SU-SEL 76-026 SU-326-P.39-11

THE STANFORD EMULATION LABORATORY

by

Michael J. Flynn
Lee W. Hoevel
Charles J. Neuhauser

June 1976

Technical Report No. 118

The work described herein was supported in part by the U.S. Energy Research and Development Administration under Contract No. AT(04-3) 326 PA 39.

DIGITAL SYSTEMS LABORATORY
STANFORD ELECTRONICS LABORATORIES
STANFORD UNIVERSITY . STANFORD, CALIFORNIA

SU-SEL 76-026

SU-326-P.39-11

THE STANFORD EMULATION LABORATORY

by

Michael J. Flynn, Lee W. Hoevel and Charles J. Neuhauser

June 1976

Technical Report No. 118

Digital Systems Laboratory

Stanford Electronics Laboratories

Stanford University

Stanford, California 94305

The work herein was supported in part by the U.S. Energy Research and Development Administration under contract #AT(04-3) 326PA39.

Digital System Laboratory

Stanford Electronics Laboratories

Technical Report No. 118

June 1976

THE STANFORD EMULATION LABORATORY

by

Michael J. Flynn, Lee W. Hoevel and Charles J. Neuhauser

ABSTRACT

The Stanford Emulation Laboratory is designed to support general research in the area of emulation. Central to the laboratory is a universal host machine, the EMMY, which has been designed specifically to be an unbiased, yet efficient host for a wide range of target machine architectures. Microstore in the EMMY is dynamically microprogrammable and thus is used as the primary data storage resource of the emulator. Other laboratory equipment includes a reconfigurable main memory system and an independent control processor to monitor emulation experiments. Laboratory software, including two microassemblers, is briefly described.

Three laboratory applications are described: (1) A conventional target machine emulation (a system 360), (2) 'microscopic' examination of emulated target machine I-streams, and (3) Direct execution of a high level language (Fortran II).

Keywords: Emulation, Universal Host Machine, Dynamic Microprogramming, Directly Executed Languages,

--- TABLE OF CONTENTS ---

1. Introduction

In recent years attention in microprogramming has been on the production environment. Machine designers have viewed microprogramming as a means of easing implementation problems and providing a limited amount of flexibility to accomodate compatability requirements among various machines in a manufacturer's line. Microprogrammable machines designed for this environment are thus fairly specific to their task and usually exhibit little flexibility with respect to the full range of machine architectures.

In this paper microprogramming is examined in a different environment, that of research and education. Central to this is a 'universal host machine', that is, a machine which is capable of efficiently emulating a wide variety of 'target' machine organizations. By taking advantage of fast read write memory, the host machine provides the user with an architectrually 'soft' system which he may structure via microprogramming to his specific requirements. In the following we discuss the Stanford host machine, EMMY, its software tools, and the current status of several typical emulation experiments.

1.1 The Research and Educational Environment

The principal environment is an experimental one, emphasizing studies on:

1) Emulation of novel, obsolete and 'paper' machines for the purpose of exposition,

2) Analysis of architectures via emulation and dynamic acquisition of performance data, and

3) Development of soft architectures which efficiently reflect artifacts of higher level languages in their use of hardware resources.

With respect to the first area, the laboratory will make available a wide variety of machines. In addition to presenting real but outdated machines (e.g. IBM 1401) the emulation laboratory provides an effective realization of 'paper' machines, such as MIX, or the users' own creations. Aside from economic considerations, the laboratory standardizes access to these machines and thus allows the user to be more concerned with basic architectural Eeatures of the target machine rather than bookkeeping details. In this environment the ease with which laboratory hardware and software can be used is a more important consideration than speed.

Architectural analysis implies direct insight into the instruction characteristics of target machine architectures. Initially, the experimenter constructs an emulator for a target machine of interest. This emulator is then be modified to collect statistics relating to the target machine's performance during emulated target machine execution. We expect this mode of instruction stream examination to be more effective and less costly than current software (i.e. trace tape evaluations) or hardware monitoring techniques. For this application we are particularaly interested in affording the experimenter efficient emulator representation and respectable target machine performance.

Another experimental situation is that of developing 'soft' machine architectures. In particular it has been demonstrated that many computational situations proceed more efficiently when executed at a relatively high interpretive level rather than via the usual compilation and machine language interpretation procedure [1]. This direct execution is accomplished by designing an intermediate or 'Directly Executed Language (DEL)' to stand between the microlanguage associated with the hardware resources and the high-level language associated with the problem environment.

1.2 Discussion of Laboratory Characteristics

Central to the emulation laboratory is the use of a 'universal' host machine. Such a machine should be capable of efficiently emulating a wide range of conventionally structured 'target' machines. With the recent introduction of high speed read-wr i te LSI memories (both bipolar and fast MOS), the possibility of such a host machine with flexibly programmed state transistions has become a reality. Correspondingly, two new architectural concepts have been developed: the 'soft' computer organization and 'dynamic microprogramming'.

Soft architectures, such as the Nanaodata QM-1 [2] and the Burroughs B1700 [3], are machine organizations which allow the user to structure the primitive computational and storage resources available in a way best suited to a particular target machine's requirements. In particular we may identify the following desirable characteristics of such soft machines*

1) Bit field handling and selection,
2) Flexible, high speed shift and rotate capability,
3) Extensive bit testing, and
4) Flexible data path specification (usually via residual control).

Our previous investigations of host machine architecture [4,5] have shown these features to be useful in both emulation and interpretive computation situations.

Soft architectures can be enhanced through the use of 'dynamic microprogramming' [7] in which the read-write control store is used, not only for microprogram storaqe, but also as the primary data storage resource of the system. This technique is a natural technological extension of previous control store applications. Briefly we may classify the use of microprogramming in current target machine architectures as being either implementation or extension oriented. Implementation oriented systems, such as the 361) series, make use of ROM control stores for the purpose of structuring the machine desiqn. ROM memory by nature is not very flexible and changes, while clearly less costly than in corresponding hardwired implementations, must still be infrequent. The HP2100 [9] series of microproqramable machine typifies the extension oriented use of microprogrammed orqanizations. RAN control store in these machines is easily written via an external I/O bus but is not conveniently accessible to the microprogram in real time, thus preventing its effective use as a data storage medium.

In a dynamically microprogrammed machine the control store is directly accessible by the microprogrammer for data storaqe in much the same sense that main memory is accessible to the assembly lanquage programmer in conventional machines. In addition, we have assumed that this store is accessible to external devices on a shared basis via the processor bus system. Thus, we term this storage level 'microstore' since its function goes well beyond control. The combination of these accessing capabilities provides several advantages:

1) Microstore now becomes the primary storage medium (with respect to access time) for data as well as microprogram. This allows the microprogrammer a continuum of trade-offs between microprogram and data storage.

2) Since the microstore may be externally loaded via the system bus it is possible to quickly load and replace different emulators.

3) The two level random access memory structure (i.e. main memory and microstore) allows the microstore to function as an ‘explicit cache’. Access to target machine data and instructions can be shortened by explicitly maintaining locally used memory storage contents in control store. In a later section we discuss the efficient use of this capability under an interpretive computation discipline.

Our notion of the emulation laboratory is that it should provide the user with a system which he can structure to a particular emulation task. To complement this capability the user must be able to observe the operation of the target machine emulator at a fairly detailed level. Therefore, the laboratory should include both software and hardware tools to support user access to, and control of, the host machine. Specifically, he should be able to access all system storage locations such as main memory and microstore. The Stanford laboratory supports access by making use of a separate laboratory control processor which is independent of the host machine. This arrangement affords the experimenter a flexible observation tool with the added advantage of allowing the host machine to be used exclusively for the emulation task.

2. Tools

In the following section we discuss the hardware and software tools available to the experimenter in the Stanford Emulation Laboratory.

2.1 Hardware

The laboratory host processor, the EMMY [26], is based on a 32 bit wide microstore word which serves both as the data and microinstruction word width. The microstore itself contains 4096 words and has a basic access time of 60 nsec and a cycle time of 180 nsec. Within the CPU are eight general purpose registers, one of which contains the machine state word. External access is via a 32 bit data bus which supports direct access to 16M locations. All address and data arithmetic in the CPU is two's complement.

2.1.1 The Laboratory System

Figure 2.1 schematically illustrates the laboratory and the interconnection of the various units. Essentially, there are three sub-systems each organized about one of the bus systems:

1) Host bus system,
2) Auxiliary bus system, and
3) Control processor system.

Primary communication between laboratory units takes place over the host bus system which interconnects the EMMY CPU and its immediate peripherals [29]. This bus system is a 32 bit wide, bidirectional communication path which makes use of a fully interlocked asynchronous control discipline. Under this discipline the bus appears as a resource, and any electrically and logically capable bus unit may seize the bus for the purposes of transferring data. Thirty-two bits of address information consisting of an eight bit command, an eight bit unit address and a 16 bit internal address may be sent simultaneously to all bus units. Thus 16M locations are directly accessible to the microprogrammer.

The host bus system consists of the following units:

1) EMMY CPU and 4K microstore,
2) 64K byte main memory system,
3) Auxiliary bus interface,
4) Maintenance console,
5) Control processor interface, and
6) Block access controller.

in addition to acting as a master device and activelv accessing the host bus, the EMMY CPU and its associated microstore may act as a 'slave' unit and be accessed by other bus units. Specifically, all microstore locations and the eight CPU general purpose registers may be read or written by the control processor, thus facilitating the loading of microprograms and the control cf experiments.

Main memory may be configured to represent the main memory system of the target machine [30]. One, two, three and four byte access options are available. Data reformatting includes left or riaht justification and sign extension. Since the main memory is able to handle these elementary, repetitive operations in parallel with EMMY processing, emulator performance is enhanced.

A block access controller allows transfer of data blocks between any two bus system units. This device supports explicit paging of data between microstore and main memory.

The host bus system will communicate with conventional peripheral units situated on the auxiliary bus system via a bus translation device. Auxiliary bus peripherals will include disk and magnetic tape storage units which will stand in the place cf similar target machine units. Currently, we are planning to design a bus translator to interface the host bus-with the PDP-11 Unibus (P) system [39], since the host bus is electrically and logically similar. This interface will allow us to use relatively low cost, mass produced peripheral units in our system. In using the laboratory the experimenter will access these peripheral units through microprogrammed subroutines which will make these units appear to be functionallv equivalent to those of the particular target machine being emulated.

An important user feature of the laboratory, particularaly in the interactive educational research environment, is the independent control processor. In our current system the control processor is a Datapoint 2200 (R), which is a character oriented 'intelligent' terminal. This processor has its own internal memory and communicates with relatively low speed peripherals via its own byte width bus system. Since the control processor has direct access to the host bus storage locations, particularly main memory and microstore, the experimenter has a convenient means of initializing, controlling and analyzing the results of an experimental run. By using this independent access capability an experiment can be partitioned so that the EMMY CPU deals mainly with its emulation aspects and the Datapoint 2200 is used for control purposes.

2.1.2 The EMMY CPU

The EMMY CPU was designed, within relatively severe cost constraints, to be an efficient, yet unbiased, host machine.

Flexibility is achieved through the use of microprogramming and low cost has been maintained by relying heavily on LSI in both the memory and processor design.

