C.2. Instruction decoding is performed via a

Table C.1: Control word formats for the S/370 Model 145
(Courtesy of IBM)

	C0	C1	C2	C3
	3 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
BRANCH AND MODULE SWITCH	3 0 0 0 | Br Hi	Br Word | Byte | Dest	Module	Next | Br Lo
BRANCH	3 0 0 1 | Br Hi	Br Word | Byte | K-H/L	OK | S/R | Srce | K	Next | Br Lo
BRANCH AND LINK	3 0 1 0 | L/R | Br Hi	Link | Insert	M/Module	Next | P | Br Lo
WORD MOVE VERSION 0	0 0 1 1 | 0 | Br Hi	Sink | Insert	Source | Mask	Next | ST OP | Br Lo
WORD MOVE VERSION 1	0 0 1 1 | 1 | Br Hi	Source | Insert	Sink | Mask	Next | ST OP | Br Lo
STORAGE K-ADDRESS	0 1 | Form | Br Hi	Data | 0 0 | Mode	Address | K	Next | L/R | Br Lo
STORAGE NON K-ADDRESS	0 1 | Form | Br Hi	Data | Update ≠ 0 | Stat	Address | Mode | Stat	Next | Dec | Br Lo
ARITHMETIC TYPE IO WORD VERSION	1 0 | Form | Op (>10)	A Word | A In | Stat	B Word | B In | Shift	Next | Br
ARITHMETIC TYPE IO BYTE VERSION	1 0 | Form | Op (≤10)	A Word | Byte | Stat	B Word | Byte | Hi Lo (A)	Next | Br
ARITHMETIC TYPE II BYTE ONLY	1 1 | Form | Op | A Gate	A Word | Byte | Stat	B Ford | Byte | Hi Lo (B)	Next | Br

