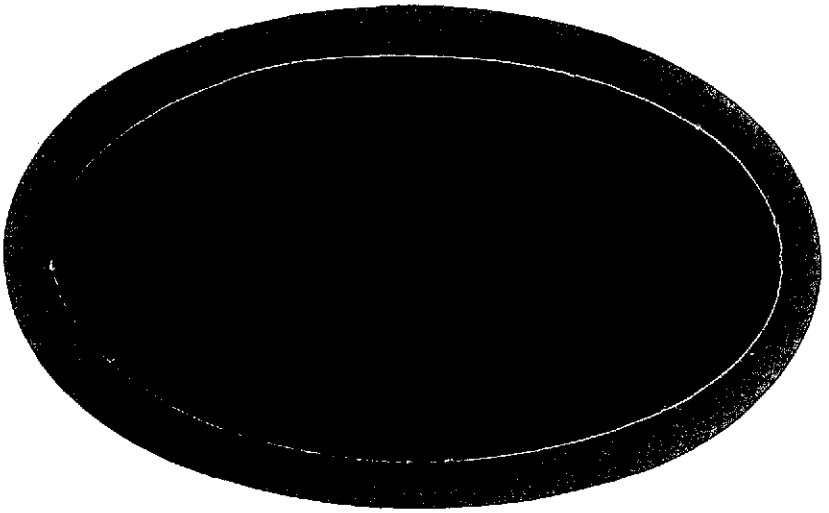
MSC-05107

NASA CR-
140222



INTERMETRICS

MSC-05107

Final Report

Advancing HAL to an Operational Status

July 1974

Prepared under Contract NAS 9-12291

Intermetrics, Inc.
701 Concord Avenue
Cambridge, Massachusetts 02138

Intermetrics Final Report #88-74

Foreword

This document is the Final Report of the delivery of HAL compilers and the engineering study reports of the design of HALM, a computer architecture to directly execute HAL statements. This program was sponsered by by the NASA Johnson Spacecraft Center, Houston, Texas, under Contract NAS 9-12291. It was performed by Intermetrics, Incorporated, Cambridge, Massachusetts over the period October 1971 to July 1974. The program was under the direction of Mr. Daniel J. Lickly and Dr. Fred H. Martin. Mr. Woodrow Vandever was the principal contributor to the HALM effort documented in Chapter 4 of this report. The NASA Technical Monitors for the Johnson Spacecraft Center were Mr. Jack Garman and Mr. Richard Carl.

Publication of this report does not constitute approval by NASA of the findings and conclusions contained therein.

Table of Contents

1. INTRODUCTION AND CONTRACT SUMMARY

The development of the HAL language and the compiler implementation of a mathematical subset of the language had been completed under NAS 9-10542. The on-site support, training, and maintenance of this compiler were completed under NAS 9-11944. The objective of this contract was to broaden the implementation of HAL to include the implementation of all features of the language specification thus permitting MSC to conduct an evaluation of the language for NASA manned space usage. The contract commenced on 31 October 1971 with these two tasks: The implementation on the 360 of all HAL language specification features and the implementation of a HAL compiler for an airborne computer. In this case, the IBM 4πEP computer was selected. This machine was scheduled to be an integral part of an MSC Shuttle avionics breadboard. Early in the contract period, it was recognized that this avionics development system was being redirected and it was pointless to continue with the 4πEP as an object machine for a HAL compiler. Fortunately, few resources had been expended in this direction. A stop work order was issued, followed by a change order directing Intermetrics to establish a HAL facility on the Univac 1108. This contract change order was effectively initiated in April 1972. In addition to the 1108 compiler effort, a task was also undertaken to conduct a study of the problems associated with implementing a HAL compiler on an air borne computer.

In July 1972, the Space Shuttle Orbiter contract was awarded to Rockwell, International (then North American - Rockwell). In October 1972, at the first meeting of the software control board, it was decided to use HAL as the programming language for the Space Shuttle computer. Intermetrics came under contract for the development of those compilers. It was then redundant to conduct the engineering study under this contract as the implementation would solve these specific problems. This effort was put aside until August 1973 when, after considering a number of alternatives, it was decided by NASA/JSC to conduct a design of the implementation of a HAL machine.

The contract was amended in January, 1973 to add an additional task to conduct a study of the possible implementation routes to construct a GOAL to HAL translator. This effort was conducted for the Kennedy Space Center using this contract or an existing vehicle and was intimately tied to the basic contract objective. The results of this work have been previously reported and are contained in the following document:

1. The GOAL-TO HAL/S Translator Specification, Contract NAS 10-8385, December 15, 1973.

This final report then addresses three basic items. Chapter 2 is a summary of activities associated with the HAL compiler for the IBM 360/75 computer. Chapter 3 is a summary of activities of the moving of the HAL/360 compiler to the UNIVAC 1108. Chpater 4 is the results of the engineering study and design of a HAL machine. Chapter 5 are the conclusions and recommendations for further work.

2. THE HAL/360 COMPILER

2.1 HAL/360 Compiler Releases

The structure of the HAL/360 compiler had been developed under a previous NASA contract. The compiler is a two pass compiler. Pass 1 performs syntactic and semantic interpretation of HAL statements. The output is an intermediate language, HALMAT. This portion of the compiler is machine independent and is written in XPL. XPL is a higher order language (a subset of PL/1) and has been designed for writing compilers. Pass 2 of the compiler is the code generator and becomes machine specific. In this compiler, the code generator translated HALMAT into Fortran. There were some portions of HAL system, the run time library, that were more ammenable to direct 360 BAL statements and were implemented in that manner. This general structure of the compiler was released for usage on 6/8/71 and implemented a mathematical subset of HAL plus certain other features. Further releases of the compiler were accomplished during the summer of 1971. These added new language features and modified the compiler to operate on the IBM 360/75 complex at MSC which utilized RTOS as the operating system.

The compiler development was managed using the development plan concept. The plan was updated and reviewed with NASA/MSC technical personnel on approximately a two month schedule. The final release schedule for compilers is shown in Figure 2-1.

HAL COMPILER RELEASE SCHEDULE

Release Number	Target Date	Predicted Date	Actual Date	Comments
360-1	4/5/71	4/5/71	4/5/71	Feasability Version
360-2	6/8/71	6/8/71	6/8/71	Most of HAL_M plus other features
360-3	7/30/71	7/30/71	7/30/71	RTOS Modifications
360-4	9/15/71	9/15/71	9/15/71	See HAL User Memo (10/71) (Appendix A)
360-5	1/10/72	1/10/72	1/10/72	See HAL User Memo (03/72)
360-6	3/15/72	4/1/72	4/14/72	Most of Real-time, complete output writer, diagnostics HAL User Memo(15/72)
360-7	5/15/72	6/8/72	6/13/72	User-aids, error handling HAL User Memo (19/72)
360-8	7/15/72	8/8/72	9/15/74	Structures, update blocks, access rights, data sharing, link to FORTRAN, Optimization, clean ups
360-9	10/1/72	10/1/72		
360-10	11/1/72	11/1/72		
360-8A			2/21/73	Final HAL 360 Release
			7/25/73	Compiler modified to correct reported errors and discrepancies

Figure 2-1

2.2 Compiler and HAL System Features

2.2.1 Real Time Features

The real time language features of HAL were released in version 6 of the compiler with some of the final clean up of these features being completed by release 360-8. The real time features of HAL provided an active means of controlling the computing system for purposes of manned space software development. These features were, for the most part, a departure from the general capabilities of most higher order languages. In particular, the Fortran intermediate approach to HAL implementation did not provide means to deal with these features. For the most part, they were implemented by linking to run time routines written in 360 basic assembly language, BAL.

Real time implies a clock, either a real or pseudo-clock. In this implementation, the actual 360 clock was used for timing. Interfaces to this clock were implemented, and access to time dependent HAL statements were thus 360 clock time dependent. A dynamic storage capability was implemented permitting multiple scheduling of the same program or task. Included in the real time statements implemented were:

SCHEDULE: A capability to activate a program or task on the basis of an event or time.

TERMINATE: The ability to terminate a program or task.

UPDATE PRIORITY: This feature permitted the change in priority of a program or task in real time. Real time dependency permitted an ability to schedule in advance a program or task dependent upon another program or task. Real time task ID was implemented. This is an ability to control multiply scheduled versions of the same program or task.

A second category are the real time services. These include:

SIGNAL: A statement that causes an event. This event could be used to wake up a task or program and be activated on the basis of the emitted signal.

WAIT: A program could be scheduled to WAIT in real time for a signal or a specified time.

Data sharing is a feature that was included with the HAL system. This is fundamentally a real time feature. The capability was included to permit data to be shared with reading and writing. A series of locks are employed such that during the time that data is being modified or accessed by a program, the operation is permitted to run through completion without being interrupted by another program which desires to modify or access the same data. The critical operations are confied by the compiler to an UPDATE block, which provides both high visibility on the program listing and the protected environment during execution.

2.2.2 Advanced HAL Language Features

The following advanced development features were included in the HAL compiler implementation:

ARRAYS: An ability to handle data structures of a very complete nature was included in the compiler implementation. These could be multi-dimensional arrays, or hierarchical tree structures of data. Structures were handled in a very general sense.

BITS AND CHARACTERS: The ability to manipulate bits and to handle characters were included in the compiler implementation. An ability to control the precision of data was part of the compiler implementation. This is called precision modifier. It gave the ability to ask for either single, double, or mixed precision of arithmetic.

The file I/O statement was included. This gave, through the language, a direct access to randomly stored file data. A feature was added to the compiler to permit Compool initialization, that is, a means to start the Compool values out at desired values.

2.2.3 HAL System Features

Of primary importance to the HAL system was the output writer. The output writer is a program which operated in Pass 1 of the compiler to put each compiled HAL program into a standardized format. The listings were annotated and indented to provide paragraphing for easy identification of programs, tasks, and procedures. It permitted quick identification of statements, such as IF THEN ELSE, and DO groups. The multiple line format was included to subscript and superscript variables, for example, vectors were marked by a bar superscripted over the variable and matrices with an asterisk. Brackets and braces were used to identify arrays and structures.

At the end of the program listing generated by the output writer, a program layout was formatted. It gave the program and all the procedures and tasks within the program, and the procedures within tasks.

A symbol table was included and a cross reference for all of the HAL variables within that program or compilation. Attributes of the HAL variables are listed, such as statement numbers for declaration, reference and use. The output writer was a significant advance as an aid to management for flight software development.

A complete system of traces and dumps was included with the compiler. There was an ability to dump at termination, and this dump was done with HAL variables. Individual HAL variables could be dumped by name at any user-specified statements. An ability to trace by HAL statements was provided. That is, an operation of the HAL statements could be traced statement-by-statement in a dynamic sense, as the program ran on the 360. One way link to Fortran was provided in the compiler. A HAL program could call a Fortran program. This program could be linked in and run with the HAL program.

A complete system of error determination and recovery was included. These fell into two categories: compile time errors and run time errors. In the compile time error category, the compiler listed where errors were found, and categorized them. A serious attempt was made to compile the program in the presence of errors. There is a limitation as to the ability to continue compilation based upon the severity of errors.

The second class of errors had to do with run time errors. Here, two capabilities are included. One capability is to perform an operation upon the event of an error. For example, ON ERROR X, performs this operation. A second class of run time errors are those associated with mathematical singularities. These are signalled through the run time system in the event of a mathematical error. For example, DIVIDE BY 0.

Compiler directives are a feature that were included within the system. As an example, the INCLUDE directive allowed a programmer to include other HAL programs with a simple inclusion statement. That is, that these are non-language features that aid in the building of HAL programs.

Access Rights: Access rights are management tools which can be employed to limit the access of programs, tasks, and procedures to the availability of data. For example, only certain programs could be permitted access to read the state vector of the vehicle, or only certain programs could be permitted an ability to write the state vector of the vehicle.

2.2.4 Documentation

A HAL 360 User's Manual was issued in November 1972. This document constitutes the User Manual for the 360 implementation of the HAL language and the compiler. The User's Manual, along with the Language Specification, contains the fundamental information needed for a programmer to write and run a HAL program on the 360 computer. The manual covered the following subjects:

Running a HAL Program: The communication required with the OS 360 in the job control language.

Compiler Outputs: The outputs of subsequent steps of the compiler were covered in detail including the compilation listings. This was the output writer that was described previously.

The User's Manual contained a complete description of the debugging aids both for real time and for non real time programs. This included compilation errors, execution errors, execution dumps and traces. In the real time category, it included compilation errors and execution errors. The manual also described HAL characteristics specific to the 360. Such things as the character set, the internal table capacities, the data type size limitations, Fortran call restrictions, program organization limits, input/output statements, program naming conventions, the include compiler directive, and compile time compatibility checking.

The execution time characteristics contained input/output, formatting of output, and execution time checks. The unimplemented features of HAL and the language restrictions was also contained in the document.

2.2.5 Resident Support, Maintenance, and Training

Mr. Carl Helmers was in residence at NASA/MSC from November 1971 through September 1972. His principle function was to aid programmers in the use of HAL and install compiler releases on the IBM 360. He did, in addition, a number of other tasks. These included: providing complete listings of the error messages for the HAL 360 compiler, and aiding in the translation of XPL to the Univac 1108. Mr. Carl Helmers, along with Dr. Fred Martin, conducted HAL training courses. These courses were given at NASA/MSC and at NASA/KSC. In the area of maintenance and training, an important function conducted was the communications with the C.S. Draper Laboratory, with whom NASA had contracted to perform an evaluation of the HAL language for manned space programming. This group of people performed a very complete evaluation of the language and compiler characteristics and its use for manned space programming. There was much communication between the two organizations to provide support for the use of the compiler and for feedback of desired language features into the HAL system.

3. THE HAL 1108 COMPILER

3.1 Method of Implementation

Under contract NAS 9-12291, Intermetrics was to provide two HAL compilers for the UNIVAC-1108 at MSC: one essentially duplicating the capabilities developed for the IBM/360 (RTCC), and one providing code generation and linking to the G&CD FORTRAN functional simulator operational on the 1108 (SSFS).

HAL/360 as it existed, compiled source HAL language and emitted FORTRAN. This approach had utility at MSC in that linking HAL to already existing FORTRAN programs was straightforward, and HAL/1108 would exhibit this feature. The HAL compiler itself is a large (~15,000 lines) program written in XPL, a derivative of PL/1. It is compiled using XCOM on the 360/75.

1. 1108 Implementation

In transferring HAL from the 360 to the 1108, three technical approaches were considered:

a) Write the compiler in HAL, that is, translate the XPL program into HAL. XPL and HAL have many similar features and the translation can be done, to a great extent (95%) automatically. The objective is to obtain a large HAL program, compile this program on the current HAL/360 compiler, obtain FORTRAN, adjust this 360/FORTRAN to 1108/FORTRAN, and transfer the compiler. The final 1108 compiler would then be in FORTRAN.

Intermetrics fully investigated this approach, wrote sample programs, examined emitted FORTRAN, and concluded that "HAL-in HAL" is not feasible. The essential reasons were that: 1) FORTRAN code generation is too general for an efficient compiler implementa-

tion, 2) resulting code would be very bulky, 3) emitted FORTRAN is unreadable, 4) numerous "handcrafted" changes would be necessary to adjust 360/FORTRAN to 1108/FORTRAN.

b) Write the compiler in FORTRAN, that is translate the XPL program into FORTRAN. This has the same effect as a) above, that of producing a compiler written in FORTRAN which can then be transferred to the 1108. The advantage here is that the translation is direct and not through HAL. As a result, an efficient FORTRAN version could be generated which would be modular (i.e. a series of small subroutines), and readable in that the names of variables, etc. would have some relation to names in the original XPL.

Intermetrics has investigated this approach and although feasible, it was not recommended for two principal reasons:

i) It required a large (essentially manual) translation job from XPL to FORTRAN. These languages are not very similar and we would expect the process to be error-prone.

ii) FORTRAN does not offer language features (control, naming conventions, block structure, data types) which enhance efficient and reliable compiler-writing.

c) Write the compiler in XPL, that is, utilize most of the current XPL source code but provide an XPL-to-1108 code generator. The result here was to augment the current XCOM, which has an XPL-to-360 code generator, with a new code generator. In addition, modularize the XPL source code by making its subroutines independent for convenient use by 1108 programmers.

Intermetrics fully investigated this approach and concluded that of all the alternatives this was clearly the best. XPL was a well known

quantity to Intermetrics, and it is particularly well-suited to compiler-writing. (This is the reason it was selected in the first place for HAL). HAL/1108 would then be a large XPL source program, similar in most ways to the HAL/360, presenting no structure, or readability problems.

Intermetrics ascertained that the technical risk of producing a new 1108 code generator was no more than that of effecting a massive translation into FORTRAN, while the benefits were much greater.

2. Trade-off Issues Between XPL and FORTRAN

a) Technical Risks

Fortran is straightforward, but error-prone because of large translation and would require a higher percentage of assembly language subroutines because of data-type and manipulation deficiencies.

XPL would require a new 1108 code generator but Intermetrics' intimacy with XPL made this task accomplishable.

Intermetrics was confident it could deliver the HAL/1108 compilers, using either approach, within the cost and schedule constraints.

b) Language Features

FORTRAN is not as well-suited to compiler-writing as XPL. FORTRAN exhibits severe name restrictions, is not block-oriented and has poor control structure.

As an example, consider the illustration selected from the HAL/360 compiler, and shown in Figure 3-1. Because FORTRAN only permits

6 letters for an identifier the expressive name HOW_TO_INIT_ARGS in XPL becomes the unintelligible HOWTOI in FORTRAN. Also, VAR_LENGTH becomes VARLEN and VEC_TYPE becomes VECTYP. Note that XPL allows the useful IF--THEN--ELSE while FORTRAN requires multiple GO TO's and the objects of the GO TO's must be numeric; thus GO TO 4, GO TO 5, GO TO 7, etc. In addition, the logical AND must be a function in FORTRAN rather than the operator &, and lastly the convenient hexadecimal constant FF must be expressed as the integer 255.

These few observations portended numerous errors and a variety of translation difficulties using FORTRAN.

c) Maintenance and Configuration Control

By having both HAL/360 and HAL/1108 in a single source language (XPL), maintenance will be less costly and configuration control easier. Maintenance personnel (whether NASA or contractor) need not master two quite different programs and changes and modifications can be effected in a straightforward manner. A separate FORTRAN version for the 1108 would encourage separation of the two compilers and permit independent modification and compilation. Once this drift developed it would be virtually impossible to keep track of or reconcile the differences, especially when the source code program design were different.

d) "Portability"

Although a compiler written in FORTRAN is theoretically portable to other machines because of the universality of FORTRAN compilers, in actuality the specific differences among FORTRAN's can be considerable. HAL/1108 would require advanced FORTRAN V features which are non-standard due to word-length and byte definitions across machines.