2.1.2.1 General Principles

Figure 2.2 illustrates schematically the funtional aspects of the host machine organization. In order to make efficient use of the rather short microinstruction word (32 bits) we have introduced a degree of parallelism internal to each microinstruction. This is accomplished by partitioning the primitive host machine resources into three classes and controlling each class of resources with a separate finite state machine. Each finite state machine receives control information from the current microinstruction word and is responsible for controlling resources associated with one instruction class. Resources are partitioned in a manner which rouqhly reflects the requirements of conventional target machines [6]:

1) T-machine - controls functional resources,
2) A-machine - controls storage resources, and
3) I-machine - controls microinstruction sequencing.

In addition, there is a fourth machine which is responsible for bus oriented transactions and operates independently of the other three.

The eight general purpose registers are the primary state storage mechanism of the host machine. The T-machine, in particular, is only capable of manipulating register data, while the A-machine (which is able to access registers, microstore and the system bus) is used to move data between the registers and other storage resources. State information to explictly direct the microinstruction sequencing is maintained in register 0 of the general purpose register file and is accessible to the I-machine.

Microinstructions in the host machine are divided approximately in half so that one half (the T control field or TCF) controls the T-machine and the other half (the A control field or ACF) controls the A-machine. The TCF is 14 bits in width and the ACF is 18 bits in width. Either half may be used optionally to modify the normal implicit sequential microinstruction fetching of the I-machine. Additionally, the ACF half may be used as immediate data by the T-machine resources on certain instructions. Thus, we have what amounts to five general types of microinstruction formats:

	TCF Half	ACF Half
1)	T-control	Immediate data
2)	T-control	A-control
3)	T-control	I-control
4)	I-control	A-control
5)	I-control	I-control

By structuring the microinstruction set in this manner we have obtained two objectives: explicit and independent control of the primitive resources and low level parallelism reflecting concurrence in conventionally structured target machines.

2.1.2.2 T-machine Instructions

T-machine instructions provide the microprogrammer with the basic functional operations necessary to emulate the transformational and control aspects of a target machine. Instructions for the T-machine may be divided into five classes as follows:

1) Logical,
2) Arithmetic,
3) Shift and Rotate,
4) Extended Arithmetic, and
5) Field Insertion and Extraction.

Figure 2.3 outlines the opcode format for T-machine instructions. Functionally, these instructions require two (or sometimes three) operands and produce a single result which is returned to the register file. One of these operands is from the register file and the other may be either a register or immediate data from the ACF half of the microinstruction. For the first four instruction classes a four bit opcode further defines the operation.

Insert/Extract instructions are designed specifically for field handling and require two register operands and immediate data from the ACF half word. In these operations a source operand is rotated by a specified amount and combined under masking designated by the ACF half word with the destination operand. On Extract instructions the destination operand register is cleared before the operation takes place, with the effect that the masked field from the first operand is isolated for further processing.

Extended Arithmetic operations are single step fragments of frequently used arithmetic operations such as multiply, divide and decimal-binary conversion. Using these microinstructions iteratively the microprogrammer can build complex target machine operations efficiently.

2.1.2.3 A-machine Instructions

A-machine instructions are used by the microprogrammer to access microstore, manipulate address pointers and communicate with external devices. Four classes of A-machine instructions are defined:

1) Move data between registers and microstore,
2) Load a register with immediate data,
3) Access memory indirectly, and
4) Manipulate pointers (and test results).

Figure 2.4 illustrates the way in which the ACF half of the microinstruction word is used to specify the A-machine instruction. For the first two classes of A-machine instruction two fields are used to specify the register and a microstore address (or immediate data). For the remaining instruction classes two register operands, an opcode and an immediate data value are specified.

Indirect memory operations control the movement of data between the register file, microstore and the host bus system. When external memory operations are initiated by the A-machine they are completed independently by the host bus access machine. This is an important source of parallelism since it allows the host machine to continue processing of microinstructions. Pointer manipulation instructions perform arithmetic on the registers for address calculation.

2.1.2.4 I-machine Instructions

Fetching of the next microinstruction is controlled by the I-machine. Normally microinstructions are fetched from the next sequential microstore location. The actual location of the fetch is maintained in register 0 of the general purpose register file and thus may be modified explictly by both T- and A-machine instructions.

There are three classes of I-machine instructions determining the sequence of microinstruction fetching:

1) Conditional,
2) Branching, and
3) Looping.

The formats for these instructions are outlined in figure 2.5.

Conditional instructions are specified in the TCF half word and consist of two fields: a conditional mask field and a test specification. Within the state word (register 0) are eight bits which record various aspects of the previous microinstruction cycle. The mask field specifies which of these bits are to be examined, and the test specification field describes whether the test is valid if any or all masked bits are set and the sense of the result. If the test is valid then the A-machine instruction defined by the ACF half word is executed, otherwise, it is skipped. Thus, the microprogrammer has the capability of specifying conditional jumps, memory access, external operations and so forth.

Branching operations are defined in the ACF half word and, as indicated by their format, are similar to the conditional instructions. The result of a valid test in the case of a branch instruction is modification of the microaddress register by the amount given in the value field.

Looping is another aspect of the pointer modification instruction. In addition to the pointer modification the programmer may test the result for one of the common arithmetic condition (e.g. less than zero) and, if valid, modify the microaddress register as in the branch instruction. Thus, the microprogrammer has the capability of defining short counting loops. In fact, the emulation of a target machine multiply instruction usually requires only one microinstruction since the TCF half word may specify a single bit multiply and the ACF may specify repetition for a particular word length.

Figure 2.6 illustrates the 32 bit machine state word which is maintained in register 0. These are four fields of importance:

1) Condition codes	(8 bits)
2) Indicator codes	(8 bits)
3) State	(4 bits)
4) Micro Address Reg	(12 bits)

Condition codes reflect the results of previous processor operations and are directly testable via the conditional and branch instructions. Indicator codes are testable in the same manner but are set by the programmer and remain undisturbed by changing processor conditions. Indicator codes have proven useful for storing intermediate information about target machine conditions (such as current word alignment). Two state bits are specified in the state field which define the condition of interrupts and whether the machine is running or halted. Lastly, 12 bits are set asside as the Micro Address Register and define the next loactaion from which the I-machine will fetch a microinstruction.

2.1.2.5 Host Bus Machine

Implicit in the design of the CPU is a fourth machine, the host bus machine. This machine handles the transactions which occur between the CPU and the external devices located on the host bus. Tasks which this machine performs include:

1) CPU access (read and write) to external devices,
2) Access to CPU registers and microstore by external devices, and
3) Interrupts generated by external devices.

CPU generated accesses are handled in an overlapped fashion by this machine, that is, once a bus access is initiated, the CPU may proceed with microexecution while awaiting results. External access to CPU storage resources by other bus devices, particularly the console and control processor interface, are processed by the host bus machine on a shared basis with CPU requests.

Although many micromachines do not make allowances for direct interrupts from external sources, we felt it was necesary in the laboratory environment to provide prompt service to dynamic peripherals such as disk and drum. The interrupt mechanism is quite simple: a device requiring service gains control of the bus system as it would for a normal data transfer. It then sends an interrupt command and a CPU microstore address. In processing the interrupt the CPU stores the current contents of register 0 (the state word) in the given location and loads register 0 with the contents of the associated even-odd microstore location. This procedure thus preserves the old machine state and initiates a new state immediately.

2.1.3 Hardware Technology

The EMMY CPU, including register file and ALU, is implemented using MECL 10K (R) series ECL logic. required in electrical and mechanical design was more than offset by the speed and functional simplicity which ECL affords. The current micromemory is fast n-channel MOS structured around 1K by 1 bit chips (AMS 7001). Bus logic is open collector TTL.

Physically, the CPU is contained on one 15" x 15" wire wrapped board. The microstore requires one additional board while the bus interface cards for specific units are built on conventional 7" x 10" wire wrap cards.

2.2 Software

Software is designed to be primarily user oriented. In figure 2.7 the software environment of the initial system is illustrated. At the outset, user effort will center on the development of specific target machine emulators and target machine programs, a task which can be supported most efficiently by a remotely located interactive system. The experimenter uses the laboratory control processor to load programs from the remote data base and to monitor the experiment. If desired, results of the experiment can be returned to the remote system for 'off line' examination and analysis. In effect, the laboratory system consists of three computational devices:

1) Remote System	--	Interactive data base development
2) Control Processor	--	User oriented control functions
3) EMMY CPU	--	Target machine emulation

2.2.1 Microassemblers

Since EMMY is to be a research oriented tool available to a larger number of users, it is important to provide users with a relatively easy to use microassembler. Thus, we developed a register transfer language similar in many respects to PL360 [10]. Although this prevents a programmer from exercising explicit control over the microinstruction content at the bit level, it substantially simplifies programming. Using this language, which is called EMMYPL [31,40], it is only necessary for the microprogrammer to be aware of the general resource and memory structure of EMMY. The EMMYPL compiler then undertakes the construction of programmer specified functions in an efficient manner. Although this is done only on a local basis in the microcode, the results to date have been good enough to justify its use in situations where implementation ease is more important than performance.

In situations where performance is very important another microassembler, EMMYXL, is available [32,41]. While basically using a register transfer format, EMMYXL requires the user to state explicitly the form and function of each microinstruction. Since each 'half' of an EMMY microinstruction may be explicitly defined, the high packing density necessary for efficient code results.

Figure 2.8 shows a short example of EMMYXL and EMMYPL taken from two separate 360 series emulators. In both examples the microprogrammer deals directly with EMMY resources through the register transfer notation. In the EMMYXL example the hybrid format of the microinstruction is explict since each line of code represents one and only one microinstruction. In EMMYPL the exact nature of the microinstruction is unimportant and the microprogrammer specifies only the function desired. Thus, one line of code may represent more or less than one EMMY microinstruction.

Both microassemblers currently run on the campus computer center's 370 system.

2.2.2 Laboratory Control Program

The system also includes a control program, called EMMYOS [33], which assists the user in establishing and monitoring an experiment. The program is resident in the Datapoint 2200 and is primarily concerned with supporting communications between the user, the EMMY system hardware and external facilities, such as the university computer center. Since the control processor has direct access to the registers, microstore and memory resources of the EMMY system set up and initialization of a experiment requires no active cooperation of the EMMY.

EMMYOS makes available simple debugging aids to allow the user to examine and modify microstore and to extablish breakpoints in the microprogram. In addition, the Datapoint 2200 may be configured as a collection of low speed peripheral units for use by target machine I/O programs.