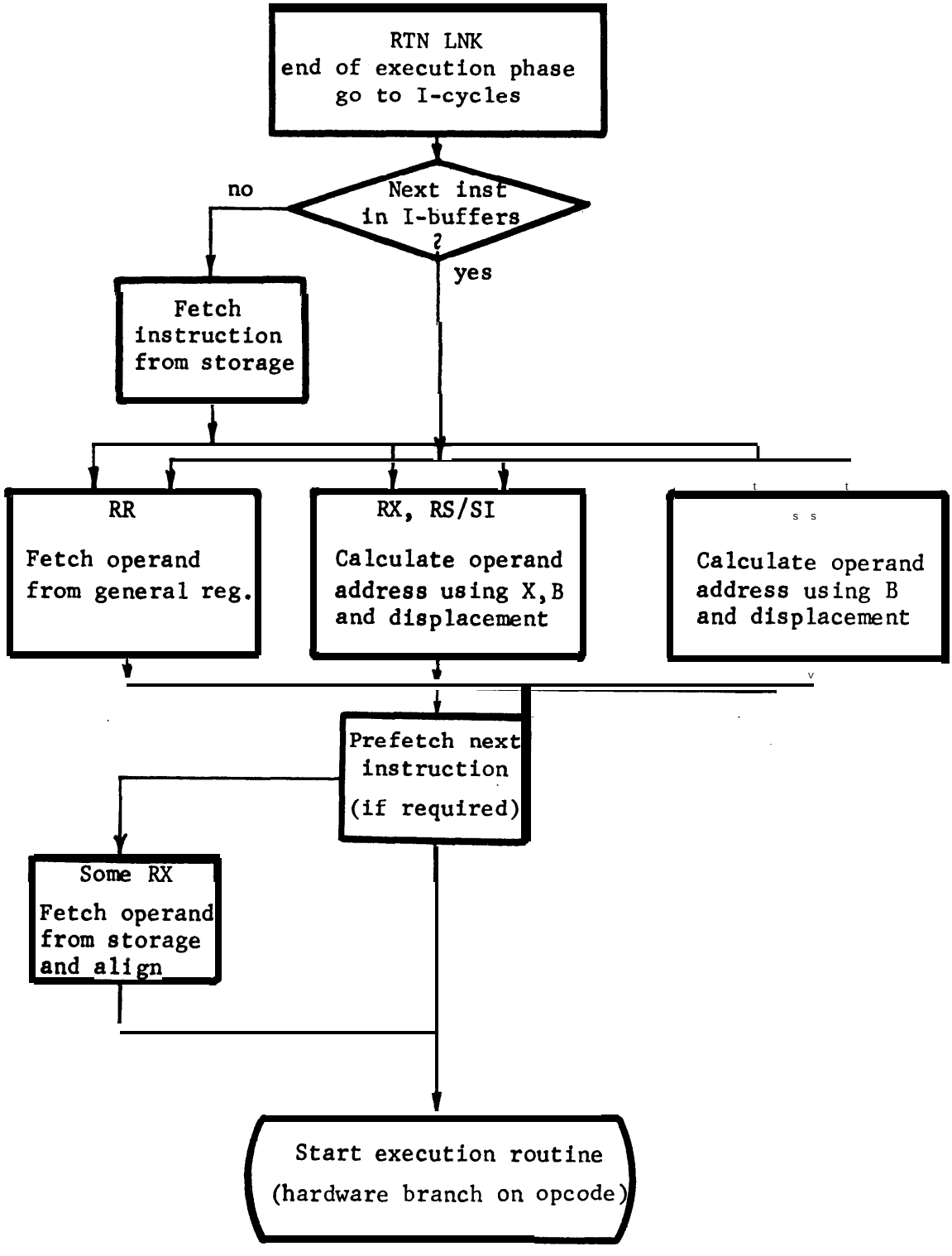


Figure C.2: Basic I-phase (I-cycle) functions

hardware-forced branch on the instruction opcode. In some cases the conditional code also participates in forming the entry address to the execute routine. Some execute routines are shared by several opcodes; for example, corresponding RR and RX operations are executed by the same routines. The last control word of the E-phase is RTN LNK (Return Link) which causes return to the I-phase routines.

APPENDIX D. MICROWORD UTILIZATION ON THE S/370 MODEL 145 AT STANFORD

This appendix presents the results of microword utilization measurements performed at Stanford. The measurement technique is described in Section 6.2. The frequencies e_j of individual microword types j measured under Case 2 (CPU is busy but not assisting in any I/O operation, i.e., CPU is processing microroutines interpreting machine instructions) can be compared with the corresponding microword mix calculated as $e_j = \sum_k f_k M_{kj} \Big/ \sum_j \sum_k f_k M_{kj}$, where f_k is fhe frequency of the instruction I_k (Table B.3) and M_{kj} is the average number of microwords of the type j necessary to execute the instruction I_k (Table 9). The instructions I_k included in the calculation of the corresponding microword mix cover only 84.6% of the total instructions executed, yet the calculated and the measured microword frequencies compare closely. The CPU wait state is microprogrammed as a wait loop; since every Model 145 microword has the ability to branch, test and branch operations of the wait loop can be accomplished by one microword (Word move).

Table D.1: Microword utilization on the S/370 Model 145 at Stanford University

j	Control word (microword) type c_j	Microword frequency e_j			
		Case 1	Case 2	Case 3	Corresponding **micromix** calculated
1	Branch and modile switch	.1592	.1377	.0021	.1264
2	Branch	.0246	.0096	.0006	.0086
3	Branch and link	.0041	.0031	.0001	.0038
4	Return	.1033	.1121	.0008	.1052
5	Word move	.1625	.1712	.9912	.1614
6	Storage word read	.1414	.1494	.0013	.1458
7	Storage word store	.0261	.0299	.0004	.0326
8	Arithmetic word **fullword** operation	.1426	.1600	.0012	.1680
9	Arithmetic word byte operation	.2362	.2370	.0023	.2482
	# microwords executed	5401474253	2804990291	9987435365	
	# microwords/instruction	9.948	9.359		9.195