3. Summary Recommendation

In view of the foregoing discussion and based upon the analyses conducted by Intermetrics, it was recommended, and NASA/MSC concurred, to pursue implementation of a HAL/1108 compiler by writing an XPL-to-1108 code generator and delivering the HAL/1108 compiler to MSC in XPL source language. Problems arising during implementation were handled by introducing a limited amount of 1108 assembly language and/or other expediencies where required.

Figure 3-1

In XPL

```
HOW_TO_INIT_ARGS :
   PROCEDURE(NA,SYT);
      DECLARE(NA,SYT) FIXED;
      DECLARE (NU,NE,TEMP) FIXED;

      IF NA <= 1 THEN  /*  IF 1 (OR ERROR) ARGS THEN JUST RETURN  */
         RETURN 1;  /* 1 INDICATES 1 ARG  */

      IF SYT_TYPE(SYT) = VEC_TYPE THEN  /*  PICK UP VECTOR DIMENSION  */
         NU = VAR_LENGTH(SYT);
      ELSE DO;
         IF SYT_TYPE(SYT) = MAT_TYPE THEN DO;  /*  GET THE M*N DIMENSION
            NU = VAR_LENGTH(SYT) & "00FF";
            TEMP = SHR(VAR_LENGTH(SYT),8) & "00FF";
            IF (NU="FF")|(TEMP="FF") THEN  /*  CAREFULE OF THE FF CASES  *
               NU = -1;
            ELSE
               NU = NU * TEMP;
            END;
         ELSE
            NU = 1;  /*  THIS IS THE BIT, CHAR, INTEGER, OR SCALAR CASE  *
         END;
```

In FORTRAN

```
      INTEGER FUNCTION HOWTOI(NA, SYT)
      IMPLICIT INTEGER(A-Z)
      COMMON SYTYPE(100), SYARRY(100), EXARRY(50), SYCLAS(100)
      COMMON SYTPTR(100), VARLEN(100)
      COMMON VECTYP, MATTYP, STRUCC
      IF (NA .GT. 1) GO TO 1
C     IF 1 (OR ERROR) ARGS THEN JUST RETURN
C     HOWTOI = 1  INDICATES 1 ARG
      RETURN
1     IF (SYTYPE(SYT) .NE. VECTYP) GO TO 2
C     PICK UP VECTOR DIMENSION
      NU = VARLEN(SYT)
      GO TO 3
2     IF (SYTYPE(SYT) .NE. MATTYP) GO TO 4
C     GET THE M*N DIMENSION
      NU = AND(VARLEN(SYT), 255)
      TEMP = AND(SHR(VARLEN(SYT), 8), 255)
C     CAREFUL OF THE FF CASES
      IF ((NU .NE. 255) .AND. (TEMP .NE. 255)) GO TO 5
      NU = -1
      GO TO 6
5     NU = NU * TEMP
6     GO TO 7
C     THIS IS THE BIT, CHAR, INTEGER, OR SCALAR CASE
4     NU = 1
7     CONTINUE
```

3.2 Implementation Guidelines for 1108 XPL

A set of general guidelines was established to transfer the 360 XPL programs that comprise HALPASS1 and HALPASS2. This contained the design decisions and implementation facts necessary to understand and implement the 1108 XPL.

1. BOOTSTRAP MEDIUM

XPL-1108 would produce assembly language subroutines to be collected together and executed. The assembler was to fix-up forward branches and the like as well as to supply relocation information. It also provided external linkages where appropriate.

2. SUBMONITOR

The submonitor was implemented via a set of library subroutines.

3. REGISTER ALLOCATION

Although a form of a general register machine, the register allocation policy for the 1108 was quite different from the 360. Some of the differences are:

a) No base registers are needed since the 1108 permits direct addressing of the whole computer. (This removes the need of R4 through R11, R13, R14, and R15 of the 360 which were all bases of some sort).

b) 16 accumulators (A regs) which need only be used for accumulators. (R0 through R3 on 360).

c) 15 index registers (4 overlap and hence are also accumulators) which are used for indexing and as link registers. (These were R1 through R3 on the 360 and R12).

d) In addition, there are auxiliary R-regs on the 1108 if any use can be made of them.

4. STORAGE ALLOCATION

Storage allocation and use differs fundamentally because the 1108 is a word machine (36 bits) whereas the 360 is byte-oriented. Although the 1108 provides partial word designators in instructions for manipulating smaller pieces, there exists no mechanism to index within a word to the next logical quantum (such as the byte point on the PDP-10). This dictates that the elements of all arrays must be located in different words (and occupy the same location within the word).

The various partial words that can be handled are the following: Signed and unsigned half-words (18 bits), signed third-words (12 bits), unsigned quarter-words (9 bits), and unsigned sixth-words (6 bits). Unfortunately, the quarter-words and third-words cannot both be used since a bit must be set in the PSW to indicate which mode is currently being used. (Actually the quarter mode eliminates all three thirds and one of the halves). After analyzing the programs using XPL, and receiving information that quarter-words are unavailable on the 1108's at MSC, the following strategy was chosen:

a) Sixth-words: used for characters and for BIT $\leq$ 6 that are packable.

b) Quarter-words: cannot be used.

c) Third-words: used for packable quantities of items of type of BIT(n) where 6 < n $\leq$ 12.

d) Signed half-words: used for packable quantities of BIT(n) where 12 < n $\leq$ 16. (It is probably only of academic interest, but the last limit should actually be 18 bits).

e) Unsigned half-words: did not appear to be useful.

f) Signed full-words: used for FIXED and BIT < 16, and all items not indicated as packable.

Items in COMMON may be packed if they have a bit length of 16 or less and are dimensioned (arrayed). For NON-COMMON items, they must be declared to be packable by declaration keyword (either ARRAY or PACKED are favorite choices). Local variables may be determined to be suitable for packing by usage context.

5. CHARACTER HANDLING

Character handling was to be similar to the 360 except word addressing was used and characters are packed 6 per word. The similarities and differences include:

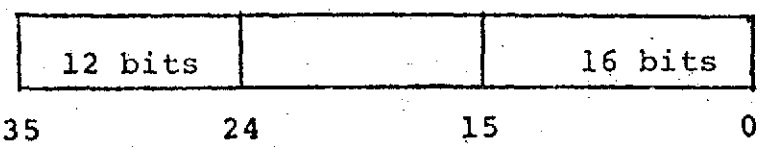
a) There was to be a free string area that was repeatedly filled as new strings were created.

b) A COMPACTIFY routine to condense these areas as necessary.

c) Strings to be designated by descriptors which would be kept in one contiguous area of memory so that these are accessible by COMPACTIFY for garbage collection. However, there was no need to limit this area to 1024 descriptors. There were to be two subdivisions within the descriptor area; one for COMMON descriptors, and the other for the rest. The best approach to form the descriptor group seemed to be to use a SEG card in the MAP processor to gather up the descriptors from all the separate assemblys. Were it not for the HAL multipass overlay requirements, a more elementary method of collecting descriptors might have been feasible.

d) A character assignment (MSG1 = CHAR2;) was to merely transfer the descriptor of CHAR2 to the descriptor location known as MSG1. Thus, a single string could comprise several character variables as was done on the 360.

e) The form of the descriptor was to be as follows:



The low 16 bits was an absolute pointer to the first word of the string. Zeros filled out the rest of the low 22 bits so that it could be used as an indirect address. (x, h, & i fields).

The upper 12 bits was the string length in characters; it could be fetched using a third-word partial word designator. This permitted strings to vary from 0 to a theoretical limit of 2047 characters. There seemed little need for the special handling accorded a zero-length (null) string on the 360.

f) The descriptor approach facilitated character procedures since they could return a full word descriptor in one of the A registers (A0) for further usage.

g) The character functions were similar to their 360 counterparts.

The LENGTH function is even faster than the 360 since there is no need of special tests for null strings and fix-ups if not because of the (length - 1) methodology of the 360.

The BYTE function has three cases:

1. Literal arguments were detected and accorded special treatment -- BYTE('H')

2. Numeric literal indexing could be accomplished efficiently on both left and right sides -- BYTE (MS6,7)

3. Variable indexing would be slower since it had to be done by subroutine because of the word organization of memory -- BYTE(MSG,J)

SUBSTR was slower then the 360 since it involved the creation of a new string rather than just a new descriptor. The reason for this decision was a desire to have strings start at a word boundary.

CONCATENATION was to be done in an analogous fashion to the 360.

6. ARITHMETIC ENVIRONMENT

Some differences do arise because of the 1's complement arithmetic used on the 1108. In particular, when the low 8 bits of -1 are examined, it is not FF but rather FE. (-0 reduced to FF). A systematic method to reconcile the HEX constants and negative numeric usages was sought. The language was extended to allow initialization with negative numbers. It was then mandated that HEX constants were not to be treated as signed on the 360.

7. CODING "TRICKS"

Advantage was taken on the 360 that storage into an 8 bit quantity (1 byte) masked off any excess bits. The effect was not identical on the 1108 if it was stored into a 12 bit storage quantum. (The 8's reducible to 6 bits were ok). The only way to exactly duplicate results would be to mask (AND with an 8 bit mask) before storage. This would have been too high a price to pay for 360 emulation when it was seldom really required. There are over 1000 STC in both HALPASS1 and 2 and 2000 STH. (This truncation may have been used on half words also). Besides, it seems inherently wrong to imitate the 360 for the purpose of propagating coding that has utilized machine dependent characteristics of the 360.

A better move seemed to be to eliminate this usage. A method was devised that trapped the dirty cases and flagged them for modification. In addition, it was helpful for people to point out all the places that they remembered using implicit characteristics of the 360 or seeing them used. A master listing was kept with all the trouble spots marked.

8. FORTRAN COMPATABILITY

FORTRAN compatibility was to be maintained if at all possible. "Compatibility" in this case, means only the ability to call FORTRAN subroutines, pass them arguments, and accept returning results from FORTRAN functions. It was anticipated that it would also be possible to call XPL procedures from FORTRAN but it is not a primary requirement and would take more effort.

A procedure or function call produced code that resembled the following FORTRAN example.

```
CALL FOOL(I,J,MEMDUM)
000117   7201 00 00 0 000000 0010
000120   7413 13 00 0 000000 0006    LMJ     X11,FOOL
000121   0000 00 00 0 000012 0000    +       I
000122   0000 00 00 0 000013 0000    +       J
000123   0000 00 00 0 000000 0000    +       MEMDUM
                                     +       0153,0
```

While this example is not exhaustive, it did illustrate the general format. The specific rules for subroutine branches were as follows:

a) Linking was to be accomplished via an LMJ using X11 as link register.

b) The argument list would immediately follow the branch instruction. The list would be constructed as addresses for each actual argument so that indirect addressing could be used to fetch their values. The list for each type was to be:

   1) Variables - the address of a full word variable (not packed). For character variables, it was the address of the descriptor.

   2) Constants - the address of a full word (36 bit) constant.

   3) Expression - the address of a temporary containing the resultant value.

   4) Subscripted variable - same as for expressions except for full word arrays when it is easy to generate the element address.

c) According to FORTRAN conventions, A0 through A5 and R1 through R3 may be modified in the subroutine. However, we planned to assume that almost all registers were invalid upon return. This required less register saving and restoring than FORTRAN. Calls to FORTRAN subroutines would then involve needless register saving but it was not incompatible.

d) A function would return its result in A0.

e) It was anticipated that the Walk-back location would be eliminated. To not do so would cost 1,000 words in HALPASS1. This would require some care in exiting from FORTRAN subroutines.

The rationale for the FORTRAN compatibility was to take advantage of existing FORTRAN capabilities in areas that were either not frequently needed in XPL or else difficult to implement. Some examples could include:

a) Floating point, single and double precision and conversions to and from integers.

b) I/O routines.

9. ASSIGN PARAMETERS

XPL was incapable of modifying parameters passed to subroutines because all calls were by value. For compatibility reasons, this decoupling would be maintained on the 1108 version even though the calling was by pointers. (See the next section for actual details). However, it was often useful to modify calling parameters by assigning them new values, (especially for arrays). The suggestion was to implement the HAL ASSIGN type of list in both procedure definition statements and calling sequences. Examples are:

```
CALL SUBGUM(A,B) ASSIGN(Y,Z);
SUBGUM: PROCEDURE(U,V) ASSIGN(W,X);
```

Before the ASSIGN all the usual XPL rules would apply. After the ASSIGN in the CALL statement, may come only variable names, with or without subscripts, but no expressions. The compiler would link them up by reference so that assignments in the subroutines will be reflected back in changes in the actual variables.

10. PROCEDURE PARAMETER LINKAGES

At procedure entry X11 was to be left pointing at a list of indirect addresses that permit accessing of the actual calling variables. Section 8 shows the list for FORTRAN calls. The treatment accorded each variable depends on how it is used in the procedure. Specifically,

a) If merely referenced in the procedure, it was to be accessed via indirect addressing.

b) If it was assigned a value (via the LHS of an assignment statement, or in other usages that could possibly change its value), its value would be copied into a temporary in a prologue and the temporary used exclusively.

c) If on the ASSIGN side of the list, it would always be referenced indirectly, including stores.

d) Arrays were to be permitted on the ASSIGN side and direct fetches and stores would be accomplished with appropriate indexing. (It was not clear whether arrays should be allowed on the other side of the lists; they were not functional in 360 XPL. If permitted, storing would be prohibited.)

11. REGISTER RECOGNITION

The System 360 has 16 general purpose registers, 15 of which may be used as base and/or index registers. The XPL philosophy allocated nine of these registers as base registers, whether the program required them or not. Three more were used to branching and subroutine linkage. This left four registers to serve as accumulators, only three of which could double as index registers. This severely limited the amount of information which could be retained in registers. Thus, no attempt was made to remember what a register contained once its value was used. On the other hand, the 1108 has 16 accumulators and 15 index registers (4 of which double as accumulators). Since many operations require a register pair, accumulators were managed as pairs, while indices were handled as single registers. Thus, at minimum 8 accumulators and 9 index registers (which is considerably more than an average XPL statement would

require) were available for use on the 1108. Thus, a system was developed which allowed the code generator to remember the contents of these registers for later use. The following quantities were remembered: 1) the name of the variable in the register, plus any variable and/or constant indexing which applied to the name; and 2) any additive constant modifier applied to the variable which changed its value from that in memory. Because of the bottom-up properties of the XPL synthesis, multi-level indexing could be remembered to as many depths as required, significantly reducing the number of storage references required by the compiled code (approximately 30% fewer instructions on the 1108 than for an identical program on the 360).

12. INSTRUCTION GENERATION

Where possible, instructions were not generated until they were absolutely necessary, such that the most information could be used to intelligently decide what code would produce the desired result. In general, constant terms in expressions were saved as modifiers, their value changing as other constant terms entered into the expression. Any constant operating on another constant was evaluated at compile time, the result being a new constant term. Any constant added to or subtracted from a variable was retained as an expression modifier, to be generated only when the expression value was forced to be evaluated. If the value or variable term represented in the compiler stacks matched up with the contents of a register whose value was known, the expression was suppressed and the register value was used instead (in some cases, it was necessary to move the register contents, if other sub-expressions required the register, and its contents were to be altered).

Unlike the 360, virtually all non-branch types of instructions on the 1108 can use immediate operands (operands specified within the actual instruction). Any constant whose absolute value is less than 2^{16} can be specified in this manner, effecting a saving in generation of constants, as well as memory references.

Statement constructions of the form "IF <relational expression> THEN" generated a TEST, JUMP sequence. Otherwise, a bit conditional was generated, using a SET TRUE, TEST, SET FALSE sequence, to be subsequently used in a JUMP testing the resulting condition. Any relational expressions involving constant terms on both sides of the relational operator are balanced, such that the constant term of one operand (the one to be used in the TEST operation) is zero. Thus, (A+3) > 8 becomes A > 5, and (A-4) < (B+5) becomes (A-9) < B. Also, since the 1108 has no "less than" or "greater than or equal" test instruction, the "greater then" and "less than or equal" tests must be used with the respective operands reversed; i.e. A < B is coded as B > A.

Instructions and data were kept physically separate, so that the two bank interleaved fetch properties of the 1108 could be taken advantage of.

Unlike other compilers, the XPL code generator does not attempt to save and restore the registers which are used within a procedure (except the linkage registers). Instead, registers whose contents are vital are saved prior to calling a function, and restored upon return. Registers considered non-vital are merely treated as if their contents were destroyed during the function, and are no longer considered to have recognizable contents. Except in very complex expressions involving functions, especially character functions, register saving is never done. In the HAL Compiler, with its 130 procedures, less than 50 register saves are performed.

By definition of the XPL grammar, any index expression on the left-hand side of an assignment will be the first quantity forced into a register. This classed the register as vital over any functions which appeared on the right-hand side, forcing many unnecessary register saves. For simple indices (variable ± constant), the loading of the index expression is deferred until the right-hand side is evaluated, thus eliminating many such register saves.

13. DATA ALLOCATION

Data on the 1108 compiler falls under three main classifications: FIXED, BIT, and CHARACTER. Within BIT are 4 sub-classifications: BIT(6), BIT(12), BIT(18), and BIT(36), which corresponds to the allowable partial word designators in 1108 instructions. All but BIT(6) are classified as signed quantities. The data generation section of the compiler follows a number of rules: 1) data declared as COMMON, ARRAY, or global in modular compilations, is explicitly packed or not-packed strictly on the declaration properties, whereas all other data attributes are subject to change depending upon its usage within the program; 2) all unpacked data is generated in the order in which it is declared; 3) packed data is sorted down by bit length, and secondly by array length. Side-by-side arrays of like data types are generated from longest to shortest until all are exhaustive. All uninitialized data is implicitly set to zero.

The following rules determined how data might become unpacked: 1) use as an ASSIGN parameter, 2) use as a formal parameter, either by name alone or modified by a constant index; and 3) not having the ARRAY attribute in global declarations. The first two reasons result from the implementation restriction that all formal parameters must be passed as full words. It is less costly to force a BIT(6) simple variable to occupy a full word than to load a packed variable and store it in a full word temporary for passing into a procedure. The third reason merely assures an identical storage layout for global data regardless of the contextual uses within the various modules sharing this data.

Any integer initialization is passed to the assembler as signed decimal numbers (negative initialization was added to the language). Any data initialized with hexidecimal or binary constants are converted to the corresponding octal representation on the 1108 (since 32 bit masks on the 360 would not be identical on the 1108 if passed as signed numbers). It is assumed, therefore, that binary initial values are not utilized as signed quantities in the XPL program.

Data is isolated into the following categories for later use by the 1108 MAP processor and collector:

1) initialized string data (all modes)

2) local data and literals

3) local string descriptors

4) global data (shared between co-resident modules)

5) global string descriptors

6) common data (shared between both co-resident and overlay modules)

7) common string descriptors

14. MODULARIZATION

Although the 1108 assembler is capable of assembling very large source programs, there is a finite maximum number of source cards it is capable of absorbing. To avoid this problem, the XPL compiler was extended to allow both EXTERNAL and ENTRY properties, permitting intermodule communication. The global data declaration facility can be used in conjunction with this facility, if so desired. Although Phase I was only separated into two major modules (scanning and analysis), it could easily have been modularized to the point where each procedures was a separate compilation. More importantly, however, this facility can be used to group mutually exclusive collections of procedures to generate an overlay structure, should space limitations become a serious problem. (The 360 implementation has the advantage of growing into any size partition the host operating system will allow).