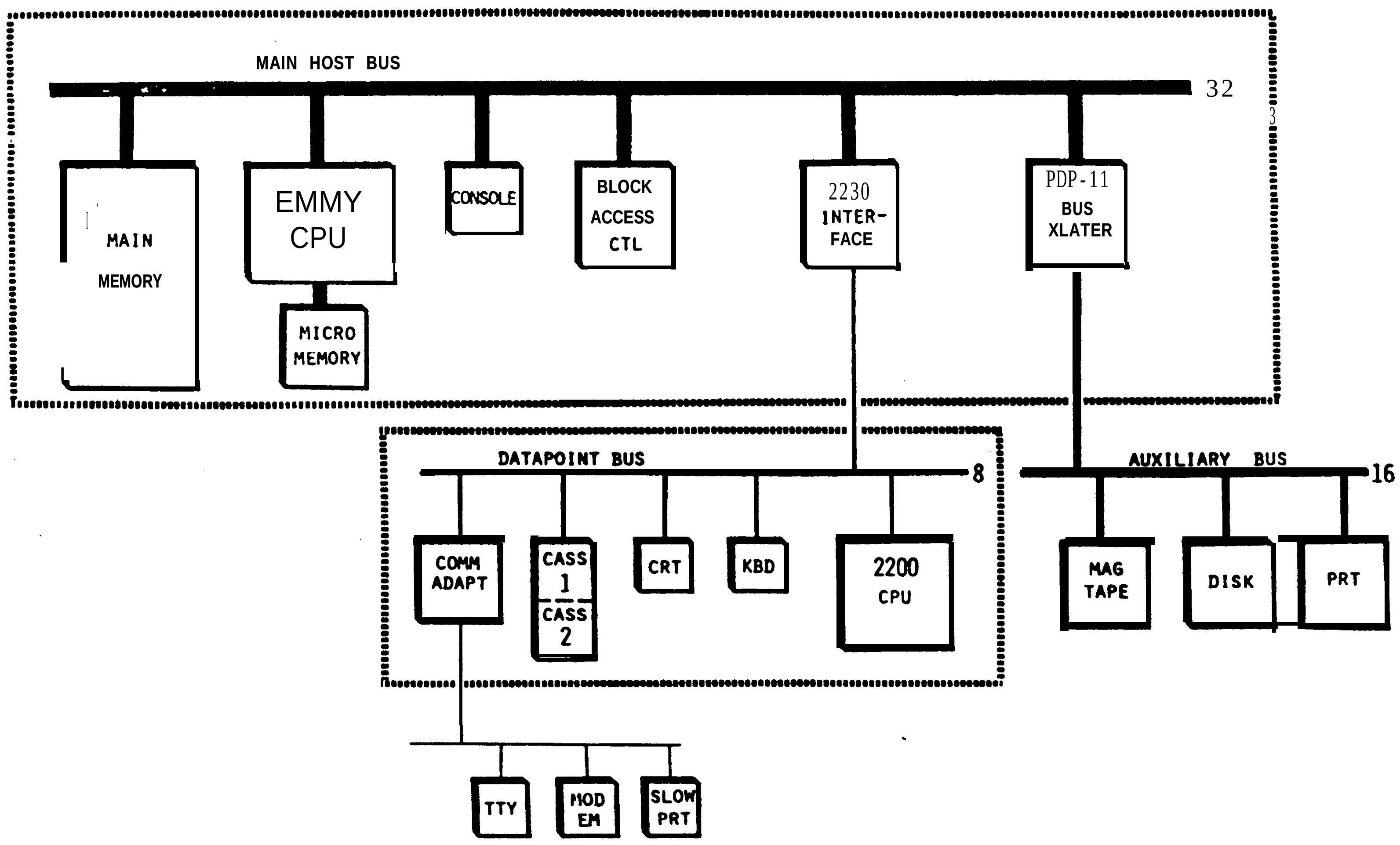
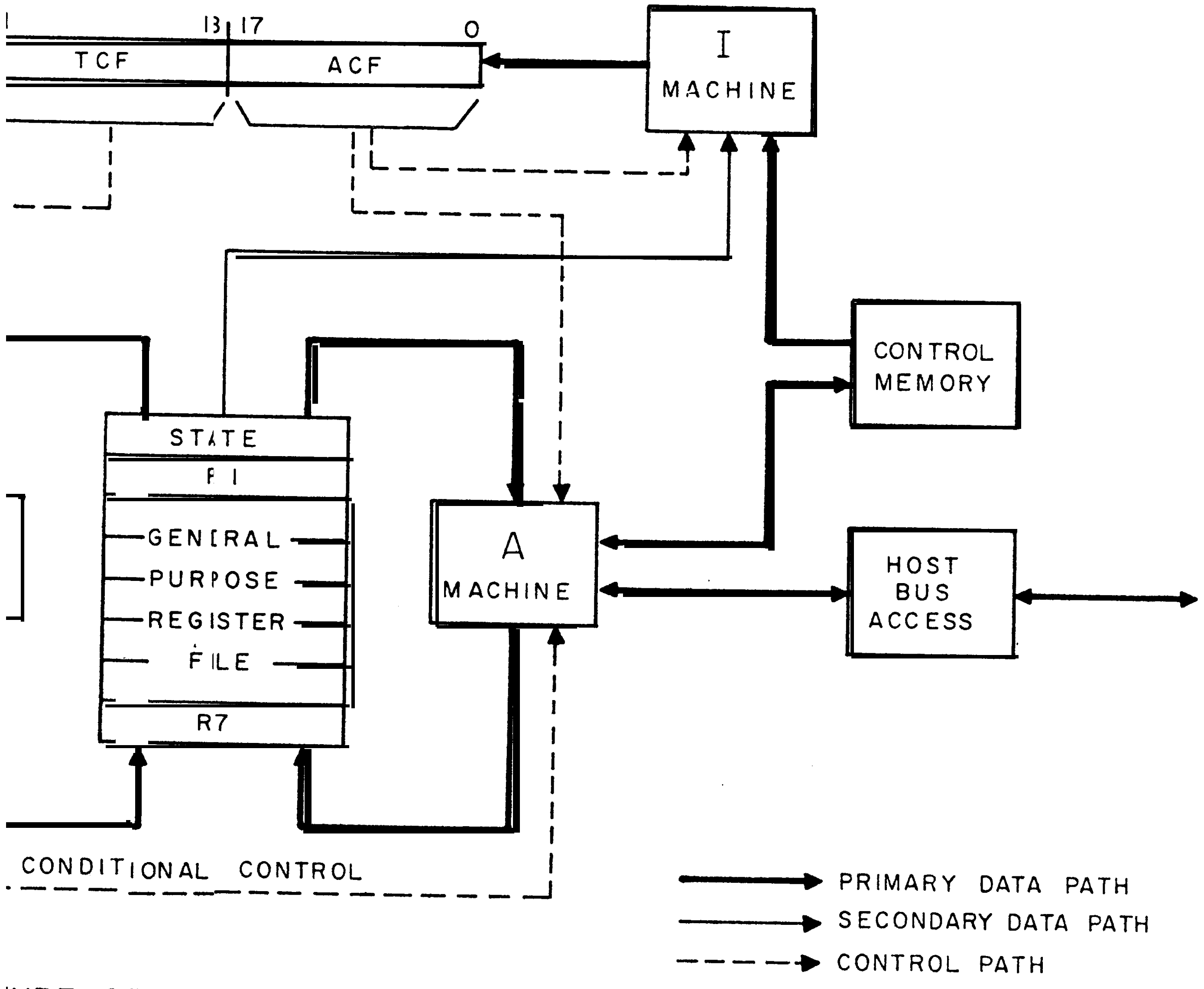
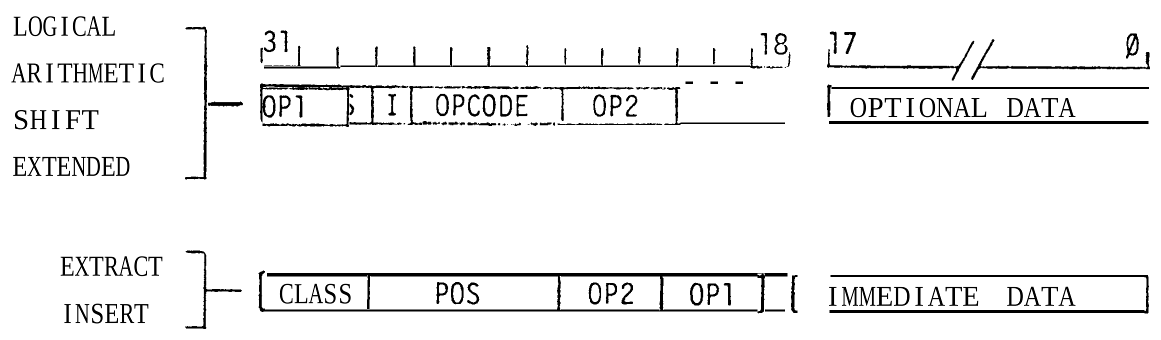


Figure: 2.1 **Struc ture** of Emulation ***Laboratory Facility***



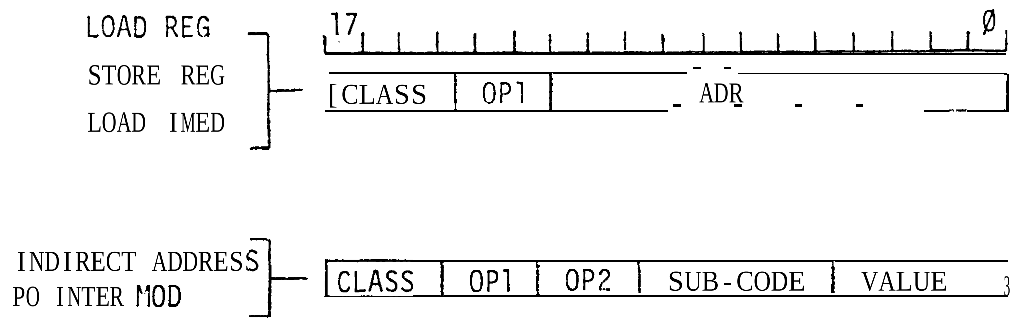
'URE OF HOST MACHINE



FIELD	SEMANTICS
CLASS	- TYPE OF T-MACHINE INSTRUCTION
OPCODE	- SPECIFIC OPERATION
OP2	- OPERAND SOURCE
OP1	- OPERAND SOURCE/SINK
POS	- ROTATE AMOUNT FOR FIELD SELECTION

T-MACHINE INSTRUCTION FORMAT

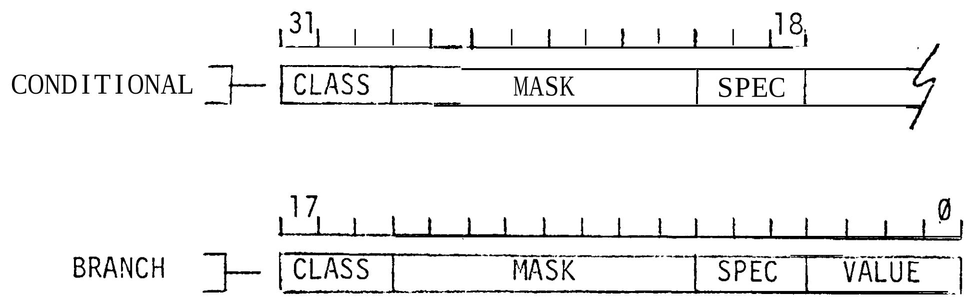
Figure 2.3



FIELD	SEMANTICS
CLASS	- TYPE OF A-MACHINE INSTRUCTION
OP1	- REGISTER POINTER
OP2	- REGISTER POINTER
ADR	- MICROSTORE ADDRESS OR DATA
SUB-CODE	- SPECIFIC A-MACHINE OPERATION
VALUE	- IMMEDIATE DATA VALUE

A-MACHINE INSTRUCTION FORMATS

Figure 2.4



FIELD	SEMANTICS
CLASS	- TYPE OF I-MACHINE INSTRUCTION
MASK	- CODE BIT MASK
SPEC	- TEST SPECIFICATION
VALUE	- MICROADDRESS MODIFIER