APPENDIX E. DERIVATION OF THE G CONDITION CONTROLLING M-PROFILE MEASUREMENTS

The value of the condition G for M-profile measurements is a function of the current instruction opcode. The Opcode register was monitored by two SUM Data Comparator probes. The A and B registers of the Data Comparator were set such that $A \geq X > B$ would be true for all opcodes monitored together as a set. (When monitoring a single opcode X_1X_2, the contents of these registers were: $A = 0000X_1X_2$, $B = 0000X_1(X_2-1)$.) The Data Comparator thus acted as an instruction selector. A 'Set-Reset Opcode register' signal was used to gate the contents of the Opcode register to the Data Comparator X register for comparison. This signal becomes active several times during execution of a single instruction; this property made measurements of instruction utilization complicated (Appendix B), but did not cause any problems here. The Data Comparator $(A \geq X > B)$-latch is set the first time a match is detected and remains set until an opcode value that does not satisfy the condition $A > X > B$ is loaded into the Opcode register. The instruction selection procedure is illustrated in Figure E.1.

The Opcode register is loaded from the I-buffer. The I-buffer is a doubleword register used to hold prefetched instructions. Normally, the Opcode register is set to the value of the next instruction opcode at the beginning of the Execute phase of the current instruction I_n (Figure E.2a). But if the next instruction I_{n+1} is not fully contained in the I-buffer, the Opcode register load operation is suppressed and the next I-phase has to start with a fetch operation. The Opcode register is then loaded when the fetch completes (Figure E.2b). The fetch operation for such instruction cannot be monitored, since it is not known during this