3.3 Implementation

Implementation of the HAL 1108 compiler followed the procedure described in Section 3.1 and used the guidelines of Section 3.2. The tasks follow in almost a straight chronological flow from the start of the effort through final delivery.

1. Design of the 1108 code generator that described each of the XPL constructs and their equivalent on the 1108 was undertaken. These followed the 1108 guidelines described in Section 3.2.

2. XPL was rewritten to produce 1108 assembly language and the 1108 constructs per the design guidelines. This version of XPL was compiled and debugged on the 360.

3. XPL 1108 subroutines and supporting routines were written in 1108 assembly language and assembled for the 1108. This involved conversion routines, character handling routines, and input/output routines, both sequential and direct access. These were programmed and debugged on the 1108, both at Intermetrics, Cambridge on a rental 1108, and at MSC in Houston.

4. When the 1108 XPL was thought to be producing reasonable code, the 1108 assembly language out of the 360 was taken to the 1108 where it was assembled, loaded, and debugged on the 1108. The result was that several simple XPL programs, such as ANALYZER, could be compiled on the 360 and executed on the 1108.

5. At this juncture, the XPL compiler itself was fed through the 1108 XPL code generator running on the 360 and the resultant assembly language taken to the 1108 added to the supporting routines and debugged on the 1108. When this was successfully completed, a working XPL compiler had then been bootstraped from the 360 to the 1108. From this point on, XPL programs could be compiled as well as executed on the 1108.

6. Work that had been proceeding on the HAL pass 2 code generator to produce 1108 FORTRAN was not a large effort since a great deal of attention had been paid in the attempt to conform to ANSI standard FORTRAN. However, there were a number of differences and in particular, the data organization had to be changed to reflect the word size, and word structure of the 1108 versus the bit oriented 360. However, this was accomplished during the XPL development process.

7. Work commenced on the 1108 HAL support routines. These included the usual HAL supporting and library routines, such as the vector/matrix package, the math routines, the character routines, post-mortem dump, and input/output. Most of these were written in FORTRAN and therefore the transfer was accomplished easily. However, there were a number of changes, especially in any area where the data had been packed for efficiency reasons, or for address constant restrictions on the 360 (i.e. LOGICAL*1, INTEGER*2). However, the sort of changes that were required could be and were done in a systematic method. Some routines, such as character handling, had to be changed drastically. However, the resultant changes effort required was much smaller due to the FORTRAN than would have been otherwise.

8. At this point, work had to begin on HAL Pass 1 and Pass 2 to make them compatible with 1108 XPL. In particular, a number of minor changes were required, such as the use of hex constants for negative values in certain areas. But, beyond these minor fixes, the major cause for revision was the different data structuring required in the 1108. In particular, large tables which had been 8 bit quantities in the 360 had to be re-analyzed to see if they would fit in to 6 bits or 12 bits on the 1108, which were the quantums of memory that could be dealt with directly. The same held true for 16 bit data, to see whether it could be reduced to 12 or had to be increased to 18 bit. XPL 1108 had already been written to permit packing of these different tables into the same words on the 1108, which was a necessity because of the indexing on a word oriented machine. This proved straightforward in an XPL 1108: however, initialization of this data area was a troublesome problem. Nevertheless,

workable techniques were devised and implemented. The requirements for this data packing came about because of the size restriction on the 1108 HAL compiler. Only about 53,000 words of memory were available to a user program under EXEC 2, the operating system which as all there was for a given pass, such as pass 1 and included the area required for code, data, buffers, and free string area. It was a fairly tight requirement compared to the 360 memory availability. It was readily achieved. The key ingredient in this success of the process to shoe-horn the HAL compiler onto the 1108 was the success of the 1108 XPL implementation. The design appears to be a good one, and is implemented efficiently on the 1108. The measure of the 1108 XPL design efficiency, the number of instructions required for the same pass 1 of the compilers was reduced from approximately 45,000 on the 360 to under 30,000 on the 1108. And, this under 30,000 figure included about 4,000 words of address constants, which were not required on the 360.

9. Finally, pass 1 and pass 2 were compiled for the 1108. The results assembled using the 1108 assembler, they and their library routines were loaded and executed and debugged on the 1108. When this process was successfully completed, HAL programs could then be compiled on the 1108.

10. The compiled HAL programs produced 1108 FORTRAN which was fed through the 1108 FORTRAN compiler and combined with HAL 1108 support and library routines written in FORTRAN for the most part, but some assembly language, (see item 3 in the list) the combination loaded and executed on the 1108. Successful completion of this phase of the process was that HAL programs could then execute on the 1108, thus terminating the move process. HAL was thus a totally independent and operational compiler system on the 1108. A number of HAL test cases were successfully compiled and run on the 1108 and their result compared quite favorably with their 360 counterpart to within the accuracy supported by the differences in the machines word length and data types.

The HAL compiler that resulted from this effort, the HAL 1108, with all things considered, was a very good implementation of HAL. In many ways, it was superior to the 360 implementation, faster and more efficient than the 360 HAL. However, it never received the amount of usage and exercise that was undertaken using the 360 HAL. One reason for this is that the HAL 1108 was not completed until January 1973. By this time, HAL had been picked for the Space Shuttle. Shuttle work had begun in earnest, and the definition of HAL/S was already undertaken. None of the Shuttle contractors had an 1108 whereas all of them have 360's. JSC did have 1108's that had already done much of its Shuttle work using other methods, were committed to other approaches. However, this does not detract from the intrinsic merit of the HAL 1108 compiler. Much was learned from the process of moving compilers from one machine to another (e.g. the essentials of maintaining transferability), and from creating a machine language code generator for 1108 XPL, (e.g. the design problems of two quite different type instruction architectures). It was learned from these processors that they should find their way into better compiler and code generators for HAL/S on the AP-101 and 360's, and any other possible HAL/S compilers that might be undertaken in the future.

4. HALM IMPLEMENTATION STUDY

This study was for the development of an instruction architecture to support HAL/S and the investigation of micro-processors in order to implement the resultant architecture. The results of this study include:

1) The investigation of addressing structures for the support of higher order language instruction architectures;

2) the results of a partial implementation indicating possible modifications to HAL/S and desirable modifications for a support micro-processor; and

3) a comparison of the initial instruction architectures code size with respect to current instruction architectures.

Table of Contents for Chapter 4

4.1 Introduction and Overview

Higher order languages have been accpeted in recent years as the proper method for programming software projects. HAL/S is to be used in the Space Shuttle program for the coding of the actual flight computer. While the advantages in software cost savings with the use of higher order languages has been well known and documented [Bo 73, Ca 68, Co 68, Co 69, Gr 70], there has often been the fear of a corresponding hardware penalty. The argument has often been receited that a higher order language generates inefficient machine instructions. The natural result of this consideration and the insentive to use higher order languages, has been the development of various machine instruction architectures which are directly oriented towards the higher order language(s) being implemented. This problem is most acute in the aerospace industry where efficiency of memory usage not only correlates to dollar cost, but also to weight, physical size and power consumption. Thus, an avid interest in higher order language instruction architecture has occurred in this industry [Co 72, Ke 70, Kr 70, Mi 72, Ni 72, We 71].

While it was admitted that an instruction architecture oriented towards a higher order language provided for efficient code generation and execution, it was sometimes questioned as to whether this was accomplished by an undue excess in hardware size and complexity. Results of the micro-program implementation of the SUNY at Buffalo's BSM instruction architecture [Lu 72, p. 15] on the QM-1 micro-processor shows that the only "complexity" in implementation is in the address (GEA: get effective address) routine. But, if the support processor aids in the function required, even this is not complex. The results of encoding higher order language emulators and second generation instruction architectures on the B1700 which have been reported by W.T. Wilner [Wi 72c] indicate that the number of bits needed to encode their respective instruction architectures is equivalent. Wilner's results lead him to claim:

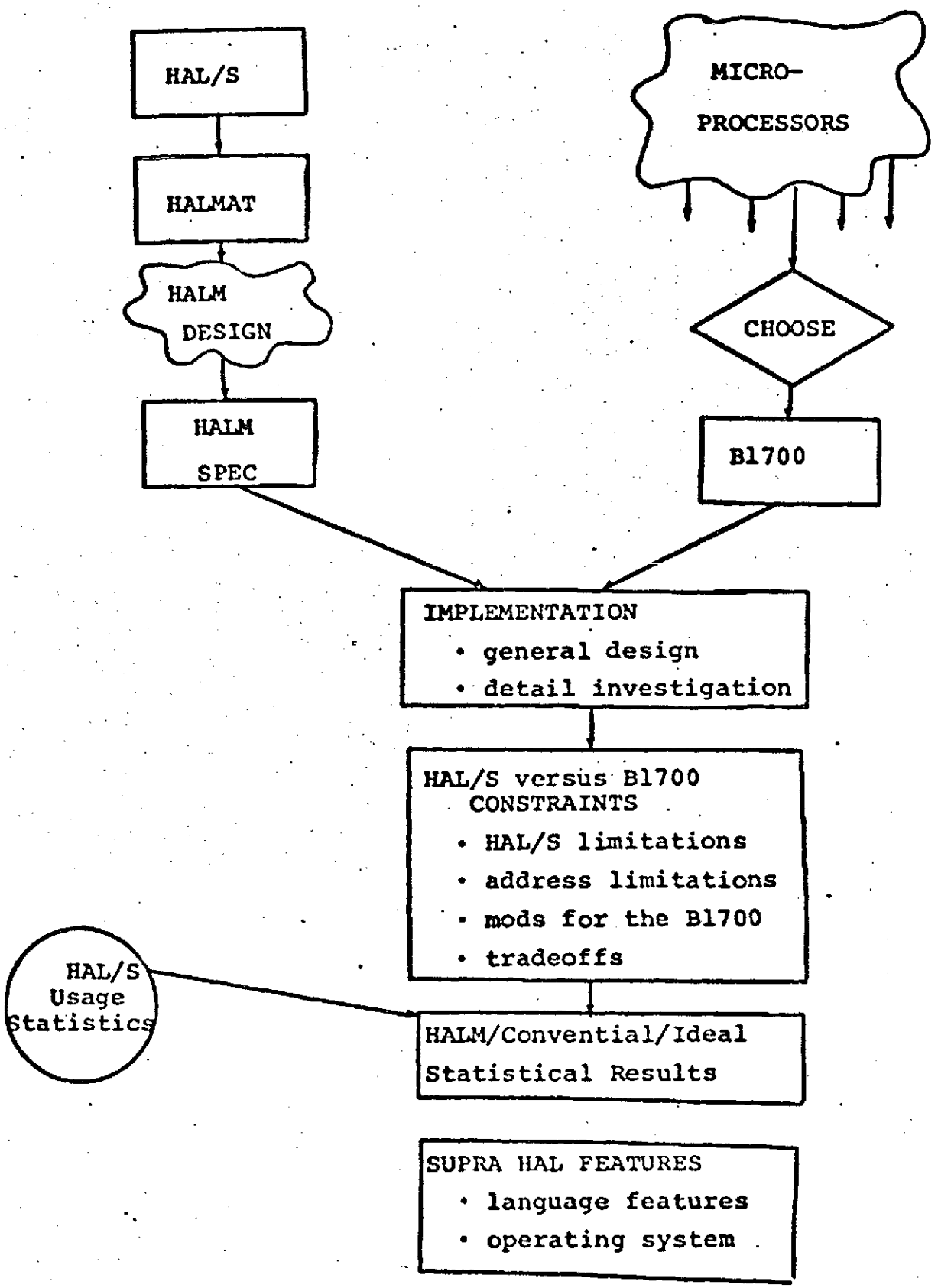


Figure 4.1-1: Study Summary

> "No matter what one is emulating, whether it be a second-generation computer, a contemporary programming language, or a futuristic abstract machine, one's interpreter tends to contain 28,000 bits of code for virtual hardware and from 5,000 to 25,000 bits of debugging aids (e.g. trace, dump, symbolic modification of memory).
>
> [Wi 72c, p. 105]

The purpose of this study was basically two fold. First, it was to develop an instruction architecture suitable for the efficient implementation of HAL/S. Secondly, it was to investigate micro-processors in order to determine and then use a suitable micro-processor for the implementation of the resultant HALM instruction architecture.

Figure 4.1-1 gives a diagram of the work performed to accomplish this study. Section 4.2 will discuss the relationship between HAL/S, its intermediate language HALMAT, and develop an initial architecture for a HAL machine, HALM. Section 4.3 will indicate the importance of addressing considerations in the development of instruction architectures and analyze the requirements made by HAL/S upon any proposed implementation methodology. It is particularly in this area that both incremental improvements and major reorientations to instruction architectures may occur. Section 4.4 will discuss the major areas of design differences of micro-processors, provide a description of the various micro-procesosrs under consideration, and indicate the choice of the B1700 for this study. Section 4.5 will give the results of the partial implementation of the modified HALM instruction architecture on the B1700. Section 4.6 will discuss the results obtained from the implementation with respect to desired modifications in both the HAL specification and in the support micro-processor. Section 4.7 will discuss the meaning of comparison between various instruction architectures, methods for performing such a comparison, and will give a brief comparison between HAL/S code generated for the IBM 360, AP-101, and the initial HALM instruction architecture.

Section 4.8 will indicate other areas besides the HAL/S instruction architecture where a micro-processor capability is of use. Section 4.9 will provide a brief summary and conculsions of the study.

Two appendices are also included with this Chapter. Appendix 1 gives a HAL/S program example and the code generated for it on both the IBM 360 and AP-101. Appendix 2 contains the same HAL/S program example encoded in the initial HALM instruction architecture. Included in Appendix 2 is a statement for statement comparison of the code generated for the program for each of the three instruction architectures.

4.2 HAL/S-HALMAT-HALM

One of the major tasks in this study, and that which forms a basis for the remainding tasks, is the development and designation of an instruction architecture for HAL/S implementation.

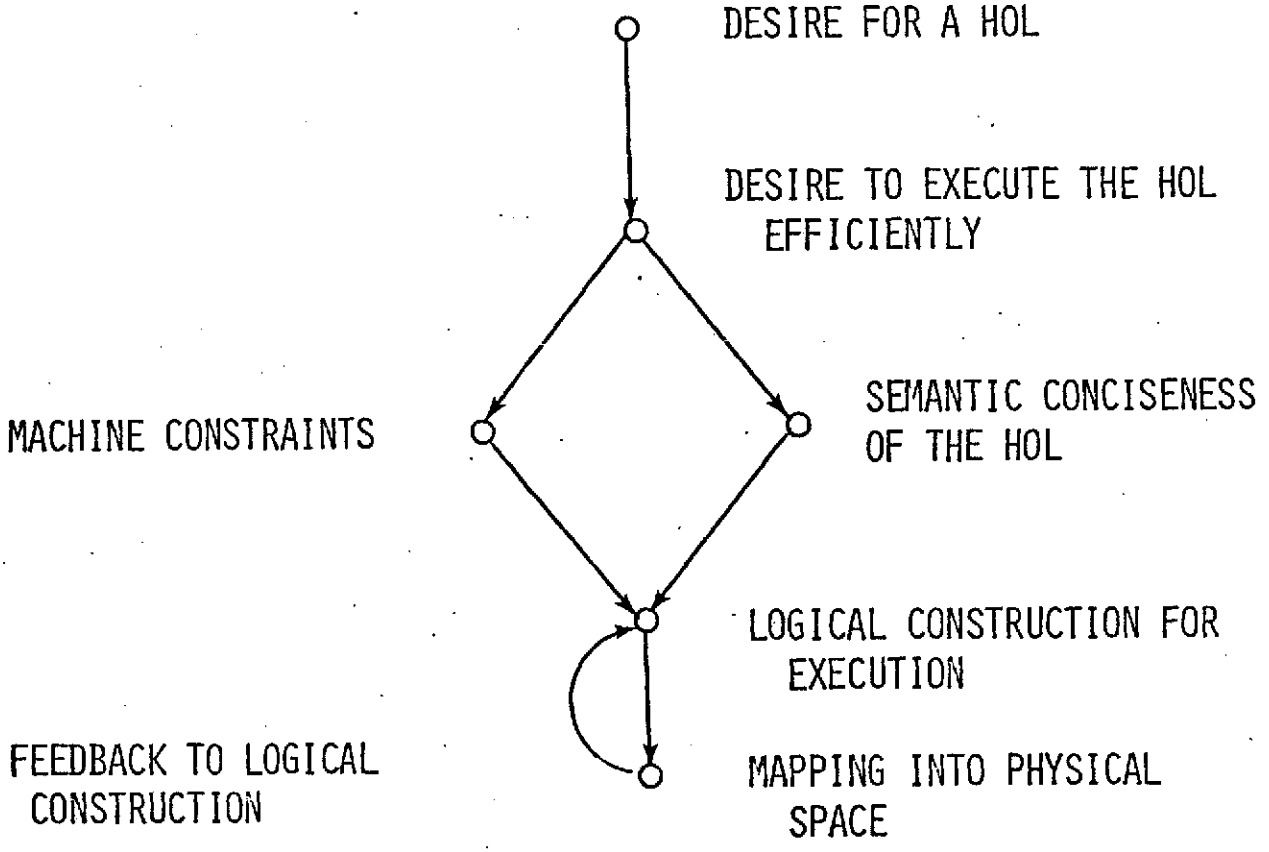
Under contract to NASA/JSC in 1972, Intermetrics developed an instruction architecture for HAL implementation as part of its multi-processor design. Chapter 2 of the final report of that contract [Mi 72] discussed in detail the design rational and methodology along with the resultant instruction architecture.

4.2.1 Design Methodology

The methodology used in the development of a higher order language machine is graphically represented in Figure 4.2.1-1.

The desire to use a higher order language is now a commonly accepted idea at NASA. The advantages of documentation, communication, maintainability, shortened programming time, fewer conceptual errors, no machine oriented errors and ease of learning have all become self apparent. HAL/S is now being used to program the Space Shuttle computer. Having accepted the use of a higher order language, the next step is to implement it efficiently.

In the aerospace community in particular, there is the requirement to have efficient execution. In particular, memory is costly, power consuming, weighty and physically large. But it is also true that the aerospace environment has many aspects which do not bear directly on the design of a general computer. Some of these aspects include the assumption that the architecture can be tailored to a single language or at least a similar family of languages. The actual use of an aerospace computer also facilitates the assumptions that it has a relatively small memory size, and that there exists reasonable limits upon the complexity of the operating system environment with respect to the number of processes in existance. Similarly, the addressing space can in general be considered to be smaller than on a commercial computer since there is a pragmatic limit to the number of variables in use. Memory management often can be accomplished in a relatively static fashion since telemetry and hybrid simulation requirements often can make mandatory a correlation between the physical address and the logical entity; reliability considerations often prohibit the free use of secondary



DESIGN AND IMPLEMENTATION OF A HOLM

Figure 4.2.1-1

storage. The result is to eleviate many of the problems found when dynamic address management is required.