MACHINE INSTRUCTION FORMATS

Figure 2.5

```
          LOAD AND STORE INSTRUCTIONS

OP10:   .LOAD POSITIVE
        XR,IR << 4  ; P=M(XR)         .R2 VALUE
        PC:=PC+2    ; IRX(PC)         .NEXT INSTR
        P:=P        ; (NEGATIVE => PCOMP ) .TEST FOR NEGATIVE
                    ; M(R) = P        .STORE RESULT
        MAR:=DECODE                   .DECODE NEXT INSTR
.
PCOMP:  .COMPLEMENT NEGATIVE NUMBER
        XR:=-1      ; s=s-s
        s:=s-P      ; M(CC)=MAR       .2'S COMPLEMENT AND SET COND CODE
                    ; M(R)=S          .STORE RESULT
        (¬OVERFLOW => ; MAR=DECODE)
                    ; MAR=ARITHOFL     .HANDLE ARITHMETIC OVERFLOW
.
OP12:   .LOAD AND TEST (RR)
        XR,IR << 4 ; P=M(XR)            .R2 VALUE
        PC:=PC+2    ;IR=X(PC)           .FETCH NEXT INSTR
        P:=P        ; M(CC)=MAR         .TEST AND SET CC
        XR:=-1      ; M(R)=P            .REPLACE R1 VALUE
        MAR:=DECODE
```

EMMYXL

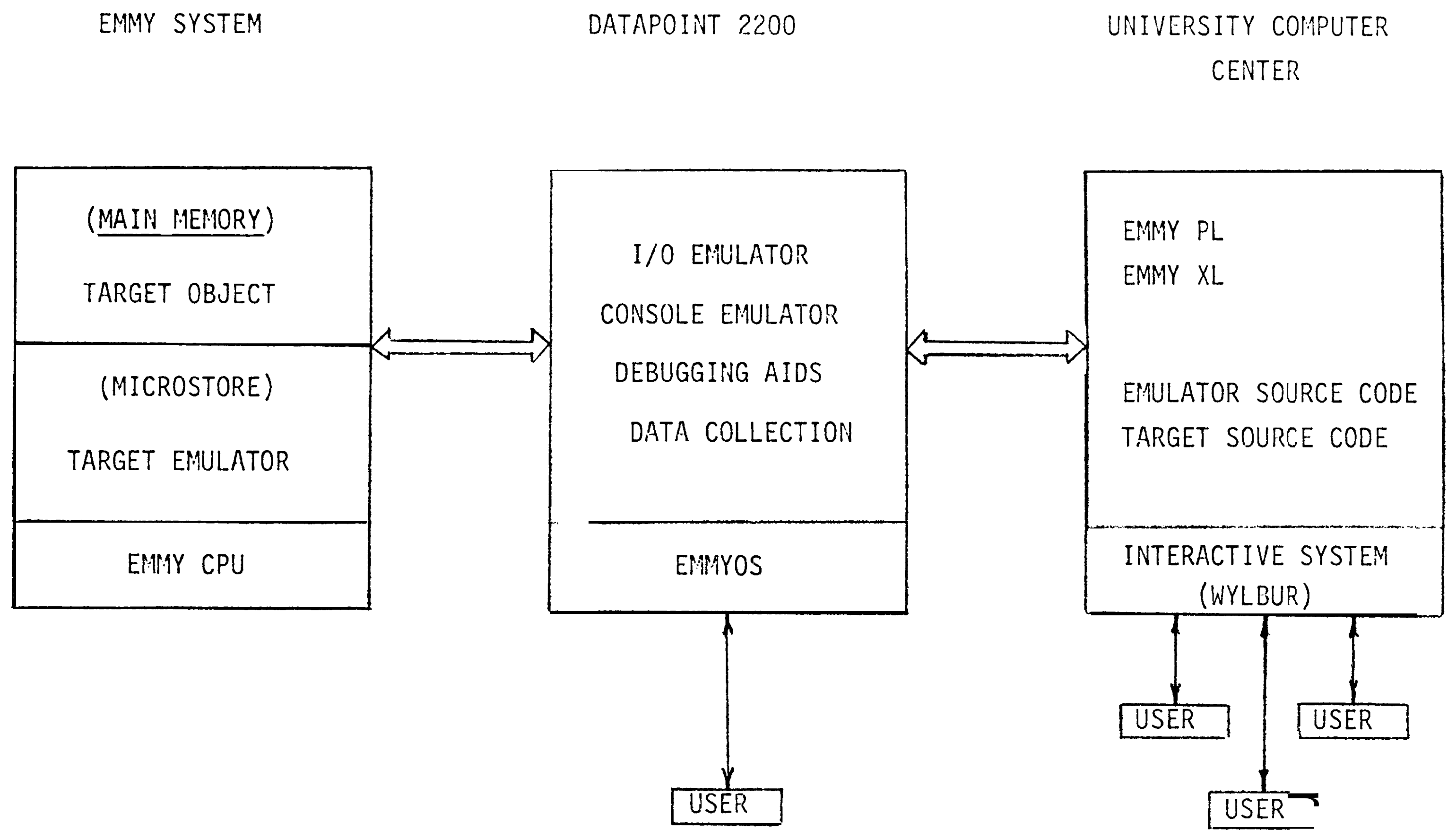
```
#10: ** LPR;
           DO
              R2:=X(R1);  R5:=M(R7);
               IF R5<0 THEN R5:=¬R5+1;
              SET_CC:  AGAIN OPWAIT:
           END;

#11: ** LNR;
           DO
              R2:=X(R1);  R5:=M(R7);
               IF R5>=0 THEN R5:=¬R5+1;
              SET_CC;  AGAIN OPWAIT:
           END;

#12: ** LTR;
           DO
              R2:=X(R1);  R5:-M(R5)+0;  SET_CC;  AGAIN OPWAIT;
           END;
```

EMMYPL

Note: Both samples of code are from the same section of two different System 360 emulators. They are nominally (but not exactly) equivalent from a functional standpoint.

Micro-assembler Examples

Figure 2.6



LABORATORY SOFTWARE ENVIORNMENT

Figure 2.7

3.0 Laboratory Experiments

3.1 'Hard' Processor Emulation

A primary objective of the emulation laboratory is to provide researchers with access to various architectures. Initially we are building a repetoire of emulators whose target architectures are conventionally structure, register oriented machines. We are interested in this class of machines first because they are in wide use and, second because one of them, the 360 series, is likely to become a primary target machine for the laboratory system. In the following section we describe our experience with two emulators, the 360 and the INTEL 8080.

3.1.1 360 Emulator

The 360 series emulator currently under development in the laboratory is intended to be 'class B'[35,42]. That is, valid 360 programs will produce corresponding results on the laboratory system, while invalid programs may fail in a manner which differs from that of the actual 360 target machine. Our long term objective is to emulate all aspects of the 360 system which directly influence the execution of problem state code and the most important of the features required by supervisor state code. Thus, object code directly from user sources, such as the Fortran compiler, may be processed without intermediate checking or translation.

For the short term the 360 emulator will only be required to handle the basic target instruction set (i.e. no floating point or decimal operations). Of this basic set seventeen instructions represent about 70% of the instructions encountered in actual practice and have been given priority in optimization. The emulator described below is capable of supporting a PL360 submonitor system which will eventually form the basis of the laboratory operating system. Consideration has been given to extending the 360 emulator to include virtual addressing [41].

As a 'class B' emulator not all target machine features are supported. Some of the differences are:

1) No dynamic address translation (360/67)
2) No address exception
3) No protection
4) No boundary restrictions

3.1.1.1 Structure

The 360 emulator is representative of the structure used in other register oriented, third generation machines we have examined. Architecturally, the 360 is well suited to emulation in general because the instruction formats are few and regular, and the opcodes are organized in an orderly, non-overlapping manner. Furthermore, the information reflecting the current state of the machine (basically the PSW) may only be accessed explicitly by the instruction stream, thus enabling the emulator to represent this information in encoded form for efficiency. Since implicit referencing (perhaps via addressing) is not allowed, recurring penalties for checking and reformatting are not necessary.

Figure 3.1 illustrates the basic approach used in structuring the 360 emulator on the EMMY host machine. The basic stage of microexecution are:

1) Handling of external interrupts
2) Operation decode
3) Effective address formation
4) Operation execution and instruction prefetch

Although the EMMY CPU accepts direct interrupts from the external world, it is advantageous in most cases to handle these interrupts using microcode. In the 360 emulator direct interrupts are accepted by the CPU during the operation decode and address formation stages of the basic-emulator loop. Any interrupts occuring during this time period are mapped into a corresponding interrupt of the target machine, but no actual processing of the interrupt will be undertaken until the current target instruction execution completes. These mapped interrupts are examined by the emulator at the first stage of each target instruction loop and are then processed (subject to masking) according to 360 requirements.

Decoding of the target machine instruction proceeds in two stages:

1) 1 out of 4 decode to determine format
2) 1 out of 256 decode to determine operation

The initial format decode is used to determine which of four routines will be used to parse the instructions, calculate addresses and update the target machine program counter. Before entering the selected routine however, the entire op-code field (eight bits) is decoded to determine the specific operation routine which will ultimately be performed. For operation decode a table of 256 entries is indexed directly from the op-code and the semantic information retrieved is used to drive the execution process. This information includes a pointer to the semantic routine itself, bus control information for the laboratory main memory system and various indicators which steer the semantic routine.

Following the decode stage one of four routines is entered to parse the remainder of the target machine instructions. During this process the actual memory system addresses (if required) of the operands are calculated and saved for the execution routine.

Execution of the semantic routines corresponding to the op-code is straight forward and relies upon information assembled in the register file during the previous stages. Exceptional conditions arising from this execution are handled by the same machinery which sets up the external interrupts.

During the execution of the routine a fetch is made of the next target machine instruction. This is done to anticipate the normal (sequential) flow of the I-stream. In the case of a conditional branch both potential target instructions are fetched and the semantic routine selects between them after evaluating the branch conditions. It is advantageous to fetch from the most likely target address first, since a penalty is incurred in awaiting the completion of the second fetch. For operations such as as BXLE and BCTR this means fetching the branch target first and the sequential target second.

In figure 3.2 the assignment of EMMY microstore to the emulation process is shown. Current 360 state information (i.e. the registers and PSW) occupies 27 locations. Decoding routines, including the semantic pointers consumes about 300 words, and finally, 1.5K words are consumed by the semantic routines themselves. 2K control store words are unused at present and will be assigned to I/O functions.

We have chosen not to implement the I/O structure of the 360 directly, that is, by emulation of such instructions as SIO and HIO. Rather, the approach has been to handle I/O through the usual convention of a system supervisor call (SVC) which starts execution of 360 code tailored to the particular I/O task and equipment in the laboratory. The basic 360 instruction set has been augmented with special I/O op-codes to allow the supervisor code to initiate and test the status of external actions. The distributed nature of the laboratory bus allows the main memory system to operate in a manner functionally equivalent to main memory in a 360 target system.

3.1.1.2 Emulator Performance

Perforance estimations of the 360 emulator on EMMY [35] are based on weighted instruction usage measurements. This approach normalizes the speed differences between the emulator and the hard machine when processing an actual instruction stream. Furthermore, under this measure overall emulator performance may be improved by more efficient emulation of frequently used target instructions at the expense of those which are less frequently encountered.

Figure 3.3 tabulates the instruction execution rates for the 360 emulator and the model 360/50 processor. Occurrances of these instructions in actual instruction streams are shown as percentages of the total stream. In all, these seventeen instructions comprise 63.4% of the overall I-stream [3]. For variable length instructions the times have been calculated based on statistics concerning expected length [14]. This, of course, is to the benefit of the emulator performance estimate since instructions such as MVC are able to amortize the overhead of decode and address calculation over the entire execution cycle.

Based on the instruction stream characteristics given in figure 3.3 the emulator has an execution rate of 96.9 KIPS and the 360/50 has a corresponding execution rate of 141KIPS. Roughly speaking then, the emulator has 70% of the computational power of the target machine. With improvements to the architecture, which are discussed in the final section, this gap will be closed so that the emulator has equivalent or slightly greater performance tnan the hard 360/50.