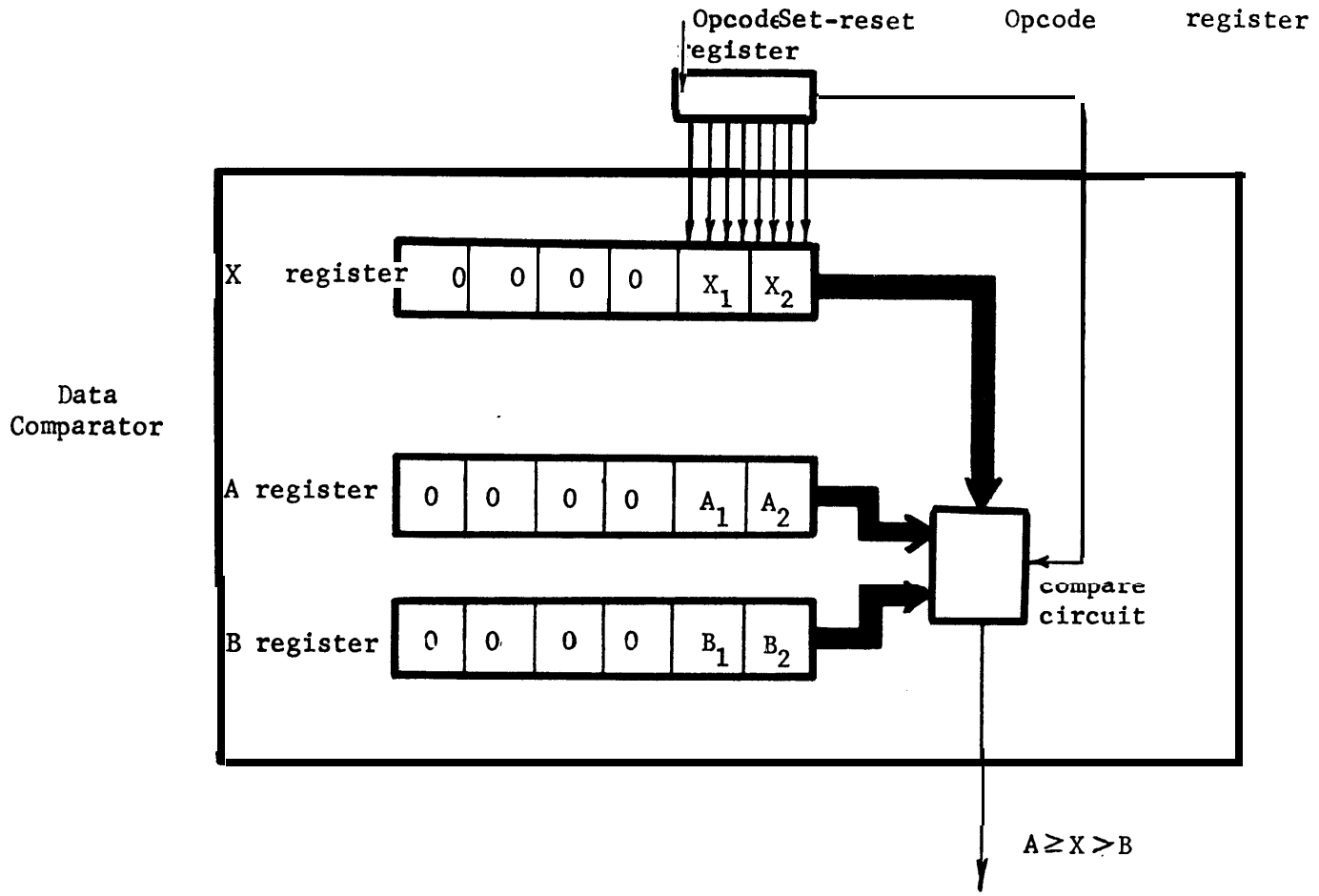
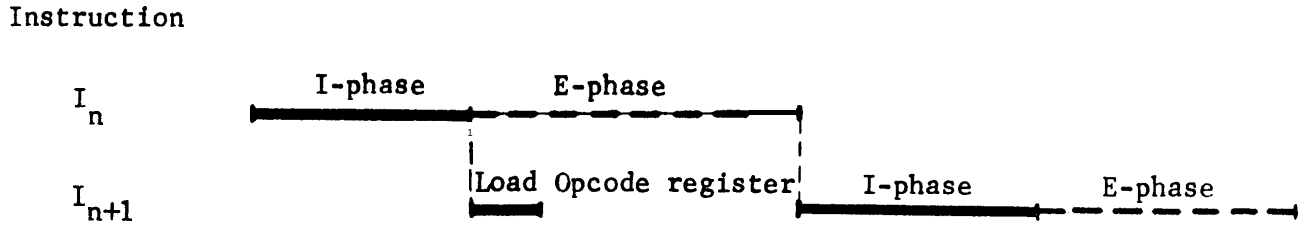
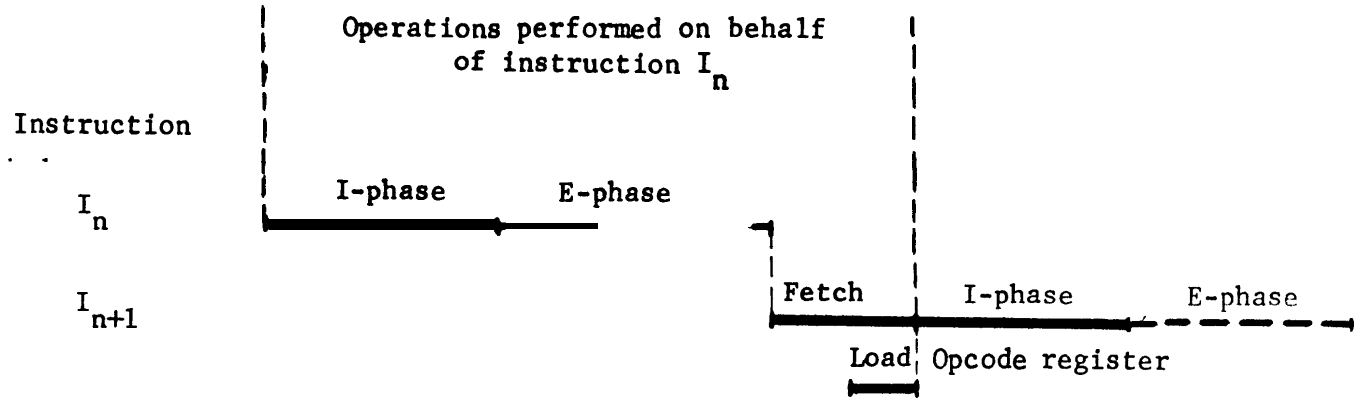