An efficient implementation of a higher order language instruction architecture has two aspects. One is the problem of forming a concise logical representation of the HOL. The second problem is to do this in such a way that it can be implemented in a cost efficient manner. It is this second constraint, the understanding of current technological limitations and the availability of support micro-processors, that limits the capability of a logical instruction design. Thus, for example, the arithmetic data type formats supported by a language are usually driven by the hardware available with the computer upon which the language is implemented.

On the IBM 360, HAL/S supports 32 and 64 bit floating point scalars. If it were implemented on another machine, such as the B6700, it would be supporting 48 bit floating point scalars. Indeed, on the Singer SKC-2000, although it has 32 bit floating point capability, this is of a different format than that of the IBM 360. Similarly, the quantum of data which is easily manipulated, and thus, efficiently supported by a language on any given processor varies. The IBM 360 supports addressing easily to the 8 bit byte level. The IBM AP-101 supports addressing only to the 16 bit half word level. Other computers support 18 bit units or 24 bit units. These data widths in turn tend to force design decisions upon an instruction architecure. Thus, for example, descriptors would be given but a single length such as 32 or 64 bit widths. If more information need be encoded, then multiple descriptors of this basic unit length would be used. And indeed, this form of machine constraint was a major driver in the handling of multi-rank descriptors in the HAL instruction architecture [Mi 72]. It is to be noted, however, that all of these particular machine constraints are not now inherent in current technology. For example, the Burrough's B1700 supports bit addressing of memory with (basically) any bit field width; and similarly it is possible with the B1700 to execute arithmetic data efficiently in varying widths.

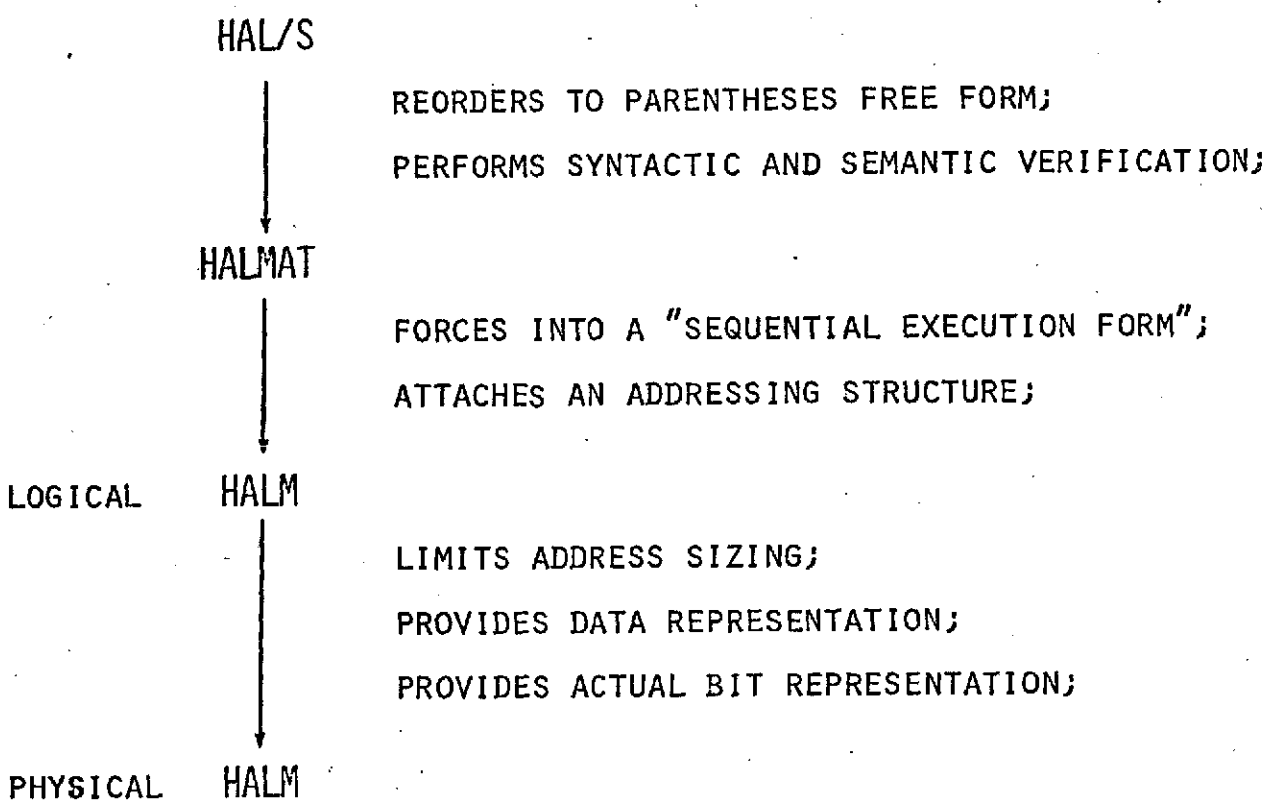


Figure 4.2.1-2: The Steps of Translating HAL/S to an Implemented HALM

Figure 4.2.1-2 is a slightly different representation of this critical step in the design process. HAL/S is the language to be implemented. There already exists an intermediate language for code generation called HALMAT. The object is to first take the intermediate language HALMAT and make it into a logical design for a HAL machine. It is this step that takes into consideration the real world of current technology and available micro-processors.

The translation from HAL/S into HALMAT accomplishes several purposes. First, it reorders the HAL/S language from a parenthetical language into a parenthetical free notation. Effectively, this is a reordering of the code which places operators and operands into a sequential form, a polish notation, so that each particle is meaningful. In the process of performing this reordering (i.e. parsing) the compiler has also performed syntactical verification and then performs appropriate semantic verification. Figure 4.2.1-3 represents such a translation from HAL into HALMAT.

The translation from HALMAT into a logical HALM again accomplishes several purposes. The current technology in general requires that an instruction stream be of a single instruction single data form (SISD). That is, a single operator is executing at any given instant upon a single object (perhaps of several operands). This is to distinguish between, for example, array processors or tree structured execution [refer Mi 72,pp. 25-28]. Included in this consideration of arranging the code for proper execution would be making explicit all the required operands. Thus, DO FOR...END; statements needs six operands: the iteration variable, the initial value, incremental value, limit value, and the next instruction address within the loop, and the next instruction address when the loop is finished. This sixth operand, for example, is not rectified in just the HALMAT. It is a pragmatic concession to machine efficiency that it is included with the operation. For example, when the loop is finished it would be possible for the processor to keep reading each instruction (and performing a NOP) within the DO block, until the END statement is located.

assignment with terminal subscripting

```
DECLARE V VECTOR (6),
        M MATRIX (7,4);
```

$$\vec{V}_{1 \text{ TO } 3} = \bar{M}_{3 \text{ AT } 4,2} + \bar{V}_{4 \text{ TO } 6};$$

1				13			Ø	N	SMRK	
2	$\vec{V}$		SYT					A	RALC	receiver subscripting
3	1		IMD	3	1	IMD		A	TTSB	
4								A	ALCE	(note 1.)
5	$\vec{M}$		SYT					A	ALC	matrix subscripting:
6	4		IMD	$\vec{3}$	2	LIT		A	TASB	row then column
7.	2		IMD		1			A	TIDX	
8								A	ALCE	
9	$\vec{V}$		SYT					A	ALC	vector subscripting
10	4		IMD	6	1	IMD		A	TTSB	
11								A	ALCE	
12	5		VAC	9		VAC		Ø	VADD	
13	2		VAC	12		VAC	Ø	Ø	VASN	assignment

Notes:

1. It is difficult to predict the order of emission of the subscript constructs for different variables in the same assignment.

[Ii 71]

Figure 4.2.1-3

The other main function that is performed by the translation between HALMAT and the logical HALM, is the attachment of an addressing structure. HALMAT refers to variables via symbol table reference and their value space does not in reality exist: only the requirements for it are described within the symbol table. And as was seen above, there are various flow addressing questions to be resolved. The question of addressing at this point can change the design drastically. It is possible to support a tagged, stack oriented architecture or a normal Von Neumann architecture or any point in the spectrum between them.

Both Figures 4.2.1-1 and 4.2.1-2 indicate there is a final translation between the logical HALM design and its physical implementation. It is during this last translation step that the final constraints are placed upon the use of the higher order language. It is here that physical limits are placed upon the addressing capabilities of the design, the number and type of data operands and the flow addressing capabilities. For example, it would be possible to keep the same basic design of the AP-101 yet reduce its displacement fields in SRS instructions from 56 to 48. Or, it would be possible to add a new addressing form allowing 32 bits of addressing space, i.e. a 3 half word instruction with 2 half words of addressing. In the case of a higher order language architecture, these limitations could either be in the field sizes or, indeed, the number of different formats made available.

In general, the implementation on a particular processor forces the exact data representation upon the implemented language. The processor effectively defines the size (16 bit, 32 bit, ...), the representation (sign magnitude, 2's complement, ..., ASCII, ...), and restrictions (only single precision) upon the languages data formats. And, of course, the physical implementation places an actual bit representation upon the operators and operands.

Figure 4.2.1-1 has one further line in its graphical representation. This is the feedback from the physical implementation to the logical design. This represents both the continual improvement possible with the gathering of actual statistics, and the discovering of problem areas in the logical design when the instruction architecture is actually used.

Further details on design methodology with examples may be found in Chapter 2 of Mi 72.

4.2.2 Initial HALM Design

The question of addressing structure is considered to be both the most controversial and the most important issue in the design of efficient HALM architectures. Section 4.3 will go into more detail in the discussion of this subject. Because of the interest in the addressing problem, it was felt that an initial instruction architecture must be choosen in order to pursue the micro-programming implementation aspect of this study. It was therefore decided to use the instruction architecture developed by Intermetrics in Mi 72 (designated MP) as the baseline for micro-processor considerations, while at the same time separately pursuing the design issues of addressing and HALM modularity. The choice of the MP architecture as the baseline was subject to varying degrees of modification when first the micro-processor was choosen and then comparisons were to be made. Section 4.2.3 will briefly discuss some of the features of an instruction architecture that are basically independent of each other, and are thus subject to change without affecting the total design.

The MP architecture is basically a modification of the Algolish design of the Burrough's B6700. It consists of a tagged architecture, stack oriented with a polish instruction stream. The floating point data types of the MP are of a compatible precision and range as that of the IBM 360 and AP-101 however.

Figures 4.2.2-1 through 4.2.2-4 briefly summarize this instruction architecture. Figure 4.2.2-1 presents the instruction set and is divided into their functional categories. Figure 4.2.2-2 presents the special words which are required for the addressing of both formal parameter and flow control. Figure 4.2.2-3 presents the format of the descriptors used within this architecture. Figure 4.2.2-4 represents the arithmetic data types as supported in the MP architecture. It also indicates the transformation that takes place between its main memory representation and its representation when residing in the stack.

A full description of this architecture is to be found in Mi 72 Section 2.4, pp. 88-156.

It is to be noted that the MP architecture was predicated upon a syllabic orientation: that the implementation would operate with a preference for 8 bit bytes. While this was a reasonable initial assumption, the B1700 is bit oriented and does not require this orientation. Thus, this machine constraint is removed when the B1700 is the host micro-processor. This would principally have design repercussions in the various format constraints:

- no need to have 8 bit quantums for operators
- no need to keep addressing within 16 bit units
- no need to keep the arithmetic data types as a multiple of 8 bit units.

The main effect upon the instruction architecture, therefore, is with respect to the actual instruction encoding and data elements available for execution.

OPERAND META-OPERATORS

COPY
GET
ADR
ADRE

LITERALS

LTS4
LTS10
LTS15
LT32
LT32F
LT64
LTS7M
LTLD
LTLDX

NAME MANIPULATORS

LOAD
LDRK
INDX
LIM
AROF

STORE

STD
STN
STDI
STNI
BST m,n

ARITHMETIC MANIPULATORS

ADD
SUB
MUL
DIV
CHSN

LOGICAL MANIPULATIONS

EQUL
GREQ
LSEQ
SAME
LAND
LOR
LNOT
BSETL n
BRSTL n
BCHGL n
BTSTL n
BSET
BRST
BTRN
BLD m,n
BOUT m,n
BIN m,n

ARRAY AND MEMORY

STT
STT3

FLOW CONTROL

JOT
JOF
JMP
JCC m
PRCS
MKS
ENTR
RTRN
EXIT
FOR

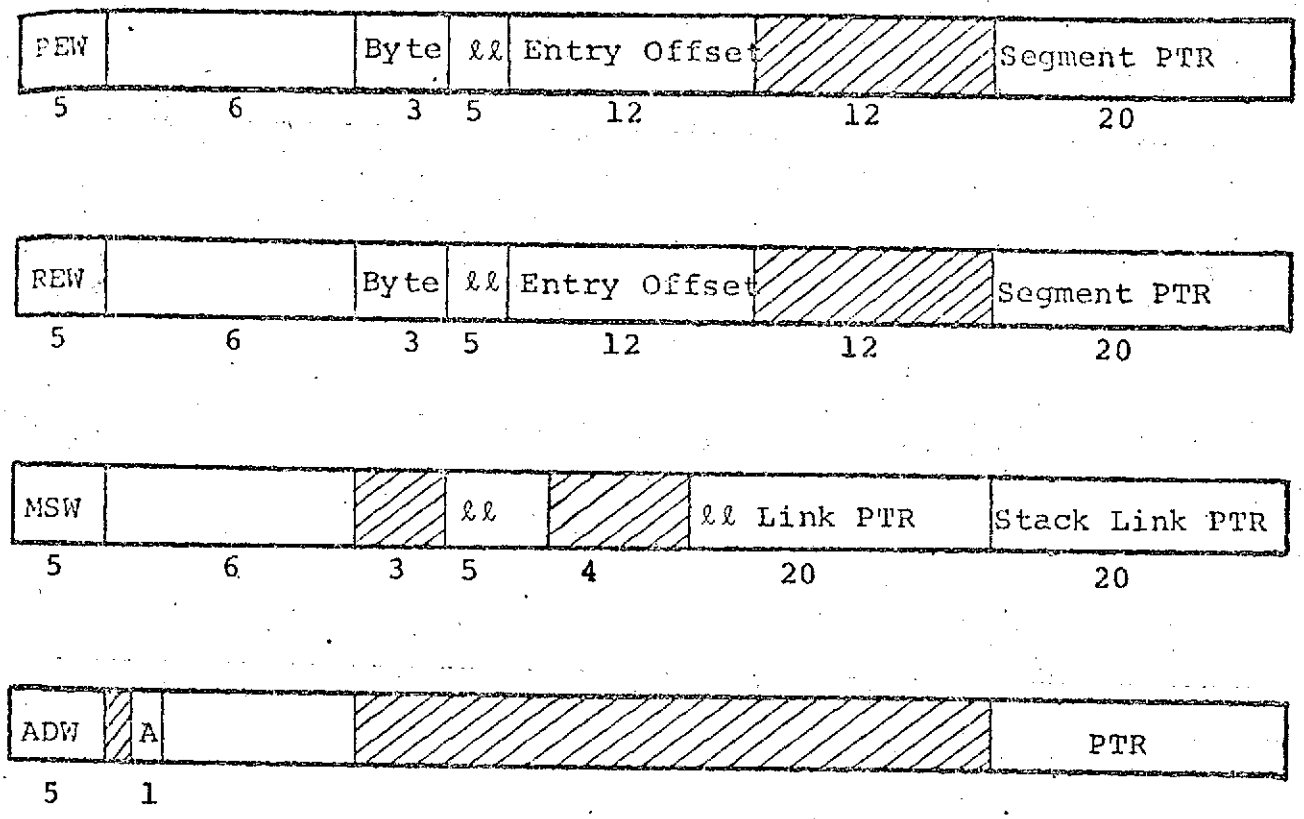
DATA FIELD MANIPULATION

INTT
INTR
FRAC
CCAT
ITOC
STOC
CTOI
CTOS

SYSTEMS CONSIDERATION

XCH
DLET
DUPL
NOP
IDLE
HALT
LDPR
STPR
STLD
LDID
IPC n

Figure 4.2.2-1 Instruction Repertoire



PEW: Program Entry Control Word

ℓℓ: Lexical level of procedure to be entered
Segment PTR: Stack number-offset of program segment
Entry Offset: Double word offset within program segment to which to transfer control
Byte: Byte identification within double word entry offset

REW: Return Entry Control Word

ℓℓ: Lexical level of procedure when control is returned
Segment PTR: Stack number-offset of return program segment
Entry Offset: Double word offset within return program segment to which to return control
Byte: Byte identification double word return entry offset

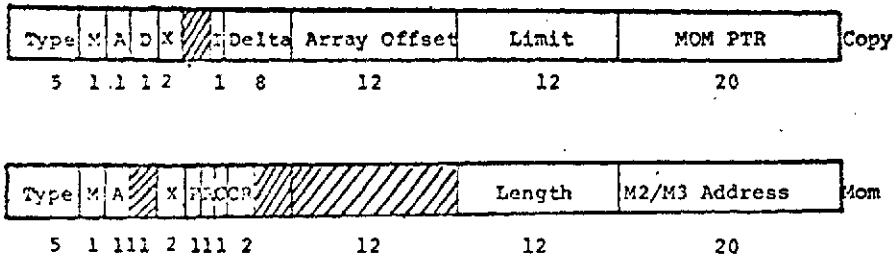
MSW: Mark Stack Control Word

ℓℓ: Lexical level of indicated procedure
Stack Link PTR: Stack number-offset of previous MSW
ℓℓ Link PTR: Stack number-offset of previous lexical level MSW

ADW: Address Word

A: Access Bit: either read/write or read only allowed
PTR: Address pointer in stack number-offset representation

Figure 4.2.2-2 Special Words



Type = Data Type Form:

- AR8: eight bit arithmetic array
- AR16: sixteen bit arithmetic array
- AR32: 32 bit single precision floating point array
- AR63: 63 bit double precision floating point array
- CHAR: character array
- PROG: code segment
- GEN: untyped descriptor

M = Copy Descriptor (uses stack number-offset pointers)
Mom Descriptor (uses M2/M3 address pointers)

A = Read/write access allowed from array
Read only allowed from array

D = Single rank array, no additional rank information present
Multiple rank array, more rank information follows

X = Compool bits:

- 00: Normal non-Compool
- 10: Compool unreferenced
- 11: Compool referenced

I = Delta, array offset, limit fields refer to (sub) arrays
Delta = 0; Limit = 0; Array offset = single element index

Delta = Distance between elements in this rank
Distance in units of elements

Array offset = index into array of first element of this rank;
In units of elements starting at ϕ