3.1.1.3 Critique of Emulation Code

Heavily used segments of the 360 emulator have been highly optimized using EMMYXL to allow explicit definition of each microinstruction. Thus, this emulator represents a benchmark in efficient microprogramming of the EMMY CPU.

Since EMMY uses an assymetrical, split microinstruction format, coding inefficiencies occur when an available microinstruction 'half' will not serve the required purpose. Static code analysis shows the following:

1) TCF half unused -- 13 percent
2) ACF half unused -- 4 percent

This implies that about nine percent of the microcode contains null operations although this actual represents only a 2 percent time inefficiency, since most null operations consume no execution time.

Approximately 26 percent of the microinstructions make data references to the microstore. Eight percent are direct references, and eighteen percent are indirect references. These references to microstore are due to its use as the primary storage resource in the emulation environment.

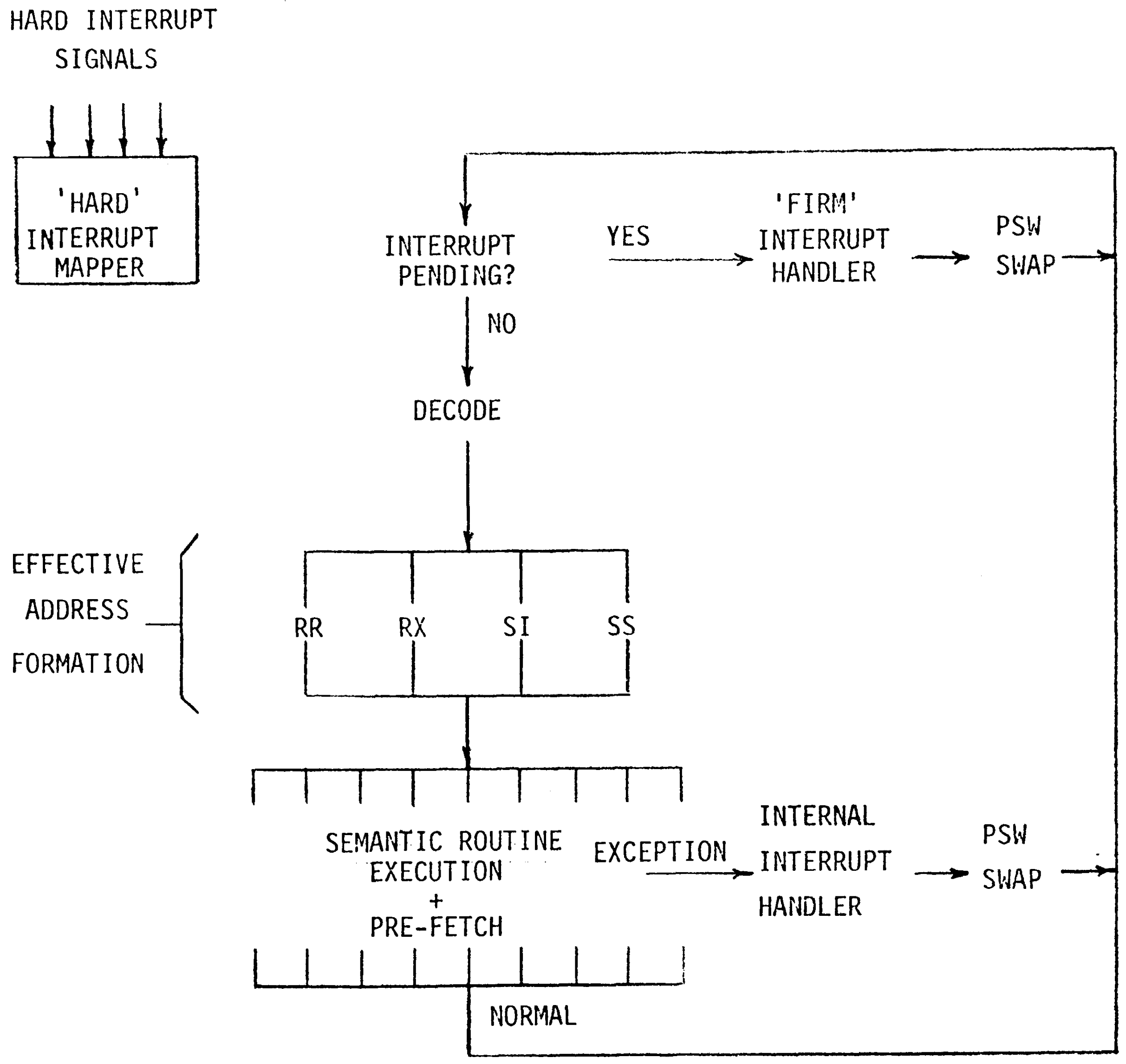
An average microinstruction consumes 12.7 internal clock cycles, which is equivalent to 445 nsec. Approximately 12 percent of the CPU execution time is spent in processing shift/rotate operations.

3.1.2 Intel 8080 Emulator

The results of one project involving the emulation laboratory has been the construction of an emulator for the Intel 8080 microprocessor [36]. This target machine is an eight bit single package processor finding wide application where control rather than compuataion is required. The instruction set of this processor is very orderly and as such provides a low end benchmark for the EMMY processor. Also, it is an interesting execise in scaling down the 32 bit resources of EMMY to the 8 and 16 bit requirements of the 8080.

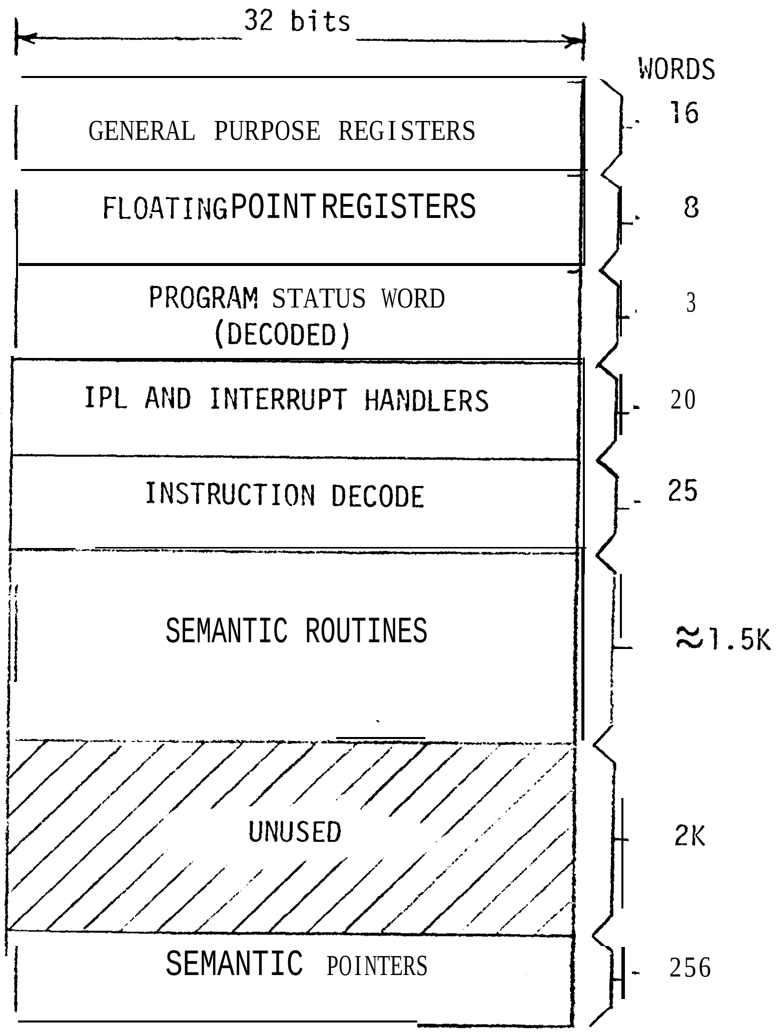
Eight bits specify the opcodes of the 8080 instruction yielding 244 executable combinations consisting of 76 basic instructions and their variations. Since 4K of Microstore is available, the approach taken in this emulator has been to perform a one out of 256 decode. This results in a fetch and decode time of 1.5 usec. Execution of the semantic aspects of the instruction consumes varing amounts of time as illustrated in figure 3.4. EMMY's 650 nsec main memory falls between the two RAM memory systems available for the 8080 system. EMMY has comparable performance in a absolute sense. In the future we will to make a determination of instruction frequency distribution in 8080 code in order to optimize the emulator in the same manner used on the 360 discussed above.

Total control store space requirements of the 8080 emulator are 1.5 words.



STRUCTURE OF 360 EMULATOR

Figure 3.1



MICROSTORE ASSIGNMENT FOR 360 EMULATOR

Figure 3.2

INSTRUCTION	%	360/50 (USEC)	360/EMMY (USEC)
L	16.1	5.5	8.3
BC	8.7	4.2	10.0
BCR	6.9	4.0	7.3
LA	4.8	3.25	9.0
ST	3.6	5.5	8.25
MVC *1*	3.5	30.0	22.4
LR	2.2	3.0	5.25
AR	2.2	3.75	6.5
A	2.0	5.5	10.25
BXLE	2.0	5.5	12.75
TM	1.8	6.25	10.1
LH	1.8	6.25	8.3
CLC *2*	1.7	24.6	27.5
BALR	1.6	4.25	8.5
S	1.6	5.5	10.2
LM *3*	1.5	20.0	24.7
MVI	1.4	5.5	8.0
	63.4%	7.00 USEC AVG.	1.0.3 USEC AVG

1 Average length = 14 characters

2 Average length = 10 characters

3 Average length = 7 characters

COMPARISON OF 360/50 AND EMULATOR

Figure 3.3

INSTRUCTION	8080 (450NSEC)[1]	EMMY (650NSEC)[2]	8080 (850NSEC)[1]
ADD B	2.0	3.0	2.5
INR B	2.5	4.7	3.0
MOV B, C	2.5	2.8	3.0
AN1 D8	3.5	3.2	4.5
PUSH B	5.5	3.6	7.0
CALL ADR	8.5	8.6	10.0

1 RAM CYCLE TIME

2 MAXIMUM MAIN MEMORY CYCLE TIME

COMPARISON OF 8080 AND EMULATOR TIMINGS

Figure 3.4

3.2 Architectural Evaluation

3.2.1 Introduction

The evaluation of specific machine architectures has traditionally been a qualitative rather than quantitative pursuit. While the performance of systems has been studied extensively on a global or 'macro' level, the local or 'micro' level has received little attention. One application area for a universal host machine, such as EMMY, is in the study of machine architectures through the microscopic examination of target machine instruction and data streams. Specifically, the emulation approach allows the experimenter to make detailed and quantitative statements about how a particular architecture dynamically employs its internal resources. In a larger sense, such an examination will also reveal the manner in which the user, either directly or indirectly (via language translators) maps problem resource requirements onto available machine resources.

3.2.2 Historical Perspective

Instruction stream analyses of an elementary nature have been carried out on most major machine architectures. Cpcode frequency data has been derived for several register oriented second and third generation machines: IBM 7094 [11], 360 series [12], RCA 70 [13,14] and the PDP-10 [17].