Figure E.1: Selection of instructions for M-profile measurements



(a) Instruction I_{n+1} fully contained in the I-buffer



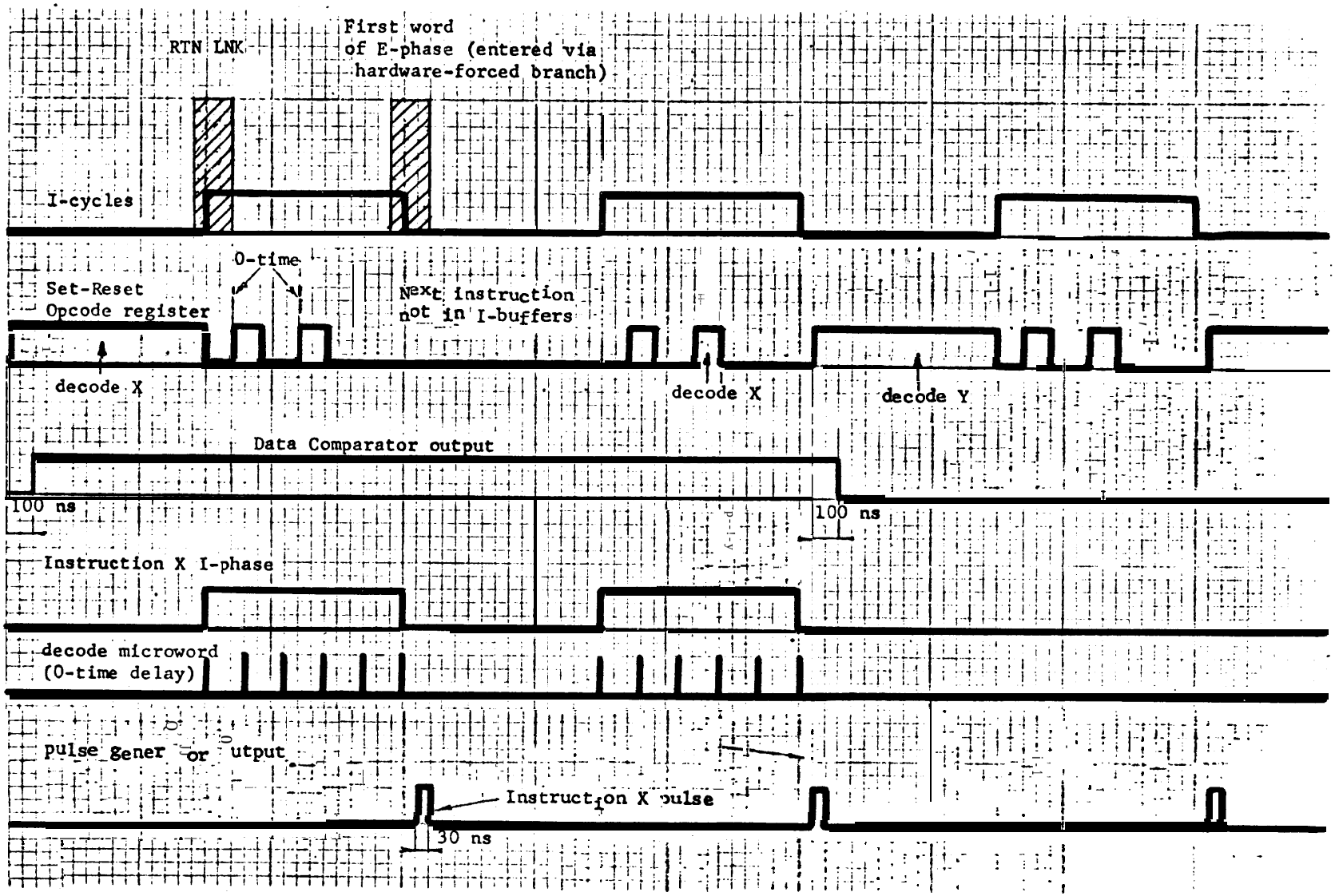
(b) Instruction I_{n+1} not fully contained in the I-buffer