Limit = Maximum limit for index into this rank in units of elements

MOM PTR = Stack number - offset of associated mom descriptor

P = Presence bit: either M2 or M3 address

R = Refer bit: segment has been referred to either by reading writing into it

C = Changed bit: Segment has been written into

CR = Critical information:

- 00: Normal, one copy stored
- XX: Critical, duplicate copies interleaved
  - 11: Both copies good
  - 01: "One" copy good, use this one
  - 10: "Other" copy good, use this one

Length = Length of segment in units of that array type (*critical data segment twice as long as length indicates)

M2/M3 address = Physical address of the segment

Figure 4.2.2-3 Descriptors

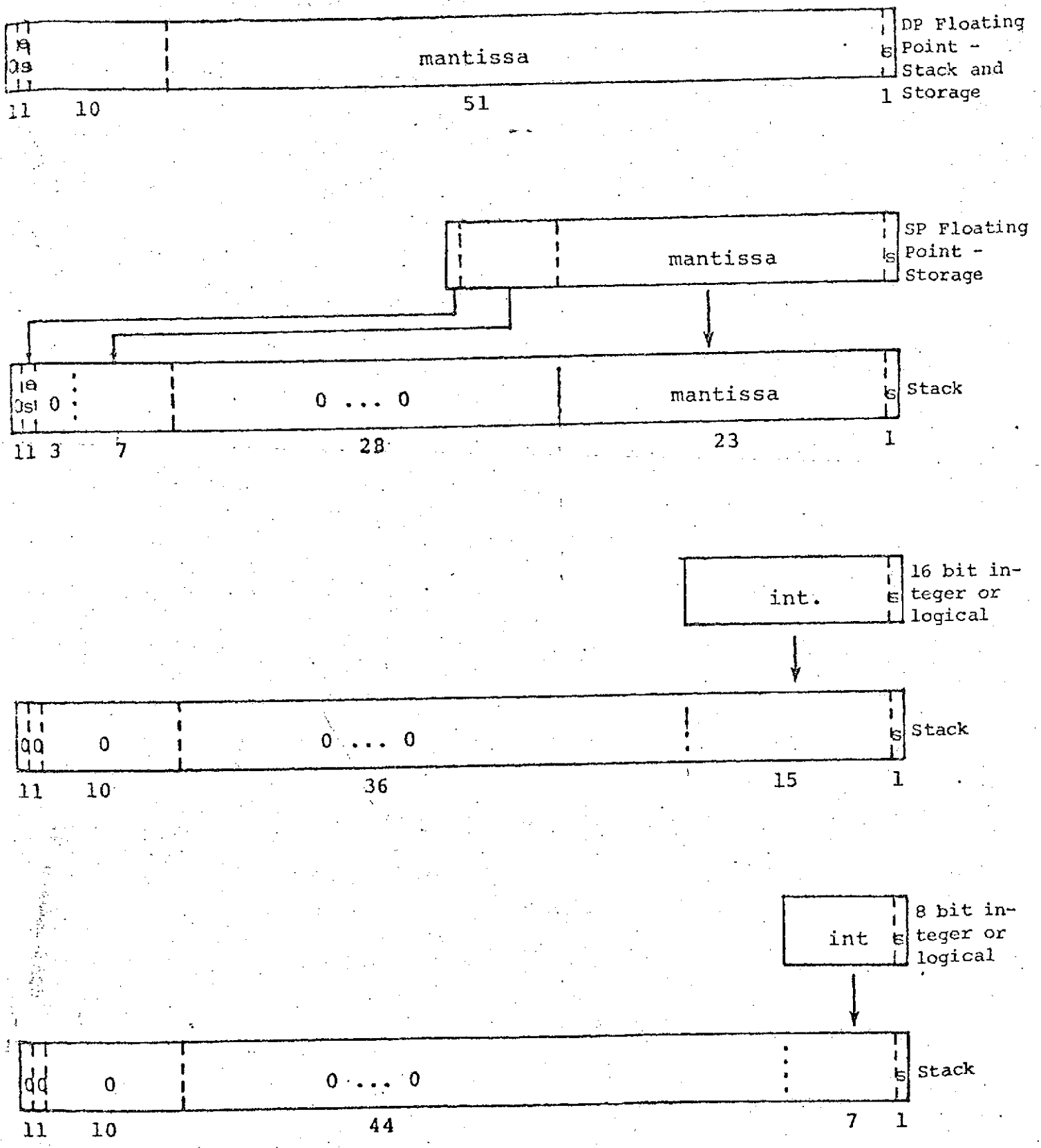


Figure 4.2.2-4 Arithmetic Type Formats and Mapping to Stack

4.2.3 Separable Implementation Issues of Instruction Architectures

Reflection upon HAL/S or other higher order languages will show that there are several areas that are basically independent of each other. These include the control sequencing of the HOL, versus the data addressing methodology, versus the set of functional transformations (operators) of the languages, versus the data representation. It is easy to conceive that any one of these areas may vary in their implementation methodology and physical representation with minimal effect upon the other areas.

4.2.3.1 Control Sequencing. The method employed for the implementation of the CALLs, RETURNs, DO FORs, GO TOs, etc., must as a minimum reflect the semantics of the language definition. If it is to be an efficient implementation, it should reflect the HOL structure, taking into account the properties of block addressing. It is also important that the implementation be efficient with respect to machine constraints and thus be able to be executed from a local context.

One can conceive that a language such as HAL/S could remove its GO TO and implement a LEAVE. Or it could modify its Procedure and Function and other block structures. These would not affect the data addressing, data representation or data transformation operations.

It is true, however, that the change of a tagged architecture to/from a Von Newmann architecture can have a drastic effect in data appearance since there is the necessity for at least one bit of tag for each data item is not referenced through a descriptor. Even this, however, is an addressing problem and does not directly affect the data transformations or their basic representation.

4.2.3.2 Data Addressing. The method used for data addressing is extremely important if one is to have an efficient HOLM implementation. The addressing of data takes up the majority of bits in the instruction stream in most architectures. Thus, improvement in data addressing compactification can provide to be a dramatic total space savings. But again, it is to be noted that how one addresses data is basically independent of data representation and the various data transformation operators. The data addressing does not of necessity interact with the control flow addressing, although they are usually combined since a given instruction architecture tends to have but one addressing methodology.

Questions as to whether the architecture supports one or two dimensional addresses; whether addresses are of a lexical level-displacement or base-displacement form; whether a single accumulator general register set, or stack exist; or whether absolute, indirect, sectored or banked addressing exists; do not prevent the implementation of the language. The question again is one of efficiency and design cleanliness.

4.2.3.3 Functional Transformations. From examining a higher order language such as HAL/S, it is readily seen that one could change, add or delete, the set of operators that perform the data transformations. One could have the exact same set of control instruction and yet remove arithmetic operators and provide list structure type manipulation. Similarly, how one addresses the data, or the exact details of the data representation, do not overly affect the concept of add, multiply, etc.

The actual implementation of a given transformation operator will, of course, be dependent upon the actual data representation. But, during implementation, this is isolated into a subroutine. That is, when the "operator" is decoded, control is transferred to the appropriate micro code routine to perform the semantics, e.g. add. Thus, while the routine may change, it does not effect the overall structure of the micro-program implementation.

4.2.3.4 Data Representation. It is obvious that the detailed data representation is basically independent of the other three areas. Indeed, often the data size, precision and range are not even defined within the higher order language other than by some vague concept such as SINGLE and DOUBLE precision.

This then is an area that can easily be modified in design for purposes of comparison of hardware restriction.

Whether an integer is represented in binary by sign magnitude, one's complement, two's complement, or is 16 bits, 24 bits, etc.; or even represented in a decimal format, the value has an identical interpretation. Three plus four is still seven: Add has a definite meaning.

4.2.3.5 Advantages of Separation of Issues. By understanding that these basic implementation issues are separable, it is possible to investigate the effects of modifying one particular area. Also, it then becomes possible to perform meaningful comparisons with other architectures, where for example, the data format is already specified. Also, when viewed in this manner, it becomes clear that the emphasis for improvement falls upon the addressing structure both for control flow and data addressing.

4.3 Addressing

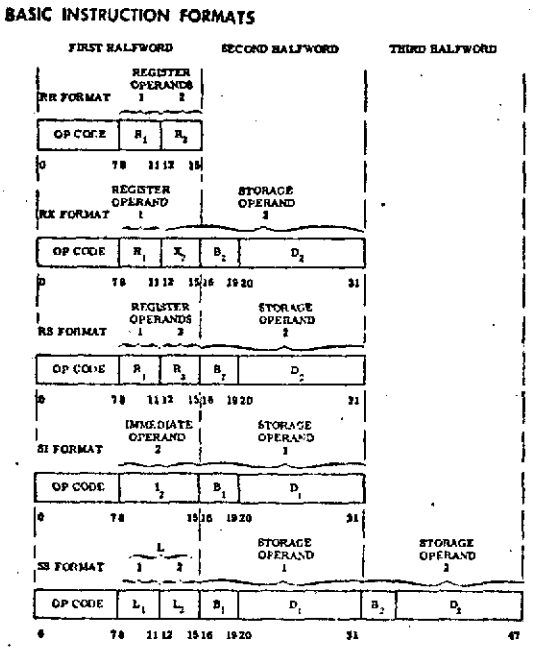
The importance of addressing in an efficient instruction architecture design cannot be over emphasized. The memory used for program code is dominated by the addressing fields. The efficiency of execution of a HOL program depends upon a clean implementation of the addressing structure. This section will discuss the importance of addressing, the requirements which are placed upon any addressing design for HAL/S implementation, an ideal encoding of this information, and the requirements for HAL/S usage statistics in order to form the basis for the proper encoding. A large amount of time was spent on this task during the study. While the method of optimal address encoding is clear when thought about (Section 4.3.3), this is but one aspect of the HALM development. More fundamental is the process of trying to understand the addressing options available in the aerospace environment and their effect upon execution efficiency. While various avenues were investigated, any conclusion must be reached when HAL/S user statistics become available.

4.3.1 Importance of Addressing

When trying to compare instruction architectures it is very useful to separate the data space requirements (D) from the program code requirements (P). The total memory requirement (M) being the sum of the two.

$$M = P + D$$

Regardless of the instruction architecture, a first approximation is to assume that the data representation must remain similar. While this is not always true, and indeed integer arrays may be greatly improved upon, the bit size of arithmetic data and character representations are usually based upon outside requirements, such as required precision. It is therefore in the program code where most of the memory savings must be found. In the aerospace industry, this is particularly evident since program memory requirements are often two, three or more times as large as that of the data memory requirements.



Format	Number of Bits	Operator		Operands	
		bits	percentage	bits	percentage
RR	16	8	50%	8	50%
RX	32	8	25%	24	75%
RS	32	8	25%	24	75%
SI*	32	8	25%	24	75%
SS*†	48	8	16 2/3%	40	83 1/3%

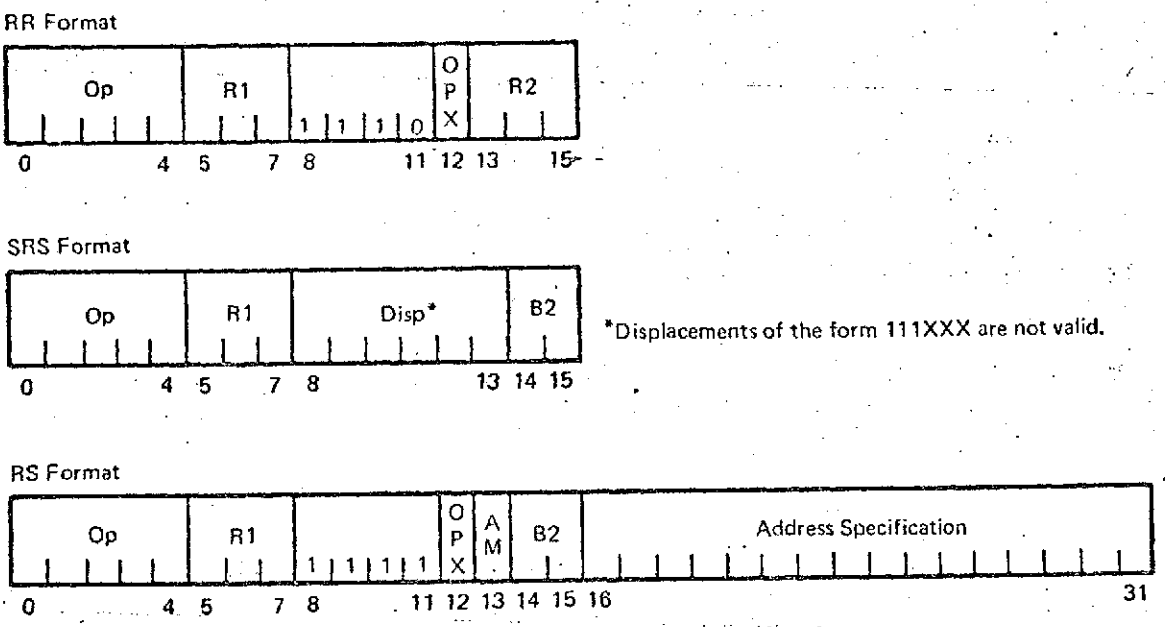
* Two memory operands present:

format	bits per operand	percentage per operand
SI	12	37 1/2%
SS	20	41 2/3%

† Length fields could possibly be considered as part of operator.

IBM 360 Instruction Formats

Figure 4.3.1-1



Format	Number of Bits	Operator		Operand	
		bits	percentage	bits	percentage
RR	16	10	62.5%	6	37.5%
SRS*	16	5	31.25%	11	68.75%
RS	32	10	31.25%	22	68.75%

* Not quite 11 bits, since only 56 displacements used.

Actually: $\log_2 56 = 5.8$; thus, $5.8 + 5 = 10.8$ bits.

IBM AP-101 Instruction Format

Figure 4.3.1-2

CUBES

HAL/S PROGRAM EXAMPLE RESULTS

(refer to Chapter 4, Appendix 1)

IBM 360 encoding:

	Total	Address	Opcode
Number of Bits	2800	2144	656
Percentage of Total	100%	76.5%	23.5%

AP-101 encoding:

	Total	Address	Opcode
Number of Bits	1888	1298	590
Percentage of Total	100%	68.7%	31.3%

AP-101 encoding compared to the IBM 360 encoding:

	Comparing Totals	Comparing Address	Comparing Opcodes
Reduction in Bits	912	846	66
Reduction Percentage	32.6%	39.6%	10.1%
Relative Size of AP-101	67.4%	60.4%	89.9%

Bit Savings compared to Total program size:

	Total Savings	Address Savings	Opcodes Savings
Fraction	912/2800	846/2800	66/2800
Percentage	32.6%	30.2%	2.4%

Figure 4.3.1-3

An investigation of the program memory usage itself leads to the differentiation between operators and operands. That is, the opcode fields (O) versus the addressing fields (A).

$$P = O + A$$

An examination of the IBM 360 instruction formats (Figure 4.3.1-1) show for the RX, RS and RI formats the opcode field constitutes only 25% of instruction, while the address field(s) contains the remaining 75%. [Qualifications of understanding can be made: the register field is a second operand, but will be here considered as an "implied" operand; indexing is here considered part of the operand specification rather than as an operator]. It is easily seen therefore, a savings in the address field representation can easily have a large impact upon total memory savings.

The IBM AP-101 instruction architecture design recognized this large bit representation dedicated to addressing. Its instruction compactification, to a large degree, depends upon having a short memory reference form. Assuming that these instructions are used with a high frequency, the total memory requirements for a program can be appropriately reduced. Figure 4.3.1-2 shows the AP-101 formats along with operator and operand break down. Even here, the addressing information dominates, except for the RR instructions.

Appendix 1 contains a HAL/S program along with both the IBM 360 and AP-101 code which is generated for it. Figure 4.3.1-3 summarizes the results of this example. There is a substantial reduction of the program size from 2800 bits for the IBM 360 down to 1888 bits for the AP-101. But when this is examined in detail, it is seen that while the addressing bits were reduced by 846 bits, or 39.6%, the opcodes were only compactified by 66 bits, or 10.1%. Even this reduction of the opcode fields is not reflective in the total program size reduction. Since the opcode fields formed but a small percentage of the bit space initially, its contribution to the resultant code compactification is only 2.4%! Sixty six bits out of the total 2800. Of the total savings of 32.6%, the address field

reduction contributed 30.2%. The reason, of course, is that even in the AP-101, the address portion of the program is 68.7% of the total while the opcode portion is only 31.3%. Even a small reduction in the address field size has a large effect upon the total reduction.

While the AP-101, for example, has been able to reduce the address portion of the instruction from the IBM 360's 76.5% down to 68.7%, it is still apparent that addressing considerations dominate. Indeed, it is very easy, given user statistics, to Huffman encode the opcode fields very efficiently, but addressing must also reflect a spectrum of capabilities. It should not be so tailored to a particular set of programs or users that is becomes inefficient in other cases. Because of this dominance of addressing in efficiency considerations, considerable time was spent investigating and analyzing various methods of address implementation in instruction architectures; the requirements that HOLs, HAL/S in particular, place upon address mechanisms if they are to be efficient; and the actual encoding techniques available for optimal encoding once the addressing methodology has been choosen.

4.3.2 Data Addressing

The forms of addressing required to access data referenced in a HOL (HAL/S) can be examined from several different points of view.

- Super compilation data (COMPOOL) versus compiled data
- Statically declared versus dynamically declared
- Formal parameters versus declared data
- Name (address) reference versus value fetch
- Name scope properties (locality) versus homogeneous treatment of data
- Formal parameters versus scopped in data versus locally declared data

Each of these different attitudes indicate the various characteristics of data which must be resolved in the code generated by a HOL. In order to create an efficient machine for the execution of a HOL, the semantics of the language must be considered along with a model as to the actual usage of the language. That is, while an instruction architecture such as the IBM 360 is capable of implementing an(y) HOL, it is in general very inefficient in doing so. While the semantics of the particular HOL can be implemented, both the instruction architecture of the IBM 360 itself and the lack of a model of the proposed language usage, cause inefficient implementation of the language.

4.3.2.1 Super Compilation Data versus Compiled Data. Data at a COMMON or COMPOOL level exist outside of any given single compilation. The data is to be referenced (by definition) by several different compiled units. It is only at link edit or actual run time that the environment of the generated code is known. It is only after the COMPOOL has been "linked" to the compiled unit(s) that actual referencing (addressing) of data is completely known. From a pragmatic point of view, this has two major implications for an instruction architecture. The first is that the Compool data cannot be massaged by the compiler. It cannot be sorted by size or frequency of reference or homo-

geneously addressed as other compiled data might be. Since the Compool is to be referenced by multiple compilations, its data must be referenced by each compilation in a set standard way. The second implication is that the actual physical addressing of a Compool cannot be "complete" until either a link edit step or at run time. While the Compool is logically addressible, its actual memory residency is not known. Thus, for example, the Compool could be considered to have, of necessity, a different memory "relocation factor", than does that data which was known in the compilation step.