Lunde [17] in his analysis of the PDP-10 has measured a much broader range of instruction stream statistics. Of particular importance is the analysis of data flow through registers and its relationship to register usage. Rossman and Rao [14] have carried out extensive analysis of the RCA trace library [13] in order to provide statistics related primarily to instruction stream seqencing. Statistics, such as the distribution of distance between setting and testing of condition codes, can be very useful in designing a high performance, pipelined implementation of a particular processor.

The instruction stream studies referred to above have all been carried out on the native machine using trace techniques augmented, in some cases, with special purpose hardware monitors. Tracing, as carried out in the above studies, involves execution of the program to be examined on the native machine. After each instruction or, in some cases, an unconditional sequence of instructions, a record of the relevant machine state is made. This usually includes current address, instruction op-code, effective addresses of operands and next instruction address. Data from the trace is captured on a permanent storage medium where it is then subject to analysis.

Tracing has the advantage that a permanent record is preserved that may be referenced later if a different aspect of the instruction stream needs to be analyzed. Since the same trace may be used many times, anomalies such as I/O access and multiprocessing interruptions may be factored out. Complex analyses, such as the 'register life' statistics of Lunde [17], may be carried out efficiently on the static data resulting from the trace step.

There are, however, limitations to the tracing process. First, the production of the trace tapes can be expensive and may involve complex trapping mechanisms or external hardware monitoring systems. Reductions in processing rates can be in the order of 50 to 1 [13] or higher. Second, production of the trace data must be carried out on the native machine in order to generate the proper I-sequences reflecting data dependencies in the execution process. If a general comparison of several architectures is to be made, availability of the actual hardware is beyond the capability of most facilities. Third, the trace production process must, in order to be efficient, produce output which has less information than the actual execution produces. Specifically, the data stream is usually lost, as is information reflecting internal machine states. Although this information may be reconstructed from the trace it is usually a formidable task.

3.2.3 I-stream Analysis via Emulation

Emulation presents a unique opportunity to gather data related to the instruction execution process. Since an emulator, by definition, tracks all external aspects of a target machine, the host machine has direct access to this information. With all aspects of the target machine state transformation available the experimenter may examine as much or as little of the emulation process as desired.

Such techniques have been employed in the study of a specific architecture (the HP 2100) on a well mapped host machine [15,16]. In one case [15] the experimenter was primarily interested in generating a program trace for debugging purposes. This is accomplished by microcoded routines which were inserted in to the host microprogram following the instruction decode and address calculation phases. Target machine instruction "accounting" [16] follows essentially the same approach to collect instruction class frequency data.

The general process of gathering I-stream data via emulation is depicted in figure 3.5. Emulation of conventional architectures consists of an I-fetch/Decode phase followed by the formation of the effective addresss and finally the actual execution phase. At each stage data may be collected corresponding to the procedural, memory and functional resource usage. With little increase in overhead, microcoded host routines may undertake preprocessing and correlation of this data. At the termination of an experiment information is dumped for further processing and evaluation.

Examples of statistics which may be gathered at each phase of the emulation process include the following:

1) Procedural phase

   a) Machine state and condition code distribution
   b) I-stream diversion resulting from state testing
   c) Explicit diversion (i.e. calls and branches)

2) Storage phase

   a) Resource usage distribution (registers, memory, stacks)
   b) Resource activity (e.g. register life)

3) Functional phase

   a) Resource usage distribution (adder, shifter etc.)
   b) Serial reusage (e.g. average string length)
   c) Usage objective (fixed, floating, indexing)

The classifications above are quite microscopic. At a higher level we are interested in capturing data which would describe the way in which resources in each of these classifications would work together in performing target machine tasks. For example, we would like to examine the connectivity of and the traffic volume between the various machine resources. Recurring patterns of resource usage [17] may also be used to measure the effectiveness of a target machine in a particular problem environment.

3.2.4 Cross Architectural Evaluations

A universal host machine, such as the EMMY, is particularly well suited to the task to architectural evaluation outlined above. Because of its flexible character the EMMY system is able to carry out an efficient emulation and associated I-stream data capture for a wide range of target machines. This in turn allows the experimenter to make cross architectural comparisons of machine characteristics. Such a comparison has been made for three machines, the 7090, 360 and PDP-10 [6,18]. Although data for this comparison was gathered via the trace method, it illustrates the form such an analysis might take. The basis of comparison is the analysis of instruction code distributions.

Opcode frequency data is divided into three classes (procedural (P), memory (M) and functional (F)) and ratios are taken between the various classes as summarized in figure 3.6. In a broad sense the P and M class instructions represent overhead in the computation process in that they are primarily concerned with decision making and the staging of data for the functional resources. Taken together the P and M instructions form a class of non-functional (NF) instructions. Thus, the ratio of NF to F and NF to floating point may be taken as an indication of how efficiently a machine performs its primary task, which is F class processing.

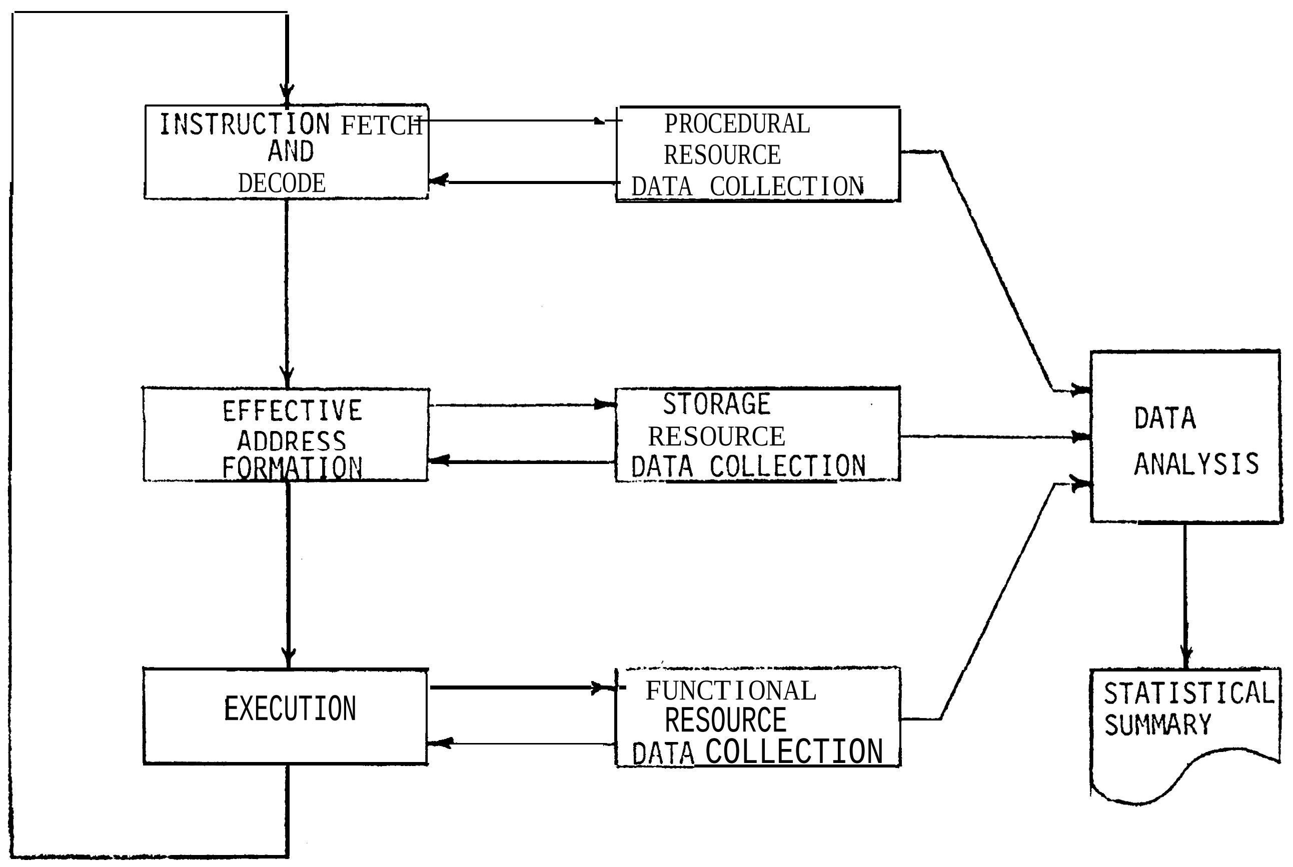
Even from this relatively gross analysis of I-streams important conclusions have be drawn concerning the three architectures:

1) Introduction of a base register addressing mechanism has required extra overhead expenditures when making machine state transistions (observe the 360's high V-ratio)

2) Direct testing of operands is more effective than using condition codes

3) The more complex nature of the program environment (control and data structures) may contribute to the high NF ratio observed in more recent machines.

This analysis is unable to distinguish the fine structure of resource usage. Thus, the M-class statistics include as overhead memory operations, which in tasks such as sorting would be considered functional. Also, specific operations such as string moves and shifting may be used to perform the same external function, but one is arbitrarily considered overhead and the other functional. We hope that a more detailed instruction stream analysis will allow us to resolve these ambiguities and make more definitive statements about architectural performance.


INSTRUCTION FETCH AND DECODE
PROCEDURAL RESOURCE DATA COLLECTION
EFFECTIVE ADDRESS FORMATION
STORAGE RESOURCE DATA COLLECTION
EXECUTION
FUNCTIONAL RESOURCE DATA COLLECTION
DATA ANALYSIS
STATISTICAL SUMMARY

I-STREAM DATA COLLECTION

Figure.3.5

	7090 [6]	360 [6]	DEC 1Ø [18]
F	25.1%	15.3%	27.7
P	20.4%	38.3%	30.5
M	49.2%	45.1%	41.5
M-Ratio (M/F)	1.96	2.9	1.5
P-Patio (P/F)	0.81	2.5	1.1
NF-Ratio (M+P/F)	2.8	5.5	2.6

CROSS ARCHITECTURE COMPARISON FOR GENERAL TECHNICAL CODE

Figure 3.6

3.3 "Soft" Versus "Hard" Architectures

We have already noted EMMY's ability to emulate traditional machine architectures. Given optimized emulator code, EMMY drives the image store of these machines at a significant fraction of the main store bandwidth. Algebraically: if A is the average number of image store accesses reqired per image instruction, N is the number of image instructions emulated per unit time, and B is the maximum number of accesses that can be made to main store per unit time; then A*N is at least 258 of B. Comparing the performance of an emulation to the technology of the host's main store is one way to obtain a reasonable efficiency estimate that is, to some degree, technology independent.

In general, a "soft" host can emulate many different image machines efficiently. In contrast, a "hard" host is designed specifically to interpret only a single "native" architecture (or small set of architectures) with similar efficiency. Within a given technology, of course, a hard host should be more cost effective than a soft host when emulating its native architecture. The generality of a soft host need not severely limit its cost effectiveness, though, as evidenced by EMMY's performance when emulating 360 machine architecture. The significant question is whether enough additional performance can be gained by exploiting the flexibility of a soft host to offset the inherent specific advantage of a hard host.

This duality is perhaps more visible in the context of a complete system in which several source languages are to be evaluated. In hard host systems, the native language must serve as a "target language" for every source language compiler, though only this native language need be emulated directly. In soft host systems, the reverse is true. There is one target language for each source language, but the soft host must be able to interpret all of these target languages efficiently.

System development cost in either instance will be roughly equal. A non-trivial and relatively complex compiler (or set of compilers) must be created for each desired source language in systems with a hard host. Given a soft host, however, only a single comparatively simple compiler (and emulator) is required for each source language. The cost of the extra component per source language (the emulator) should be offset by a reduction in the complexity of the compiler and attendant run-time support. Further, if nothing can be expressed in an image architecture that does not have a direct analogue in the source language, then the impetus for re-coding system (and production) programs in "assembler language" is removed.