(I-phase starts with instruction fetch)

Figure E.2: Timing chart for Opcode register loading

operation what the instruction opcode is; the state of condition G cannot be resolved until after the Opcode register has been properly loaded. Such a situation arises mainly after a successful branch, when the stream of prefetched instructions has to be reinitialized. So it can be said that the time delay in the opcode decoding process (fetch time) for the instruction I_{n+1} is a direct consequence of the operation performed by the I_n instruction and as such it can be taken to be a part of the time needed to execute the instruction I_n. This relationship is illustrated graphically in Figure E.2b.

The monitor also has to count how many instructions of the monitored type are executed. The value of the condition G alone cannot be used for this purpose; if two or more instructions of the type selected for analysis are executed consecutively, G remains true for the total period it takes to execute the entire sequence. The I-cycle signal was used again to trigger the pulse-generation circuit built of the SUM logic components. The output of the pulse generator was AND-ed with the output of the Data Comparator to produce one pulse each time an instruction of the monitored type was executed. Figure E.3 shows the relations among the monitored signals and the signals generated by the SUM.



instruction X - type of instruction selected
for monitoring
instruction Y - all other instructions

Figure E.3: Timing chart of signals generated by the S/370 Model 145 and the SUM logic circuits

Bibliography

ANDE 67 Anderson, S. F., Earle, J. G., Goldschmidt, R. E., Powers, D. M., "The IBM System/360 Model 91: Floating-Point Execution Unit," IBM J. Res. Dev., Vol. 11, No. 1, January 1967, pp. 34-53.

ARND 72 Arndt, F. R., Oliver, G. M., "Hardware Monitoring of Real-Time Computer System Performance," Computer, Vol. 5, No. 4, July/August 1972, pp. 25-29.

ASCH 71 Aschenbrenner, R. A., Amiot, L., Natarajan, M. K., "The Neurotron Monitor System," AFIPS Proc. FJCC, 1971, pp. 31-37.

BALY 71 Bailey, W. O., "The Processor Figure of Merit," Honeywell Computer Journal, Vol. 5, No. 4, 1971, pp. 201-204.

BELC 71 Bell, C. G., Newell, A., Computer Structure: Readings and Examples, McGraw-Hill, Inc., 1971.

BONN 69 Bonner, A. J., "Using System Monitor Output to Improve Performance," IBM Systems Journal, Vol. 8, No. 4, 1969, pp.290-298.

BORD 71 Bordsen, D. T., "Univac 1108 Hardware Instrumentation System," Proc. ACM SIGOPS Workshop on System Performance Evaluation, April 1971, pp. 1-28.

BUSS 70 Bussell, B., Koster, R. A., "Instrumenting Computer Systems and their Programs," AFIPS Proc. FJCC, 1970, pp. 525-534.

COOK 70 Cook, R. W., Flynn, M. J., "System Design of a Dynamic Microprocessor," IEEE Transactions on Computers, Vol. C-19, No. 3, March 1970, pp. 213-222.

DRUM 69 Drummond, M. E., Jr., "A Perspective on System Performance Evaluation," IBM Systems Journal, Vol. 8, No. 4, 1969, pp. 252-263.