4.3.2.2 Statically Declared versus Dynamically Declared. Ideally, storage allocation would follow the semantics of language definition both explicitly (static versus automatic) and implicitly (the life duration of a process or procedure). Pragmatically, the data storage policy is often otherwise. In aerospace applications, it is often useful to have information be truly static regardless of how it is declared. This can facilitate both hybrid simulation and testing, and provide a "down link" capability for further analysis. However, if procedures are either reëntrant or recursive, this is not often viable,and true automatic storage need be provided.

Another static/dynamic question involves the question as to when data memory is allocated to a process. This question also involves the decision as to how storage is allocated. That is, is it allocated as a single contiguous block (region), or as several blocks. One model of data memory allocation that can be assumed is that which is similar to the Space Shuttle model. All data which is to be allocated statically will reside in one contiguous block, which can be assumed (if necessary) to be resolved at linkage edit time. All data which is dynamically allocated will be from another contiguous block (stack area) which may be allocated at run time. Thus, there appears three blocks of data to be addressed by a program: the COMPOOL, the static data, and the dynamic data.

4.3.2.3 Formal Parameters versus Declared Data. The actual data reference for declared data can be resolved during compilation. The reference for a formal parameter, by definition, is more complex. While it is possible to address the "formal parameter", the data must either be placed in this known address or else must be obtained via an indirection step.

While declared data can be either static or dynamic (and in the Space Shuttle model the static appears to predominate), formal parameters by their nature are dynamic, and need exist only when the procedure in which they are used is actually called.

The addressing structure of the instruction architecture should then be able to handle both static addressing and dynamic addressing. The static addressing would be used for both Compool data and the majority of compiled data, while the dynamic addressing would be used for formal parameters and dynamic data of reëntrant or recursive procedures.

4.3.2.4 Name Reference versus Value Fetch. An instruction architecture must be able to both generate the address of an operand and also be able to fetch the value of the operand. When a "call by value" occurs, as with a formal parameter, the value must indeed be passed and sent rather than an address, or else side effects of the change of value could occur. Similarly, if a "call by reference" of a formal parameter occurs, an address must be passed in order to assure that the value of the parameter changes as the variable changes, and also in order that the formal parameter itself can be assigned into.

Besides their use in formal parameter passage, addresses must also be able to be generated if the instruction architecture separates "operators" from "operands". This occurs in standard stack organized machines (e.g. B6700, and baseline MP) where the store operator needs the address of the operand to be stored into to reside on the stack. Of course, this usage could be circumvented if the stack operator (and any other memory changing operator: set bit, move, ...) were allowed to be incorporated into the "operand" versus "operator" class. This is not as strange as it might first sound, since the "operand" class is itself standardly a load stack and/or load address to stack operator.

4.3.2.5 Name Scope Properties versus Homogeneous Treatment of Data. Instruction architectures which are developed from a HOL often use the name scope properties of the HOL to develop the addressing structure of the machine. The pragmatic result is an immense saving in memory requirements by the efficient compactification of the required address field. This result comes from two phenomena. One reason is that name scopes (lexical levels) inherently narrow the amount of data that need be addressed in any given instant. The name scope forms a static tree and identifies that data which can be seen by the instruction. Only that information which is in the name scope can be referenced, by definition of the HOL. Hence, only that amount of data (information) need be addressible. This greatly reduces the number of bits needed to address the allowable data. In conventional Von Neumann architectures, all of memory is addressed (although often partially compacted by a static two dimensional address - base and displacement - as in the IBM 360).

The second reason is that instruction architectures developed from a HOL recognize that they only need to address variables, e.g. integers, scalars, vectors, matrices, bit and character strings, arrays, and structures. They do not have to explicitly address each element of a vector, matrix, array, ... Hence, the number of entities which must be addressed are simply the number of names, of variables, which appear in the program. While a Von Neumann architecture would have a large enough address field to address each element of a 100 element array, a HOLM would need only reference the array itself. To reach the i^{th} element of this 100 element array, an index operation is performed.

The reason then for the savings of address field size when a HOLM is developed is that of the "locality" of the appearance of the possible data is taken into condieration. The address field can be compacted by both considering name scope rules (hence HOL self imposed data referencing restrictions) and by directly addressing only the HOL named data and not elements of the data.

4.3.2.6 Formal Parameters versus Scopped In Data versus Locally Declared Data. From another point of view, a HOL procedure must be able to reference from three distinct sources. One source following name scope rules is scoped in from other procedures. Formal parameters are passed into the procedure and can either be known as values or must be referenced indirectly to the indicated data. Locally declared data must either be created upon entry or else be static throughout the life of the process.

In the use of most name scoped HOLs, local data is "created" upon entry to the procedure and thus requires a dynamic characteristic similar to formal parameters. Indeed, this implies that scoped in data has been, in general, so created from a previous outer level procedure. In this case static data which is to exist throughout the process could be handled by moving it physically (addressability) to the outer most level of the process to the program level.

The Space Shuttle, however, uses the other assumption: most data is to be considered static and only exceptionally will it be dynamic (formal parameters, local data of reëntrant or recursive procedures). This then would imply that if the addressing scheme is to be efficient with the Space Shuttle model, and hence make use of "locality", this static data can not be simply moved in the program level, but rather the standard lexical level referencing must be able to reference both static local data and dynamic local data (formal parameters and reëntrant local data).

4.3.2.7 Solutions to Addressing. One major motivation for a HOLM design is to be efficient. No matter what the form of addressing available for an instruction architecture, it must be able to support the various HOL addressing modes indicated in Sections 4.3.2.1 to 4.3.2.6. Indeed, all (almost) addressing methodologies do have solutions since such languages as Fortran, Algol, and Cobol can be implemented with them. The question therefore turns rather on efficiency: the minimizations of addressing space requirements. As Section 4.3.2.5 indicated, the advantage of the lexical level-displacement form of addressing is in fact that it minimizes the space of variables which must be spanned. The size, therefore, of a sufficient displacement field can be more compact than if all memory had to be addressed. That is, it makes use of:

1) Name Locality.

Associated with this concept is the use of descriptors. This then allows the limitation of the addressing to only the number of variables declared, and is not dependent upon their size. Thus, an array of a hundred elements counts as but one entity for instruction addressing. This again reduces the requirements on the size of the displacement field, thus:

2) reference only declared entities.

If these two features are examined, it is seen their saving is a result of the reduction in the address field width. The conclusion to be drawn is that any form of addressing which can reasonably support the addressing requirements of a HOL (Section 4.3.2.1 through 4.3.2.6) is sufficient if it can be made efficient by the reduction of the address field width. In order to do this intelligently it is necessary to have very explicit statistics of actual programmer usage. While one may have to support a possibly large address space, if the majority of the time only 16 or 32 entities need be addressed, then this would only require for the majority of cases only $\log_2 32$ or 5 bits worth of information. This of course depends exactly on how these variables are distributed across the classes of data described in Section 4.3.2.1 through 4.3.2.6. Section 4.3.4 will indicate the forms of statistics on addressing that should be acquired by HAL/S programs in order that a tailored efficient addressing structure can be designated.

4.3.3 Addressing with the B1700

The baseline MP architecture for HAL/S was predicated upon the use of a byte oriented microprocessor. For efficiency reasons, it was assumed that an access to an 8 bit byte was optimal and that there was no advantage, because of hardware restrictions, for entities of a non 8 bit multiple. However, the use of the B1700 opens another set of possibilities. The B1700 has been designed (refer to Section 4.4.3.4) to have bit addressible memories with access width of varying sizes. This has been accomplished without paying an execution penalty for any quantums of 24 bits (i.e. 1-24 width field each requires the same access time; 25-48, etc. ...). This possibility of efficient bit addressing therefore opens the way for more efficient encoding. No longer are bytes sacred and both instruction and data built upon these units. The baseline MP instruction architecture consists of a majority of 8 bit "operators" and 16 bit "operands"; data is in multiples of 8 bits with the basic arithmetic types being of 32 and 64 bits width; and stack usage was predicated upon 64 bit quantums both for special words and descriptors. With the B1700 it is possible to actually have 3 bit or 7 bit operators without paying an efficiency penalty. Indeed, such encoding is as quick as 8 bit encoding, yet can be spacially more efficient.

The paper "Burroughs B1700 Memory Utilization", by W.T. Wilner, [Wi 72b], presents the results of Burrough's own success in developing implementation for their Fortran, Cobol, RPG and SDL (system development language) on the B1700. These results can be summarized in Figure 4.3.3-1. Figure 4.3.3-2 reports similar results from another paper by Wilner [Wi 72a]. These results appear to be dramatic, and indeed they are. The results come from properly encoding the information of the respective higher order language under actual user statistics. The best example presented by Wilner was with respect to SDL. Since Burroughs is the sole user of this language, the accurate sample (namely all that exists) of its usage was available.

B1700 Comparisons

	Other System	Percent Program Memory Reduction	Percent Faster Execution Speed Comparison
FORTRAN	System/360	50%	-
FORTRAN	B3500	40%	-
RPG	System/3	50%	25% to 5%
COBOL	System/360 Mod 30	70%	60%

Burrough's Encoding Comparisons for the B1700

Figure 4.3.3-1 [Wi 72b, p. 585]

Language of Sample	Aggregate Size on B1700	Aggregate Size on Other	Other System	Percent Improved B1700 Utilization
FORTRAN	280KB	560KB	System/360	50
FORTRAN	280KB	450KB	B3500	40
COBOL	450KB	1200KB	B3500	60
COBOL	450KB	1490KB	System/360	70
RPG II	150KB	310KB	System/3	50

Amount of Program Compaction on B1700

Figure 4.3.3-2 [Wi 72a, p. 495]

Encoding Method	Total Bits for MCP's Opcodes	Utilization Improvement	Decoding Penalty	Redundancy
Huffman	172,346	43%	17.2%	.0059
SDL 4-6-10	184,966	39%	2.6%	.0196
8-bit field	301,248	0%	0 %	.4313

Comparison of SDL Opcode Encoding
Against Extreme Methods

Figure 4.3.3-3 [Wi 72b, p. 581]

Figure 4.3.3-3 represents the chart Wilner presents showing a comparison between an 8 bit opcode encoding of SDL versus the ideal Huffman encoding versus the method which they adopted. This method was to have three opcode sizes: 4, 6, and 10 bits in width. What is interesting to note is how close their choosen encoding approaches the ideal case, and yet how much it saves from the 8 bit encoding. Besides the opcode encoding, the SDL B1700 implementation provides for both flow control addressing and data addressing. Flow control addressing is a triple as follows:

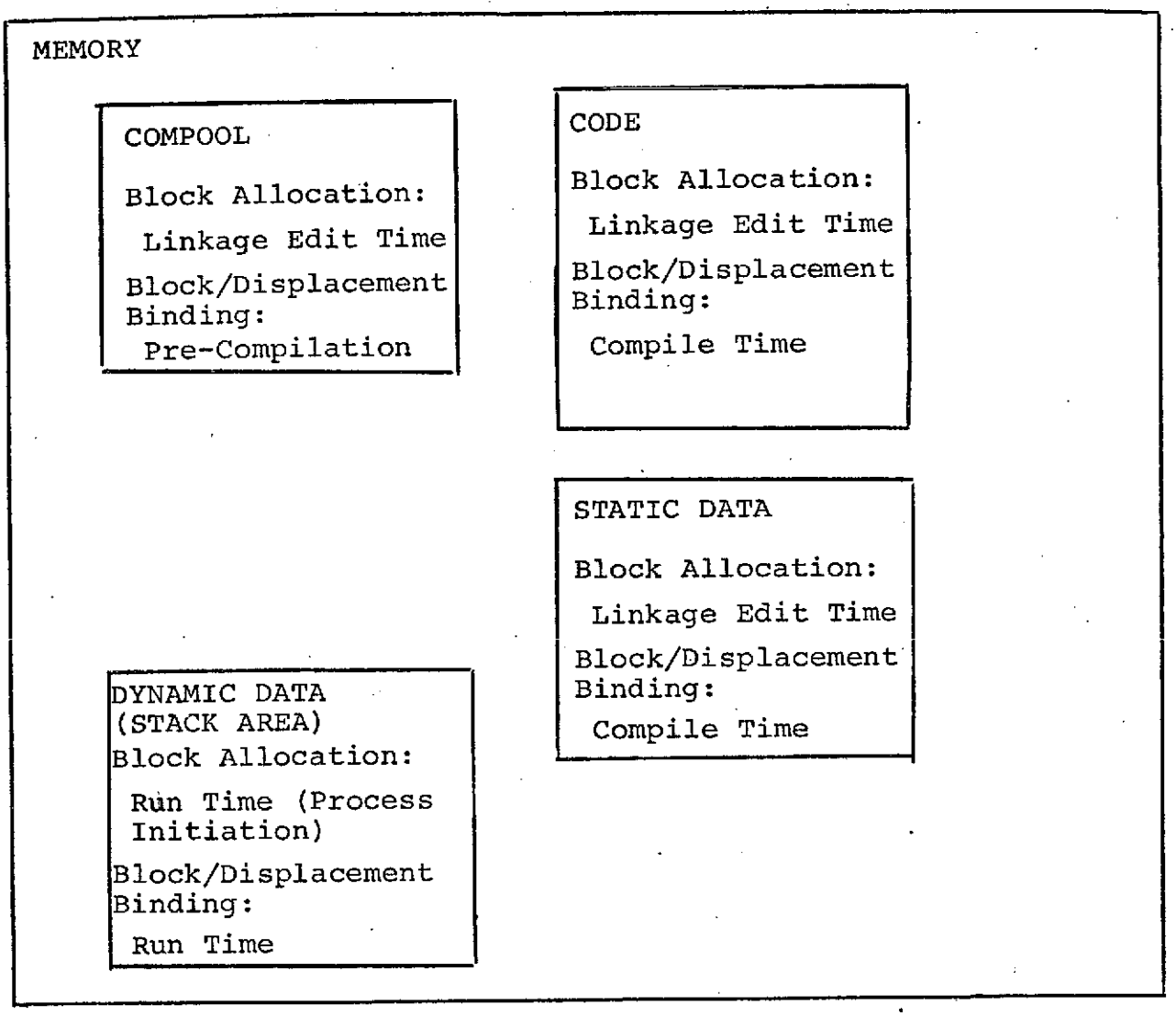
field description	segment name	displacement
3 bits	0, 5 or 10 bits	0,12,16 or 20 bits

where the field description indicates which of the eight allowable addressing possibilities is present. The data addressing is also a triple but of the form as follows:

field description	lexical level	displacement
2 bits	1 or 4 bits	5 or 10 bits

where the field descriptor indicates which of the four formats is involved.

While the width of these addressing structures produce "operands" of varying length, the data addressing can be either 8, 11, 13 or 16 bits in length, while the control addressing can possibly vary between 12 bits and 33 bits.



Block Allocation:

Memory management assumes memory has been assigned by the linkage editor with the possible exception of the stack area. Overlays to be handled statically and resolved by the linkage editor.

Block Displacement:

The displacement relative to the start of the block is known except in the case of the stack area. In the case of the stack, displacements are relative to a mark stack word (i.e. procedure level usage).

Memory Usage Model

Compiler Generated Blocks and Data Referencing

Figure 4.3.4-1

The methodology applied in the reduction of opcode fields, and the control flow and data addressing is straight forward and produces a near optimal result. But in order to accomplish this in a realistic way, it is necessary to obtain real user statistics of both opcode appearances (HAL/S language usage) and information as to the distribution of the referenced operands.

Unfortunately, it was not possible during the period of this study to gather meaningful HAL/S usage statistics since HAL/S had only begun to be used in the Space Shuttle program.

4.3.4 Useful HAL/S Statistics

Gathering statistics for HALM development, not only allows for optimal encoding of operators and operands, but it can also make possible an understanding of what forms of operands need be cleanly supported. While lexical level-displacement addressing follows the name scope rules of a block structured language, it is not the most efficient method when parameters outside the current name scope are to be passed. Similarly, the aerospace environment often requires a more static environment than is implicit with a stack organization. This too can cause inefficiencies in lexical level-displacement addressing, forcing a disproportionate number of variables to the higher lexical levels. This of course then requires a large displacement field at these levels. Named Compools also can provide addressing problems for lexical level-displacement addressing. While a single Compool can be easily handled by allocating a single high lexical level for its addressing, multiple Compools demand multiple addressing capabilities, and hence resolution. In aerospace usage, there is also the possibility that the sizes required are smaller than is a more general earth bound environment. From these considerations, it is apparent that good distributional statistics of actual address usage, not only can provide for efficient encoding, but will also perhaps indicate another appropriate form of addressing. The requirements for addressing discussed in Section 4.3.2 must be fulfilled, but is a minimimum, efficiency is to be found in compactification of the result addressing fields. It would be hoped, therefore, that patterns of locality of operands would be detected in the statistical distributions. Figures 4.3.4-1 displays one possible model for code and data blocks generated by HAL/S in an aerospace environment such as the Space Shuttle. From this diagram, it is seen that even here there is localization of addressing requirements to very specific blocks.

Assuming a lexical level-displacement form of addressing is a real possibility for implementation, it is necessary to know:

- The number of procedures at each lexical level that define n variables.
- The number of procedures at each relative lexical level that define n variables.

In the HAL/S environment it is useful to know:

- The number of Compools with n defined variables referenced at each lexical level.

In the aerospace environment it is of interest to know:

- The distribution of procedures with respect to the number of locally declared dynamic variables, the number of locally declared static variables, and the number of formal parameters.
- The distribution of variable references with respect to their lexical level (or Compool) definition for each procedure at level n.

To develop a reasonable control flow addressing structure, information of the following nature should be obtained.

- The number of programs expected to be in the system at any given time.
- The distribution of the number of procedures per program.
- The number of tasks within a program that can be expected.

It is expected that with the progress of the Space Shuttle program, these statistics will become available. This will allow for both a near optimal encoding of HAL/S, and for further investigation into the addressing possibilities open to aerospace applications.

4.4 Micro-Processors

The level of design immediately below the instruction architecture is that of the processor that will implement the instruction architecture. The development of micro-processors and their availability for use, has allowed the tailoring and development of varying instruction architectures. These in turn have aided in the development of appropriately designed micro-processors. This section will first give a brief history of the development of the concept of micro-programming. This will facilitate an understanding of why micro-processors tend to differ so dramatically from each other and the motivation for their design. Next, several important issues for micro-processor design will be discussed along with their relevancy for higher order language emmulation during the instruction architecture development stage. Finally, several specific micro-processors will be examined and the reasons for the selection of the Burrough's B1700 indicated.

4.4.1 History of Micro-Programming

A lot of confusion and difference of opinion regarding micro-programming arises because each author and corporation uses this term in their own manner with their own connotations. In the literature on micro-programming, there are at least four different attitudes and hence four different connotations in using the term micro-programming. The four divergent views of micro-programming can be classified as follows:

1) clean systematic hardware design;

2) computer manufacture cost savings with a "family" of systems;

3) "User" being able to save "old" software via compatibility and tailoring of the system to his needs; and

4) special requirements such as teaching and research, and associated cost savings in singular developments such as found in the aerospace industry.