3.4 Directly Executed Languages (DELs)

We call an intermediate language that is tailored to a specific source:host combination a DEL. This notion is similar to a number of ideas proposed in other contexts. An UNCOL, or "Universal Compiler Oriented Language", is an intermediate language designed for maximal machine-independence and portability. It need not be "tailored" to any specific source language, however, and efficiency of execution is usually sacrificed to obtain independence of the host machine [19].

A "Machine Oriented Language", designed to allow users to interface with a host at a low level, is really only "half" a DEL. Although well-tuned to a specific host machine, it may not be a good target language for any high level, user-oriented source language. A "High Level Language" architecture -- also called a "Language Oriented Machine" -- is closer to our concept of a DEL [20]. If the High Level Language architecture is realized directly in hardware (as is Bashkow's FORTRAN machine [21]) however, it may be too inflexible for any but its pre-selected "native" source language. Our research is directed toward systems based on flexible, microprogrammable hosts; i.e., unbiased "soft" hosts like EMMY. EULER, an intermediate language for ALGOL:360/30 systems, and the Burroughs "S-languages" for FORTRAN and COBOL:B-1700 systems are noteworthy examples of DELs as the term is used here [22,23].

The study of DELs includes language design, implementation of associated processors, and prediction/analysis of performance. The goal is to develop methods for synthesizing a DEL which minimizes the space and time required to evaluate a "typical" user program, given a full specification of the source language and host machine. Some of the aspects of a system's performance that can be optimized by choosing an "ideal" DEL are:

1) The size of compiler required.

2) The time needed to compile source,programs.

3) The size of a typical DEL surrogate.

4) The time needed to execute a DEL surrogate.

5) The complexity of analyzing a source program by observing the behavior of its surrogate.

The first four factors refer to system resources; the fifth refers to "user" space and time. Although use of a specifically tailored DEL does not improve a source language itself, it usually leads to a comparatively simple run-time system, permitting more transparent program check-out and debug at the source level. This is particularly important in systems where development and maintenance costs often exceed actual production costs.

3.4.1 Toward a Theory of DELs

Our results to date, though limited, do provide motivation for further investigation [1,6,25,34]. An "Existence Theorem" for non-trivial DELs has been developed by analyzing several different methods of evaluating programs. It shows that neither the high level source language for a soft system, nor the primitive micro-orders of its host machine is an "ideal" DEL -- under easily satisfied technological assumptions [l].

With respect to the synthesis of "ideal" DELs, we have established the feasibility of combining register and stack oriented instruction streams within a single DEL [25]. Even naive compilers can generate code relatively free of Load/Store (Push/Pop) overhead instructions by using the set of 'templates', or generalized instruction formats, resulting from this union. This is pragmatically significant, since trace-tape statistics show that the memory-accessing instructions eliminated using this technique account for 308 of "traditional" instruction streams [6,14,17].

We have also experimented with the implementation of Direct Operand Addressing via a Data Reference Table maintained in microstore. Elements of this table are dope vectors in l-l correspondence with the named entities in a source program. By establishing a unique Data Reference Table for each scope activated during execution -- binding only the size and ordering of this table during compilation, a minimal number of bits are required to uniquely identify DEL variables (as operands). In effect, this reduces the "address space" of a DEL surrogate to the "name space" of the original source program. The l-l correspondence between elements of the Data Reference Table and named entities in a source program also eliminates the problem of register allocation, significantly reducing the complexity of the compiler required to translate source code into DEL code [1,25].

There is, however, a time penalty (one microstore cycle per data access) incurred during execution in order to obtain this space reduction. Although the time needed to access a simple variable would be reduced by having the compiler map scalar variables directly into microstore cells, it would also complicate the passing of parameters (by reference), and would certainly introduce some Load/Store overhead instruction units into DEL code. Difficulties with direct mapping of data are compounded in partitioned hosts like EMMY, in which different micro-operators must be used to access main and micro stores. Operand addresses must be checked to determine the module to which they refer, and a conditional branch must be executed that is dependent on the outcome of this check. The actual data transfer to or from main store need not impose a large time penalty if it can be overalpped by other decoding operations [34,35].

Having conducted these preliminary investigations, we were left with a strong conviction that a DEL derived from the above theoretical studies would result in dramatic improvements in system performance -- as compared to traditional machine language DELs. To test this thesis against "real-world" problems, we have designed and implemented a DEL specifically tailored to FORTRAN-II. We expect to use "interpretive probes" imbedded within the emulator for this DEL to gather statistics on salient characteristics of user behavior -- in much the same way that we intend to monitor the use of traditional machine architectures.

3.4.2 The DELtran Design

DELtran is an intermediate text tailored to FORTRAN-II as a source language and EMMY as a host machine. It is designed to optimize space and time during execution subject to the following compilation constraints:

1) At most, two passes over the source code be required -- the first pass to generate a Symbol Table; the second pass to produce the DELtran equivalent for the FORTRAN source.

2) The time needed to generate DELtran code be linear with respect to the number of source program operators.

3) There be a simple correspondence between FORTRAN-II and DELtran operators (including built-in functions).

4) There be a simple correspondence between explicitly named entities (i.e., variables, labels, constants, etc.) in a FORTRAM-II source program and explicitly referenced items in its DELtran surrogate.

Although all of the features in FORTRAN-II are captured by DELtran, not all I/O and Floating Point operators are implemented at this time. The total program and data space for DELtran is limited to the 64K bytes of existing main store, and no more than about 2000 distinct named entities are permitted in a single source program. However, some FORTRAN restrictions have been removed: any expression may be used as an array subscript; procedure names may be passed as arguments; subprocedures can be defined within the body of an enclosing procedure (or MAIN): and no "run-time" subroutine library is required (built-in functions are implemented in micro-code).

3.4.3 General Structure

3.4.3.1 Control Structure

DELtran programs are linear arrays of "instruction units" stored at the upper end of EMMY's main memory. COMMON and LOCAL data is stored at the lower end of main memory, just above a 64 word LIFO evaluation stack. The emulator for DELtran occupies less than 2K words of microstore, including all I/O and Floating Point operators (not all of which are implemented in the current version, which takes less than 800 words of microstore). The upper 2K words of microstore are reserved for Data Reference Tables. Given an asynchronous Block-Access Controller that moves segments between main and micro stores using bus access interleaving (a device anticipated in the design of DELtran, but not yet implemented on the Stanford EMMY), Data Reference Table requirements could be reduced to 256 words. In this case, the entire emulator -- including all I/O and Floating Point operators -- would fit within 2K words of microstore.

Individual instruction units vary from 8 to more than 32 bits in length, and may be packed into one or more words of physical storage. Each instruction unit is composed of one or more sub-fields, called "syllables", each of which must lie entirely within a single storage word. Syllables within a given instruction unit are use-ordered with respect to the emulation process to minimize the size of the emulator's internal state and make effective use of scarce host resources.

A typical flow of control during emulation of an instruction unit might be:

1) Decode a leading "template" syllable, and transfer to a micro-routine that: decodes any explicit references, fetches operand values, calculates the result address (if any), and establishes a standard interface.

2) Decode deferred operator syllable, and transfer to the micro-routine which performs the required function -- this routine decodes references for, and fetches values of, deferred operand syllables.

3) Store the result and check for exceptional conditions, if required, and start emulating the next instruction unit at Step 1.

This general sequence would vary in terms of the number of explicit reference syllales decoded, operand values fetched, etc., depending on the specific codes in the instruction unit being executed. Also, the lead syllable may be an immediate program control operator (these are encoded along with interface specifications due to the observed frequency of Branch/Test instructions in the instruction streams for traditional architectures).

The nature of DELtran is such that almost any sensible DELtran program can be directly expressed in FORTRAN-II (the only exception here is that an implicit evaluation stack can be manipulated in DELtran in ways not permitted in FORTRAN-II, however, this is easily remedied by a few changes in source language syntax). Its basic superstructure appears to be suitable for most high-level, block-structured source languages, although the existing CALL/RETURN and data accessing code would have to be augmented to support recursion and dynamic storage allocation and reclamation.

3.3.4.2 Instruction Unit and Data Structure

DELtran instructions always begin with a leading syllable that is 5 bits wide. This syllable indicates whether an instruction is:

1) a program control operation (like "GOTO" or "CALL")

2) a binary template (for operators like "+" and "*")

3) a unary template (for operators like "FIX" and "EXP")

4) a null template (for operators like "REWIND").

There are 10 program control operators (including "MOVE", which performs simple scalar assignments), 15 binary templates, 5 unary templates, and the null template -- leaving one unused leading syllable code. Program control templates directly invoke a unique semantic routine for each of the 10 possible control operations. Binary and unary templates parse operand reference syllables appearing explicitly within an instruction unit, decode a four bit "deferred operator" syllable, and transfer control to the micro-routine that implements this deferred operator. The null template merely decodes a three bit deferred operator and branches to an appropriate micro-routine. There is a different interpretation of deferred operator codes for binary, unary, and null templates -- so that the same code may stand for different operators in different contexts. Only the zero-code has an invariant meaning -- "fetch the next instruction word".

Explicit operand references are n-bit codes (where n depends on the size of the current scope -- i.e., the number of entries in the active DRT). An internal host register "s" contains the base address of the current Data Reference Table and value of n. This register is saved and restored by the execution semantics for CALL and RETURN.

The dope vector for a variable with reference code "z" is located at "s + z - 1" in microstore. The upper byte of each dope vector contains a code defining the “shape” of the data to which it refers, while the lower three bytes contain the address of the cell in which this data item is stored. “Shape” refers to the elementary size and address units for a DELtran variable. Byte, halfword, and fullword (32 bit) signed integers, and halfword and fullword floating point numbers are permitted. Both one and two dimensional arrays are stored as linear sequences of data elements.

The shape of an array is consistent with the size of its elements: i.e., arrays of halfwords are addressed in halfword increments, arrays of fullwords in fullword increments. All mapping of shaped addresses and alignment of values is performed asynchronously by EMMY’s main memory control unit. The internal representation of integers and floating numbers is such that all permissable shapes can be treated in a uniform manner by execution routines. Conversion between shapes for Load/Store operations is performed entirely within the main memory control unit, based w-I the information contained in the upper byte of the dope vector for any given data element. The "zeroth" word of a two dimensional array contains a left-justified multiplier for the higher order subscript value, which is used by array accessing operators to linearize references to two dimensional arrays.

Not all operand references appear explicitly within the DELtran instruction stream, however. Both binary and unary templates may desiqnate one or both operands (and/or the result location) as “implicit”. Implicit values are kept on a 64 element LIFO stack in main store: the upper 5 bits of environment vector "S" point to the top element on this stack. The length of all intermedate results is assumed to be one word, so shape-typing of stack entries is not required.

The evaluation stack is maintained in main store for the same reason that scalar values are not kept in micro store: the time needed to distinguish between main and micro store addresses exceeds the time required to complete a main store access, and substantial space-time compression results from treating all addresses in a consistent manner. Additionally, we would not be able to take advantage of the mapping capabilities of the main store control unit for data maintained in microstore. In general, we have tried to reference variables in the most direct manner possible: consistent with the requirements of FORTRAN-II (e.g., COMMON and EQUIVALENCE specifications, and the “by-reference;’ calling convention), and the inherent limitations and capabilities of the chosen host.