FOST 71A Foster, C. C., Gonter, R. H., "Conditional Interpretation of Operation Codes," IEEE Transactions on Computers, Vol. C-20, No. 1, January 1971, pp. 108-111.

FOST 71B Foster, C. C., Gonter, R. H., Riseman, E. M., "Measures of Op-code Utilization," IEEE Transactions on Computers, Vol. C-20, No. 5, May 1971, pp. 582-584.

IBM 68 "IBM System/360 Principles of Operation," IBM Systems Reference Library, GA22-6821-7, September 1968.

IBM 70 "IBM System/370 Principles of Operation," IBM Systems Reference Library, GA22-7000-0, June 1970.

IBM 71 "An Introduction to Microprogramming," IBM Data Processing Techniques, GF20-0385-0, December 1971.

IBM 72 "IBM System/370 Model 145 Functional Characteristics," GA24-3557-3, August 1972.

JOHA 71 Johnson, A. M., "The Microdiagnostics for the IBM S/360 Model 30," IEEE Transactions on Computers, Vol. C-20, No. 7, July 1971, pp. 798-808.

KNIG 66 Knight, K. E., "Changes in Computer Performance: A Historical Review," Datamation, Vol. 12, No. 9, September 1966, pp. 40-54.

LUCA 71 Lucas, H. C., "Performance Evaluation and Monitoring," Computer Surveys, Vol. 3, No. 3, September 1971, pp. 79-90.

MCLU 65 McCluskey, E. J., Introduction to the Theory of Switching Circuits, McGraw-Hill, Inc., 1965.

MURP 69 Murphy, R. W., "The System Logic and Usage Recorder," AFIPS Proc. FJCC, 1969, pp. 218-229.

ROBE 72 Roberts, L. S. "Performance Measurement with Microcode," presented at the Seminar on Computer System Performance Measurements, Whippany, New Jersey, June 14-15, 1972.

ROSI 69 Rosin, R. F., 'Contemporary Concepts of Microprogramming and Emulation," Computing Surveys, Vol. 1, No. 4, December 1969, pp. 197-212.

RUUD 72 Ruud, R. J., "The CPM-X - A System Approach to Performance Measurement," AFIPS Proc. FJCC, 1972, pp. 949-957.

SAAL 72 Saal, H. J., Shustek, L. J., "Microprogrammed Implementation of Computer Measurement Techniques," Proc. ACM 5th Annual Workshop on Microprogramming, September 1972, pp. 42-50.

SAL1 73A Salisbury, A. B., "A Study of General-Purpose Microprogrammable Computer Architectures," Technical Report No. 59, Digital Systems Laboratory, Stanford University, Stanford, California, July 1973.

SAL1 73B Salisbury, A. B., "The Evaluation of Microprogram Implemented Emulators," Technical Report No. 60, Digital Systems Laboratory, Stanford University, Stanford, California, July 1973.

SUM 70 System Utilization Monitor User's Manual, Computer Synectics, Inc., Santa Clara, California, 1970.

SUM 7oc "Model 3024 24 Bit Comparator II," System Utilization Monitor, Computer Synectics, Inc., Santa Clara, California, September 1970 (preliminary draft).

TUCK 71 Tucker, A. B., Flynn, M. J., "Dynamic Microprogramming: Processor Organization and Programming," Communications of the ACM, Vol. 13, No. 4, April 1971, pp. 240-250.

WILK 51 Wiikes, M. V., "The Best Way to Design an Automatic Calculating Machine," Manchester University Computer Inaugural Conference, July 1951.

WILN 72 Wilner, W. T., "Design of the Burroughs B1700," AFIPS Proc. FJCC, 1972, pp. 481-497.

WIND 73 Winder, R. O., "A Data Base for Computer Performance Evaluation," Computer, Vol. 6, No. 3, March 1973, pp. 25-29.