These will each be briefly discussed in turn.

4.4.1.1 Systematic Hardware Design. Historically, micro-programming was a term coined by M.V. Wilkes in 1951 [Wi 51]. He states:

> "My object was to provide a systematic alternative to the usual somewhat ad hoc procedure used for designing the control system of a digital computer. The execution of an instruction involves a sequence of transfers of information from one register in the processor to another; some of these transfers take place directly and some through an adder or other logical circuit. I likened the execution of these individual steps in a machine instruction to the execution of the individual instructions in a program. Hence the term micro-programming. Each step is called for by a micro-instruction and the complete set of micro-instructions constitutes the micro-program. The analogy is made more complete by the fact that some of the micro-instructions are conditional." [Wi 69a]

The term "micro-programming" used in this way applies only as a hardware concept. It is a "method" of logical design which has all the advantages of modular development for complex systems. Many authors who are hardware oriented (refer to [Va 71]) still tend to regard this as its main value, while recognizing others.

4.4.1.2 Manufacture Cost Savings. Large companies find that micro-programming is a means to provide system compatability over a wide range of performance and cost. The IBM 360 series of computers is able to have even its smallest computers have the same "power" as its large brothers since they can be encoded via micro-programming. S.S. Husson's book, "Microprogramming: Principles and Practices", [Hu 70] is representative of this attitude. In this book (pp. 72-74), he discusses seven advantages with the use of micro-programming.

1) flexibility and tailoring;

2) changeability;

3) ease of designing, maintaining and checking;

4) uniformity of design;

5) ease of education;

6) micro-programming can extend the useful life of the system; and

7) economy.

In discussing each of these, the emphasis is from the system design point of view, that of the manufacturer. He states (pp. 16-19):

> "We have seen that microprogramming offers many advantages over a conventional hardware control in factors such as cost, performance, flexibility and tailorability, ease of maintenance, and many others which will be reviewed in more detail in a later chapter. Yet in reality, except for few isolated cases, microprogramming remains in the domain of the design engineer. Why? What is holding the different interested disciplines from taking advantage of the flexibility and efficiency microprogramming can offer? The following is a partial list of observations on this question.
>
> 1. Microprogramming was not intended for the novice programmer. ...
>
> 2. Except for few special system designs, the control programs are stored in read-only storage devices that are difficult and expensive to modify. ...

3. The lack of standard assembly language and standard micro-orders and micro-instructions discourages the users from attempting to apply micro-programming. ...

4. A fourth major problem is the lack of sufficient educational effort in preparing the potential user to cope with the problems he is to be confronted with in instructing him of the available means for solving them, and in acquainting him with the advantages and disadvantages of this additional option for any given class of problems. Basically, micro-programming has been treated as an adjunct to machine design; no particular effort has been made to separate the related information or to make micro-programming itself convenient. Clearly, such a responsibility does not all fall on the designer.

5. ....

6. A sixth reason for the lack of widespread usage of the micro-programming option is the manufacturer's concern for the preservation of the architectural identity of the system and with preserving its effectiveness and its compatibility with other models in the product line.

   This problem becomes a simple one if the system's original identity or affiliation with any product line or any operating system is not needed, that is, if the system is to become completely and permanently a slave to one fixed mission or task. ..."

From a manufacturer's point of view, the emphasis upon system compatability and orderly growth becomes paramount. This is both obvious and necessary, since they wish to market their product on a mass scale.

4.4.1.3 Maintenance of Old Software. The same desires of compatability and orderly growth is expressed from the user's view but with a different emphasis. This point of view is well expressed in the birth of the Standard Computer Corporation and developed in a paper by its Vice President, L.L. Rakoczi [Ra 69].

The user of computer systems has a financial investment and therefore a real desire to maintain the working set of programs that he already has. When new computer systems are bought, the expense of changing the current programs to make them compatible with the new system can be prohibitive. It is widely known that those programs written in a HOL for portability reasons, seldom are truly transferable, and programs written in an assembly language are usually given up as a hopeless loss.

IBM has recognized this form of problem by allowing its small 360's to have a special 1401 emmulation mode in which the 360 "looks like" a 1401, and hence the old 1401 programs can be run while "the new and improved programs" can be written for the 360. The user, however, is not necessarily interested in just one computer manufacturer, but wants to be able to salvage his software from any computer. By emmulating the "old" machine while taking advantage of the new, he obtains the best of both worlds.

> "(the user) often finds it costly and time consuming to rewrite his proven and useful programs so that they will run on the new generation computer. A related problem is faced by users of large scale computer installations who have a number of computer systems. These computer systems frequently have different machine-language repertoires which are not compatible with each other. In other words, a program written for one computer system of the user will not perform on another computer system of the same user."

> "any fourth generation machine can be 'dedicated', not in one direction, but in many. Vintage software, massaged and made workable through frequent use and long study, can now be employed as required without locking the user in or out."
>
> " The fourth generation computer will save training, sales and service costs for its maker and will permit its user to call on an infinite variety of industry resources and know-how for the execution of his functions and the solution of his problems." [Ra 69]

Besides this general gain which all users can hope to obtain, the micro-programmed computer can be of assistance in another way.

> "for their part, fourth generation thinkers were planning to combine micro-programming with some form of inner computer solely to execute subroutines. Then they started adding features to increase micro-programming efficiency."

Commonly executed subroutines can be made into executed micro-code. If sine or cosine, for example, are used repeatedly they could be implemented as an instruction. Similarly, when common table search and date lookup routines are the main occupation in a commercial application, they become measurable bottlenecks which can be opened by making their execution efficient. These two features then are of the utmost concern to the user:

a) To make all his current software available to him as he goes to the new (or different) computer systems. This also would allow access to any software available from any source; and

b) the possibility of tailoring the computer to particular needs when identifiable bottlenecks in subroutine execution can be located.

4.4.1.4 Special Singular Users. The fourth attitude is rather limited by its very nature of being specialized. There are applications where the three previous economic incentives meant generalization. Designing systematically with a general method means ultimiate savings. Building a compatible system across a spectrum of price range means economic savings in developmental cost and a market base for growth. Being able to save current software and being able to use any other software in existence saves rewriting and much developmental cost.

In the university environment, one runs into the needs for education and research. These both have their special requirements [Ro 71]. Micro-programming becomes both a teaching tool to train people in system architecture and a device for research to expand the frontiers of knowledge.

The use of micro-programmed machines for aerospace applications is another example of special usage. Patzer, et al. [Pa 70] states:

> "Attention is focused upon three systems engineering considerations:
>
> (a) Specialized Operations - A micro-programmed computer organization is shown to be well suited to applications where very specialized tasks require a significant percentage of total execution time.
>
> (b) Restructurable Architecture - The case with which the computer instruction set, data representation, interrupt system, and input/output system can be restructured via micro-programming is shown to be a significant consideration.

(c) Efficient Simulation - The unique capability for efficient simulation (emulation) inherent in micro-programmed computers is shown to permit a significant reduction in development time and overall cost when a previous system is upgraded or an experimental system is used."

and

" Two costs are relevant to the aerospace systems implications of micro-programmed computers. The first is the cost of the computer itself; the second is total system cost. The former includes electrical and logic design, packaging, drawing release, tooling and qualification and environmental testing of the computer. The latter includes the cost of the computer, its peripheral devices, other system components, software, operating costs and any costs assigned to intangibles. For a specific aerospace system application, the cost of a micro-program controlled computer by itself may or may not be less than that of an alternative computer. However, the system engineer's freedom to modify computer characteristics without major hardware redesign, repackaging or requalification and his ability to extend system life by micro-program changes may lower overall system cost. This freedom can often allow later incorporation of a new weapon system, navigation aid or mode of operation. System cost analysis for each application must quantitatively account for such factors qualitatively discussed herein."

4.4.1.5 Current Micro-Programming Usage. It is as a direct out growth of research by both the Universities and by the aerospace industry that an emphasis upon higher order language machines has occurred.

For the universities, this has been an emphasis upon research in developing new instruction architectures and improving programming practices. The aerospace industry has been highly interested in the compactification of memory in order to reduce computer cost, weight, power consumption, and physical size.

While hardware designers, industry, and large software users still maintain their particular orientations, the usefulness and capabilities of micro-processors in HOL execution has become a major area of investigation. It is with respect to this attitude, concern for HOL implementation, that the various microprocessors have been examined in this study. The ability of a micro-processor to implement a HAL machine is the criteria by which they were judged.

Field	ROS Bits	Function of Field
	0	Parity of bits 0-30
LU	1-3	Left input to mover
MV	4-5	Right input to mover
ZP	6-11	Bits 0-5 of next ROS address
ZF	12-15	Source of bits 6-9 of the next ROS address
ZN	16-18	ROS addressing mode
TR	19-23	Destination of adder latch contents
	24	Spare
WS	25-27	Source of local storage address
SF	28-30	Local storage function
	31	Parity of bits 32-55
IV	23-24	Invalid digit test and instruction address register control
AL	35-39	Shift control and gating into adder latch
WM	40-43	Mover destination
UP	44-45	Byte counter function
MD	46	MD counter control

Field	ROS Bits	Function of Field
LB	47	LB counter control
MB	48	MB counter control
DG	49-51	Length counter and carry insertion control
UL	52-53	Mover function -- left digit
UR	54-55	Mover function -- right digit
	56	Parity of bits 57-89
CE	57-60	Emit field (used as data)
LX	61-63	Left input to adder
TC	64	True/complement control of left adder input
RY	65-67	Right input to adder
AD	68-71	Adder function
AB	72-77	Condition branch test A (furnishes bit 10 of next ROS address)
BB	78-82	Condition branch test B (furnishes bit 11 of next ROS address)
	83	Spare
SS	84-89	Stat setting and miscellaneous control

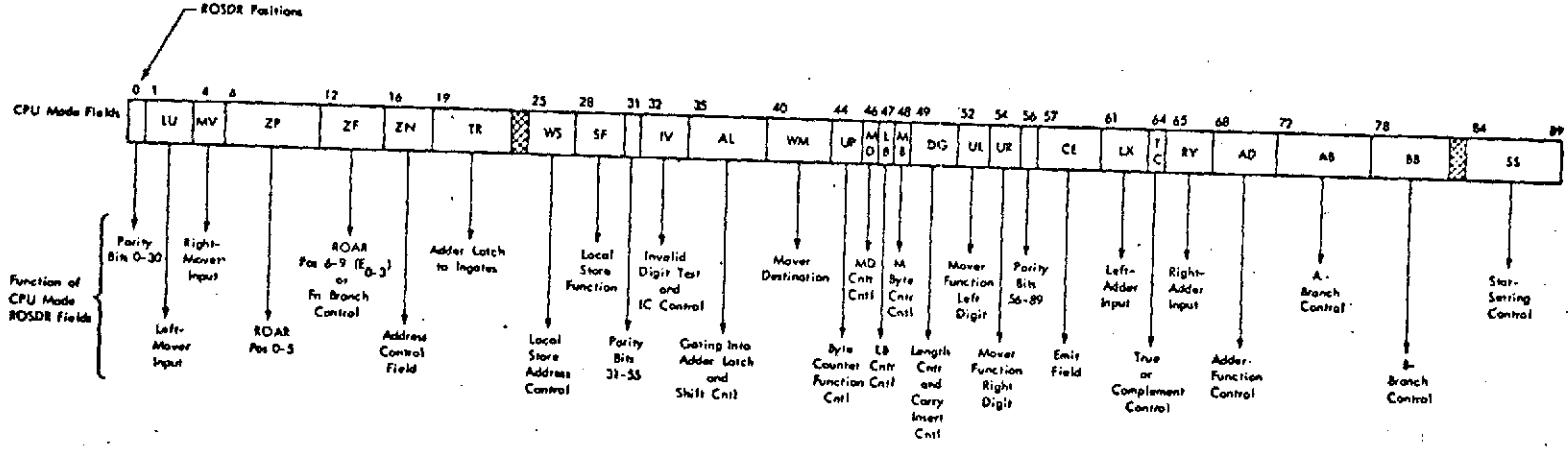


Figure 8.14 Microinstruction format—CPU mode.

Figure 4.4.2-1

Micro-Instruction Formatting for IBM 2050 [Hu 70, p. 322]

4.4.2 Important Micro-Processor Design Issues

There have been a large number of micro-processors designed and developed in recent years. They vary in their internal bussing, computational capability, data width, and methodology of micro-instruction encoding. In order to appreciate their basic differences and associated advantages with respect to higher order language implementation, it is necessary to discuss certain of these differences.

4.4.2.1 Horizontal Versus Vertical Micro Encoding. Micro-instructions are used to control the execution of the processor. It does this by specifying at the adder, shifter, and register level both the inter-connections between these elements and the function which the active elements are to perform. How this information of inter-connections and functional specification is encoded differentiates "horizontal" versus "vertical" micro-programming.

Horizontal is meant to imply "wide", a large number of bits. With many bits available it is possible to encode very low level information, specifying all the gating at the adder, shifter, and register level. Thus, any of the capabilities of the circuit can be potentially exercised. Similarly, this in turn implies that any possible parallelism (e.g. independent shifter, and adder action) can be taken advantage of. The wide width of a horizontally encoded micro-instruction also in general allows for a fairly reasonable form of micro addressing to occur. That is, the address specification of the next micro-instruction can be directly specified with each micro-instruction. The micro instruction format (Figure 4.4.2-1) for the IBM 2050 (processor for the IBM 360/50) is 89 bits wide and is an example of this form of "horizontal" encoding [Hu 70]. The Nanodata QM-1 slightly modifies this normal concept of horizontal encoding to include four "time steps" within a single micro-instruction. This width is a total of 342 bits [Nc 71, p. 9-1].

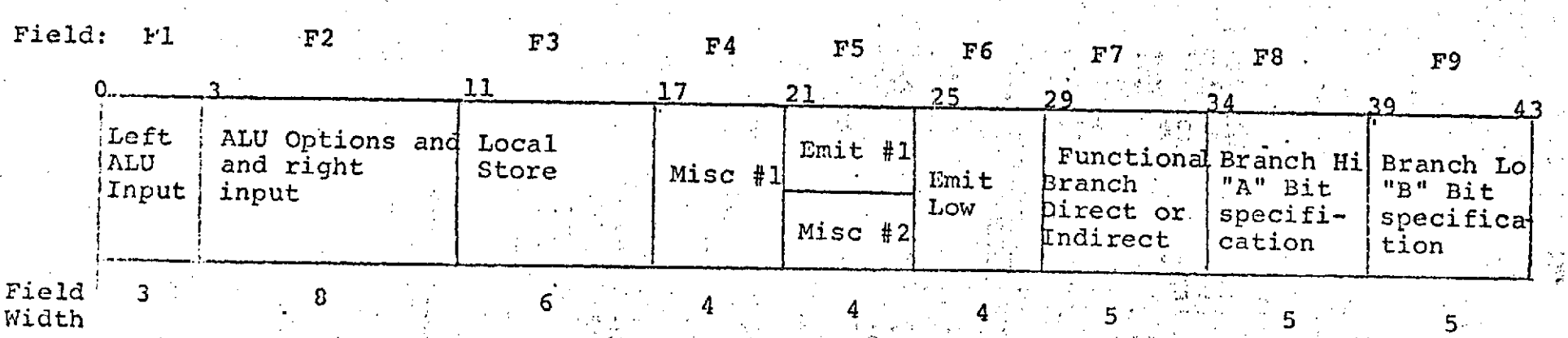


Figure 4.4.2-2

AP101 Microprocessor - Micro Instruction Format [Va 72]

Vertical is meant to imply a "narrow" width for the micro-instruction. This is accomplished either by encoding the possible gatings into mutually exclusive fields, as for example selecting but one register to be the left input to the adder, or by minimizing the address capability within each micro-instruction, (either requiring a separate instruction to branch, or allowing only a branch of a few bits), or a combination of these two. The Shuttle computers, the AP-101, is an example of a partial vertical encoding (Figure 4.4.2-2) being 43 bits in width. The B1700 has a micro-instruction width of 16 bits [Va 73] being extremely encoded.

The Burrough's D-machine [Bi 70] combines both of these concepts. It has a two level encoding. On the lowest level, it has a "nano" store with a horizontal encoding being 54 bits in width and a vertical encoding of 16 bits in width. While the nano-store (horizontal) indicates the normal inter-connections, function specification and simpler micro addressing; the micro store (vertical) is used as a source of literals and larger addressing fields (Figure 4.2.2-3).

The QM-1 has also adopted this concept. Besides having a horizontal encoding, as mentioned above, it also contains a vertical encoding used to provide access to the routines of the nano store (horizontal).

From the practical point of view, the difference between these methods of encoding is a question of dollar cost and execution speed. The more "horizontal" a micro-instruction, the less decoding required and thus potentially the faster the execution. But this in turn requires a larger micro-instruction store (more bits) which in turn is more costly. In the other direction, the more vertical a micro-insturction, the more decoding that is required before the designated inter-connections can be completed and functions executed. But in return, there is a reduction in the amount of micro-instruction storage.

INTERPRETER

MICROPROGRAMMING REFERENCE CARD

MICRO CONTROLS

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	*	SAR					ø	ø	ø	ø	ø	ø	ø	ø
1	0	*	SAR					LIT							
1	1	0	ø	*	AMPCR										
1	1	1	0	ø	ø	ø	ø	LIT							
1	1	1	1	*	NANO ADDRESS										

ø Unused
* Shorter fields are right justified

NANO CONTROLS

Parentheses surround optional lexic units, provided by default.

Brackets contain DC 2000 mnemonics

? Codes not produced by TRANSLANG.