The interface between binary templates and deferred operators consists of three host registers designated "p", "q", and "r". "p" contains the value of the first operand, "q" the value of the second operand, and "r" the location of the result. Only "p" and "r" have meaning for unary templates, and the interface for the null template is empty (i.e., no requirements on "p", "q", or "r").

Deferred operators are categorized as being "binary" (two operands, one result) or "unary" (one operand, one result). The algebraic type (integer or floating point) of an operand is not checked dynamically due to the time penalty for doing so, and because FORTRAN is such a strongly typed language in its own right. Hence, there are seperate operators for integer and floating arithmetic functions. However, the main memory control unit for EMMY manipulates values so that the same semantic routines (and hence operator codes) can be used for byte, halfword, or fullword operations.

3.4.4 Preliminary Results

The prototype EMMY, operating at an internal clock rate of 50 ns., executes DELtran code at 60-80 thousand instruction units per second. For a FORTRAN version of the 8-Queens problem [40], this is equivalent to 50-65 thousand lines of source code per second. Static comparisons with 360 code (as generated by the FORTRAN-IV compiler with OPT=02, excluding the standard epilogue/prologue and address constant space) indicate program space compression factors on the order of 4-10. Higher static compression factors are possible for short subroutines with several arguments, or which are rich in program control or multi-dimensional array operators. Examination of code generated for arithmetic expressions appears to justify only a compression factor of 2-5. Although in some cases the system 360 optimizing compiler generates extra in-line code in order to reduce execution time, further study is required to account for the high compression factors observed thus far. An average of 3.5-4.5 360 instructions (median = 4) are required to duplicate the function of a single DELtran instruction unit.

In comparison to Bashkow's code, DELtran programs occupy an average of 4-6 times less space. There are some significant differences between the source language specifications assumed in the designs of these two DELs, however. No procedure CALL-RETURN operators are implemented in the Bashkow machine, and not all of the built-in functions permitted in DELtran are accepted in his machine. Further, Bashkow's machine operates only on 16 bit entities (all floating point internally), while DELtran captures a far wider range of numeric types. Hence, our comparison is limited to the examples presented in [21].

We have also compared DELtran to a stack machine for FORTRAN (based on a 16 bit uniform syllable length) designed by McClure in 1968 [24]. Space compression factors of 2.5-10 occur for sample code fragments, the average being 5.5 (median 5). This DEL implements the full FORTRAN language and is intended for production applications. However, its design is highly constrained by the architecture of the 16 bit minicomputer selected as a host; part of the space compression observed here is probably due to the greater flexibility of the host for DLLtran.

Clearly, we cannot properly compare execution times for the above DELs with DELtran, since the host technology is an overriding consideration in this question. However, we can compare execution times for DELtran and 360 machine code DELs when both are being emulated on EMMY: register-to-register 360 instructions require about 20% less time than the equivalent DELtran instruction units; reqister-to-memory instructions require almost the same time: program control instructions take 20 to-500% longer; and references to built-in functions may take several thousand per cent longer in 360 code than in DELtran. Preliminary data indicate a speed-up factor of 4-12 for simple programs -- perhaps 30 (or more) to 1 for programs invoking more complex functions such as logarithms, cxponentiation, etc. Again, additional experimentation is required to verify this result. A production version of EMMY, incorporating some of the implementation enhancements discussed in the next section, would execute DELtran at 150 to 200 thousand instruction units per second. The percentage improvement in DELtran execution that would be realized on an enhanced machine is higher than that for 360 code because greater use can be made of the increased parallelism and overlapped operation.

A cross-compiler for translating FORTRAN into DELtran is under development. It is expected that this compiler will take 2G to 100 times less time than the system 360 optimizing compiler. The next block of experiments will involve about 50C lines of FORTRAN code, selected from a library of standard benchmarks, and about 1000 lines of FORTRAN code extracted from the data base used by Knuth in deriving his statistics [26].

3.5 Laboratory Enhancements

Use of the laboratory hardware has indicated several areas where the EMMY system might be enhanced technologicaly and structurally [37]. From a technological point of view, there are two possibilities. First, it appears feasible to reduce the current 35 nsec internal cycle time to 30 nsec by using a printed circuit interconnection rather than wire wrap. Second a static, bipolar control store may be substituted for the current psuedo-static implementation. Although both control stores have approximately the same access time (70 nsec) the current control store has a 130 nsec recovery time. In situations where control store is accessed explicitly for data this recovery period delays the start of the following microinstruction fetch. Taken together, these two changes would increase the effective microinstruction execution rate by 40 percent.

Structurally, we are examining several schemes to improve the field handling and decoding capabilities of the machine. Currently the shift/rotate unit executes a single shift step every machine cycle (35 nsec). Doubling the shift rate would increase performance by about 10 percent. If a barrel shift capability of one to eight bits was included performance enhancement in critical decode operations would be enhanced by about 40 percent.

We are continuing to study the ways in which host machines might be structured with the objective of attaining real time or hyper-real time emulation of third generation machines [38]. Execution rates of one to five MIPS seem possible in a highly overlapped system.

4.0 Conclusions

A universal host machine and its laboratory environment has been described. The experiments conducted so far indicate that a universal host machine, particularly the Emmy, can be a useful realization of both hard and soft target machines. Experience gained with both hardware and software tools shows that direct user microprogramming can be used effectively in the experimental environment.

5.0 Acknowledgements

Realization of any project with the scope of the emulation laboratory requires the specialized talents of many people. The authors are particularly indebted to Dr. Robert McClure (Palyn Associates, San Jose, California) for his contributions to the system design and implementation. The authors also wish to acknowledge the help of their other colleagues at Palyn -- Mike Fung, Bob Domineco, Dan Davies and Stan Levy -- for the detailed hardware implementation of the system. Walter Wallach (Stanford Digital Systems Laboratory) implemented the System 360 emulator. EMMYPL was implemented by John Polstra and EMMYXL by Tom Hedges, both of the Stanford Digital Systems Laboratory.

6.0 REFERENCES

[01]	L. Hoevel	'Ideal' Directly Executed Languages: An Analytic Argument for Emulation IEEE Transactions on Computers; August 1974.
[02]	_____	QM-1 Nanodata Corporation Buffalo, New York.
[03]	_____	B-1700 Systems Reference Manual Burroughs Corporation Detroit, Michigan; 1972.
[04]	L. Yelowitz, et al	A Simulator for a Microprogrammed Computer, Its Microassembler, and an Emulation. Hopkins Computer Research Report No. 1 Johns Hopkins University; Baltimore, Marylar April 1971.
[05]	J. Davison	The JHU Universal Host Machine II Part I: The Machine, Hopkins Computer Research Report No.31 Johns Hopkins University; Baltimore, Marylar March 1974.
[06]	M. Flynn	Trends and Problems in Computer Organization: Proceedings of IFIPS Congress, North Holland Publishing: 1974.
[07]	R. Cook and M. Flynn	System Design of a Dynamic Microprocessor, IEEE Transactions On Computers; March 1970.
[08]	R. McClure	Parallelism in Microprogrammed Controls, The International Advanced Summer Institute on Microprogramming; 1972.
[09]	_____	2100 Computer Microprogramming Guide, Hewlett-Packard Company; 1972.
[10]	N. Wirth	PL360: A Programming Language for the 360 Computers, J.ACM; January 1968.
[11]	J. Gibson	The Gibson Mix, TR.00.2043, IBM Systems Development Divisior Poughkeepsie, New York.

[12] W. Connors, et al
S/360 Instruction Usage Distribution,
TR.00.2025, IBM Systems Development Division,
Poughkeepsie, New York, May 1970.

[13] R. Winder
A Data Basse for Computer Performance Evaluatior
Computer (IEEE Publication), March 1973.

[14] G. Rossman and B. Rao
System 360 Program Statistics,
Palyn Associates Inc.
San Jose, California, January 1974.

[15] D. Barnes and L. Wear
Instruction Tracing via Microprogramming,
Preprints - MICRO 7, September 1974.

[16] S. Ma and L. Wear
Dynamic Instruction, Set Evaluation
Preprints - MICRO 7, September 1974.

[17] A. Lunde
Evaluation of Instruction Set Processor
Architecture by Program Tracing,
Carnegie-Mellon University,
Computer Science Department, 1974.

[18] A. Lunde
More Data on the O/W Ratios,
Computer Architecture News,
Vol 4, No.1, 1975.

[19] T. Steel
UNCOL: the Myth and the Fact,
Annual Review in Automatic Programming,
1961, 325 - 344

[20] T. Hu
High Level Machine Architecture,

[21] T. Bashkow
Systems Design of a FORTRAN Machine,
IEEE Transactions on Computers,
August 1967; pp 485-499.

[22] H. Weber
A Microprogrammed Implementation of
EULER, Communication-of the ACM,
September 1967; pp 549-558,

[23] W. Wilner
Burroughs B1700 Memory Utilization,
Proceedings of the FJCC, 1972,
pp 499-586.

[24] R. McClure
CUC Basis Fortran Description,
1968, (Private communication).

[25] L. Hoevel
Languages for Direct Execution,
Sig-Micro 7, September 1974.

[26] D. Knuth — An Empirical Study of FORTRAN Programs, Software Practice and Experience, Vol 1, 1971, pp 105-133.

[27] N. Wirth — Program Development by Stepwise Refinement, Communications of the ACM, April 1.971

```
*****************************************
*  The reports below are avalable from: *
*    Digital Systems Laboratory         *
*    Stanford University                *
*    Stanford, CA 94305                 *
*****************************************
```

[28] C. Neuhauser — An Emulation Oriented, Dynamic Microprogrammable Processor (Version 3), Technical Note No. 65, October 1975.

[29] C. Neuhauser — EMMY System Peripherals -- Principles of Operation, Technical Note No. 77, December 1975.

[30] C. Neuhauser — Functional Description of the EMMY Main Mem,ory System Technical Note No. 57, August 1975.

[31] J.Polstra and A. Proskurowski — EMMYPL: A Description and Implementation, Technical Report No. 99, November 1975.

[32] T. Hedges — EMMY/360 Cross Assembler, Technical Note No. 74, December 1975.

[33] R. Lee — EMMYOS: Description.and Implementation, (To be issued).

[34] W.Wallach and L. Hoevel — A Tale of Three Emulators, Technical Report No. 98, November 1975.

[35] W. Wallach — System/360 Emulator Performance Estimate, Technical Note No. 66, November 1975.

[36] F. Wingate — Intel 8080 Emulator Report, (Class report), December 1975.

[37] W. Wallach and L. Hoevel — Proposed Enhancements to EMMY, Technical Note No. 67, November 1975.

[38] W. Wallach — High Performance Emulation,
Technical Note No. 102, November 1975.

[39] W. Wallach — EMMY/UNIBUS INTERFACE, Preliminary
Specification,
Technical Note No. 88, June 1976.

[40] J. Polstra — EMMYPL User's Guide,
Technical Note No. 86, April 1976.

[41] W. Wallach — EMMYXL User's Guide,
Technical Note No. 84, March 1976.

[44] W. Wallach — Virtual Addressing for the EMMY/360,
Technical Note No. 89, June 1976.

[43] W. Wallach — EMMY/360 Functional Characteristics,
Technical Report No. 114, June 1976.

NOTES

1) UNIBUS, PDP-11 and PDP-10 are the registered trademarks of the Digital Equipment Corporation.

2) DP2200 and Datapoint are the registered trademarks of the Datapoint Corporation.

3) MECL 10K is the registered trademark of Motorola Inc.