[1 2 3 4] Condition Tested
Result is Boolean end

0	0	0	0	GC1	
0	0	0	1	GC2	
0	0	1	0	LC1	
0	0	1	1	LC2	
0	1	0	0	MST	
0	1	0	1	LST	
0	1	1	0	ABT	
0	1	1	1	AOV	
1	0	0	0	COV	
1	0	0	1	SA1	[RMI]
1	0	1	0	ROC	
1	0	1	1	LC3	[RMA]
1	1	0	0	EX1	[EXT]
1	0	1	1	INT	
1	1	1	0	EX2	[SRQ]
1	1	1	1	EX3	[URQ]

[5] FT Condition Value

0	NOT	$\overline{\text{cnd*:SC}}$
1	--	cnd*:SC

[6] Logic Unit Conditional

0	Do Unconditionally
1	Do Conditionally if SC

[7] Ext Op (MDOP/CAJ) Conditional

0	Do Unconditionally
1	Do Conditionally if SC

[8 9 10] Condition Adjust -- CAJ

0	0	0	--
0	0	1	SET LC2
0	1	0	SET GC2
0	1	1	RESET GC
1	0	0	SET INT
1	0	1	SET LC3
1	1	0	SET GC1
1	1	1	SET LC1

[11 12 13] Successor

Then Part Used if SC=1			to MPAD Ctls	Else Part Used if SC=0		
0	0	0	WAIT	0	0	0
0	0	1	(STEP)	0	0	1
0	1	0	SAVE	0	1	0
0	1	1	SKIP	0	1	1
1	0	0	JUMP	1	0	0
1	0	1	EXEC	1	0	1
1	1	0	CALL	1	1	0
1	1	1	RETN	1	1	1

[14 15 16]

[17 18 19] Adder X Input

0	0	0	(0)
0	0	1	LIT
0	1	0	ZEXT [EXT]
0	1	1	CTR
1	0	0	Z
1	0	1	A1
1	1	0	A2
1	1	1	A3

[20 21 22 23 24 25 26] Adder Y Input

0	0	-	-	-	-	-	B0--
0	1	-	-	-	-	-	BT--
1	0	-	-	-	-	-	BF--
1	1	-	-	-	-	-	B1--
-	-	0	0	0	-	-	B-0-
-	-	1	0	0	-	-	B-T-
-	-	-	-	-	0	0	B--0
-	-	-	-	-	0	1	B--T
-	-	-	-	-	1	0	B--F
-	-	-	-	-	1	1	B--1
Comp		1	0	0	Comp		B-F-*
Comp		0	0	0	Comp		B-1-*
0	0	0	0	1	0	1	LIT
0	0	0	1	0	0	0	ZEXT [EXT]
0	1	0	1	1	0	0	CTR
0	1	1	0	1	0	1	Z
0	0	1	1	0	0	1	AMPCR
0	1	1	1	1	0	1	[L0]
Others							?

*Use Adder Operation with Complement Y

[27] Inhibit Carries into Bytes

0	--	Allow
1	IC	Inhibit

[28 29 30 31] Adder Operation

						Logic
0	0	0	0	X	+ Y	
0	0	0	1	X	NOR Y	$\overline{X}\ \overline{Y}$
0	0	1	0	X	NRI Y	$\overline{X}\ Y$
0	0	1	1	X +	Y + 1	
0	1	0	0	X	NAN Y	$\overline{X} \vee \overline{Y}$
0	1	0	1	X	OAD Y	X + (X ∨ Y)
0	1	1	0	X	XOR Y	$X\overline{Y} \vee \overline{X}Y$
0	1	1	1	X	NIM Y	$X\overline{Y}$
1	0	0	0	X	IMP Y	$\overline{X} \vee Y$
1	0	0	1	X	EQV Y	$XY \vee \overline{X}\,\overline{Y}$
1	0	1	0	X	AAD Y	X + (XY)
1	0	1	1	X	AND Y	XY
1	1	0	0	X -	Y - 1	$X + \overline{Y}$
1	1	0	1	X	RIM Y	$X \vee \overline{Y}$
1	1	1	0	X	OR Y	$X \vee Y$
1	1	1	1	X	- Y	$X + \overline{Y} + 1$

[32 33] Shift Type Selection for BSW

0	0	--	No Shift
0	1	R	Right End Off
1	0	L	Left End Off
1	1	C	Right Circular

[34 35 36] A Register Input from BSW

0	0	0	-- No Change
1	-	-	A1
-	1	-	A2
-	-	1	A3

[37 38 39 40] B Register Input Select

0	0	0	0	--	No Change
0	0	0	1	BC4	Comp 4 Bit Carries
1	0	0	0	BAD	Adder
1	0	0	1	BC8	Comp 8 Bit Carries
1	0	1	0	BBA	BSW ∨ Adder
1	0	1	1	B	BSW
1	1	0	0	BEX	External
1	1	0	1	BMI	MIR
1	1	1	0	BBE	BSW ∨ External
1	1	1	1	BBI	BSW ∨ MIR
Others				?	

[41] MIR Input from BSW

0	--	No Change
1	MIR	

[42] AMPCR Input from BSW

0	--	No Change
1	AMPCR	

[43 44 45 46] Mem Dev Address Input

0	0	0	-	--	No Change
-	-	1	0	LMAR	From LIT
-	-	1	1	MAR	From BSW
-	1	0	-	BR2	From BSW
-	1	1	1	MAR2	From BSW
1	-	0	-	BR1	From BSW
1	-	1	1	MAR1	From BSW

[46 47 48] Counter Input

-	0	0	--	No Change
0	0	1	LCTR	From LIT*
1	0	1	CTR	From BSW*
-	1	0	INC	+1
-	1	1	?	

*Ones Complement

[49 50] SAR Input

0	0	--	No Change
0	1	CSAR	Complement
1	0	SAR	From BSW
1	1	?	

[51 52 53 54] Mem Dev Op--MDOP

0	0	0	0	--	No Change
0	0	1	0	MR1	
0	0	1	1	MR2	
0	1	1	0	MW1	
0	1	1	1	MW2	
1	0	0	0	DL1	[ASR]
1	0	0	1	DL2	[ASE]
1	0	1	1	DR1	
1	0	1	1	DR2	
1	1	0	0	DU1	
1	1	0	1	DU2	
1	1	1	0	DW1	
1	1	1	1	DW2	
Others				?	

Burroughs Corporation
DEFENSE, SPACE AND SPECIAL SYSTEMS GROUP
PAOLI, PENNSYLVANIA 19301

D-Machine Micro-processor - Micro and Nano Instruction Formats

Figure 4.4.2-3: [Bi 70]

Besides these cost arguments, the vertical encoding by its very nature removes some of the possibilities for the circuits usage. While this could potentially reduce some useful execution capabilities, it in general does not, since few of all the possible horizontal encodings would ever be useful.

What is more serious in extreme vertical encodings is the cost of micro-instruction addressing capability. If no sequencing information is provided, then this vertical instruction becomes in nature, similar to the normal Von Neumann machine architecture. That is, for example, a lack of parallelism in processor execution and the general requirements for two instructions to be executed in instruction sequencing,i.e. a separate branch instruction is required. Thus, the penalty becomes not only an execution time lossage due to "vertical" encoding, but indeed time lossage due to a second micro-instruction fetch before a change in sequencing can occur.

It is seen that the choice of encoding effects both dollar cost and execution time capabilities. From the point of view of the development of a higher order language architecture, however, this is a minor consideration. Time can be conveniently counted in time steps rather than the real execution time. The implementation of the HOLM architecture on a micro-processor gives insight into problem areas, but being a tool in design is not overly restrictive. In a production version of an HOLM, the insights gained in its developmental implementation would allow for the appropriate modification of the underlying support processor.

4.4.2.2 Degree of Parallelism. One advantage of a micro-processor over the standard mini computer is that it is possible to make more efficient usage of the processors circuitry. As mentioned in the last section, many micro-instruction encodings allow for sequencing information in each micro-instruction. Thus, upon the execution of each micro-instruction there can be made a conditional choice of the next micro-instruction. This saves a time step when compared to the normal computers which are required to execute a following branch instruction.

It is also often possible to execute the various active elements in the same time step within a micro-processor. Thus, for example, the shifter and adder of the QM-1 can be executed separately in the same time step while even incrementing another register. Often, memory accessing can be initiated and overlapped with the micro-processor, e.g. Burroughs D-machine.

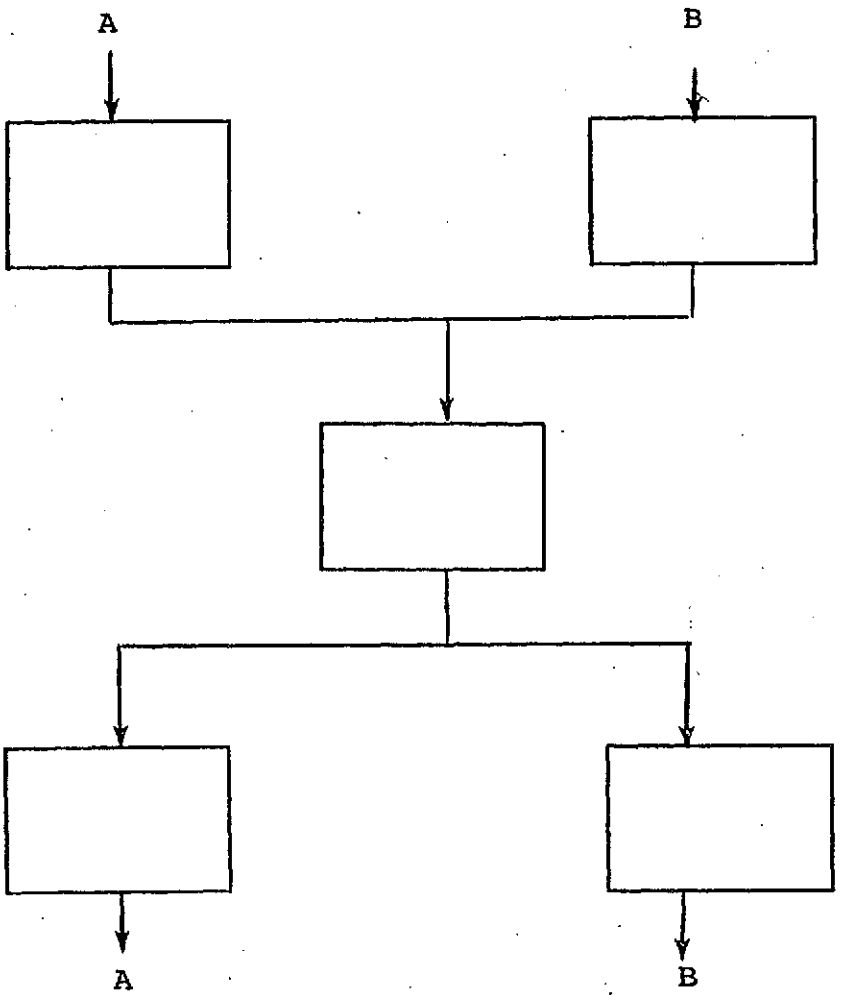
The advantages of the use of parallelism within the processor is, of course, the time savings involved. The price is having a relatively wide micro-instruction encoding and the complexity of more than a simple single bus between the various executing elements.

From the point of view of the implementation of a higher order language machine, any specific parallelisms are not required for development. But a production version can benefit highly from the appropriate combination of certain limited parallel functions, such as stack manipulation (maintenance of stack indicators) while also using the ALU.

4.4.2.3 Bit Testing and Field Extraction. There are two basic extreme philosophies with regard to accessing particular bit fields within a main store instruction. One extreme is to be able to access any bit field within the main memory instruction within any given micro cycle. This would thus allow for a micro-processor to have a general emmulation capability: no matter how the (any) instruction architecture is encoded, it can be swiftly and efficiently decoded since any bit field can quickly be accessed and tested. The price for this capability is to have a "barrel switch"; a field isolation unit that can both shift, mask and test the resultant value of a word within one micro-instruction clock time. This indeed is included in the design of the Burrough's D-machine. The B1700 has a similar capability but is done differently and can often require several micro-instructions in order to complete the process. Once such a feature as a barrel switch is developed and the initial developmental cost covered, it can become an effective element of any processor.

The other extreme is to allow access to just those bit fields that are of interest for the particular instruction architecture being implemented by the micro-processor. This is indeed the method used by the AP-101. Thus, this does not require the use of such a complex element as a barrell switch within the processor. It is accomplished instead by placing the appropriate random logic required to access and test the fields of interest. While this does not therefore lend itself to the capability of general instruction architecture emmulation, it does prove to be a cost effective engineering technique in the development of a production computer.

One other difference between these two methodologies should be noted. The second method allows the following micro-instruction to be executed upon results of the fields and/or conditions specified. In the first case, though more general, the field of interest to be tested must often first be isolated via the barrel switch, and which would take an extra micro-instruction clock step to do. (Sometimes, of course, it could take more, and other times the second method itself would take several clock steps in order to generate the desired result). Between these two extremes are many possible design compromises. While a micro-processor may have an oritentation to a particular main instruction architecture format, it may also have a fairly good field isolation and testing capability.



Common Sub-Routines

Figure 4.4.2-4

The B1700 has extremely good testing and sequencing control since it can efficiently manipulate and isolate fields from zero to 24 bits.

In order to use a micro-processor for instruction architecture development, it is extremely important to have the capability of field isolation and testing. It is to be noted, however, that this generality often causes more time steps than if an "ideal" micro-processor was available which is specifically oriented towards the instruction architecture being emmulated. While this is no handicap during development, and indeed can be considered a great advantage since no bias towards certain field usages and designations are present, it is not the most efficient method for a final production version.

4.4.2.4 Sequencing. In conjunction with bit testing and field isolation, the method of sequencing found in a micro-processor can allow for both efficiency and ease of implementation of a higher order language machine, or it can allow for the opposite: inefficiency and difficulty. It has already been stressed how micro-instruction addressing correlates to micro-instruction bit width (horizontal/vertical) and how this inturn can imply either parallel next instruction selection or the need for an extra clock step.

The capabilities of the micro-instruction addressing are also of interest. Often, these consist of but simple branches which thus form a linearization upon the micro control flow. If a sequence such as in Figure 4.4.2-4 is required, this would in turn require the setting of a flag in order to differentiate the source and hence the return from the common subroutine. While this is often not a problem with micro-processors used for standard Von Neumann architectures, it can pose a problem for those processors used to emmulate higher order languages. The solution, of course, is to provide for modularization: CALL and RETURN. This is effectively done on the D-machine by use of a simple alternate micro program counter, thus providing for the savings of the return address. In the B1700, an actual return stack is provided for several levels of calling. As in most cases, the penalty for this cleanliness is, in general, a degree of inefficiency: that is, a call and a return must be performed.

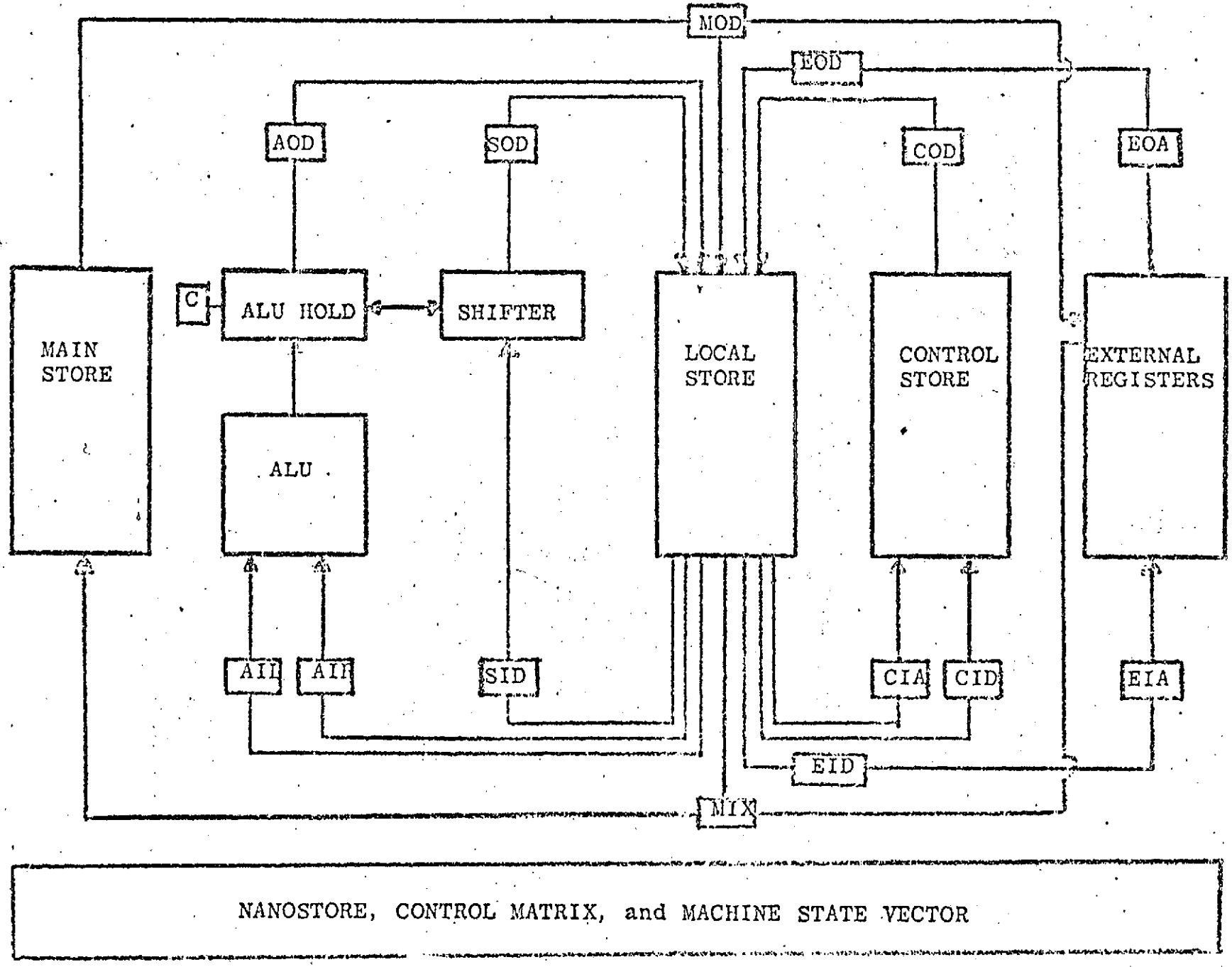
(In the D-machine, this form of sequencing is part of each nano-instructions option and thus does not provide a time penalty). However, if modularization is required, this is no more inefficient than the setting and testing of some flag.

Another consideration is with respect to the design process: modularization allows for a clean design which can be modified rather than having the design controlled in development by addressing and size restrictions. Thus, modularization is also an important feature for a micro-processor if it is to support general emulation.

4.4.3 Micro-Processors Under Consideration

In recent years there have been a plethora of micro-processors made available on the market. Many of these have been developed by mini computer manufacturers in order to gain entry into the "micro computer" market and in order to allow their customer to do some tailoring to his specific needs. Often, however, these mini micro computers are merely the standard mini computer with some access to fast memory. That is, the micro-instruction set itself is basically indistinguishable from a standard mini computer instruction set. The instruction format is vertical with no parallel processing capability, and requiring sequencing instructions. Further, it is usually required that any "new" instruction added rigidly follow their current instruction formats with regard to field size, location, and meanings. Finally, they are usually limited in the amount of control store available for this extra usage. These statements do not, of course, pertain to all cases.

For a variety of reasons, the micro-processors which were seriously considered and examined were the Nano Data QM-1, The Burrough's D-machine, the IBM AP-101, and the Burrough's B1700. The QM-1, D-machine and B1700 each have been initially designed to be emmulators and interpreters for higher order languages. The AP-101 on the other hand is the micro-processor used for the Space Shuttle program and upon which HAL/S will be implemented in the standard fashion. The B1700 is a newer design than either the QM-1 or the D-machine and has taken emmulation a step further than the other two. The B1700 uses bit addressing of memory and is basically free of any particular bit width restrictions (no inherent bytes or words), fields or formats. This, along with its commercial availability, makes it the most desirable micro-processor for developmental work.



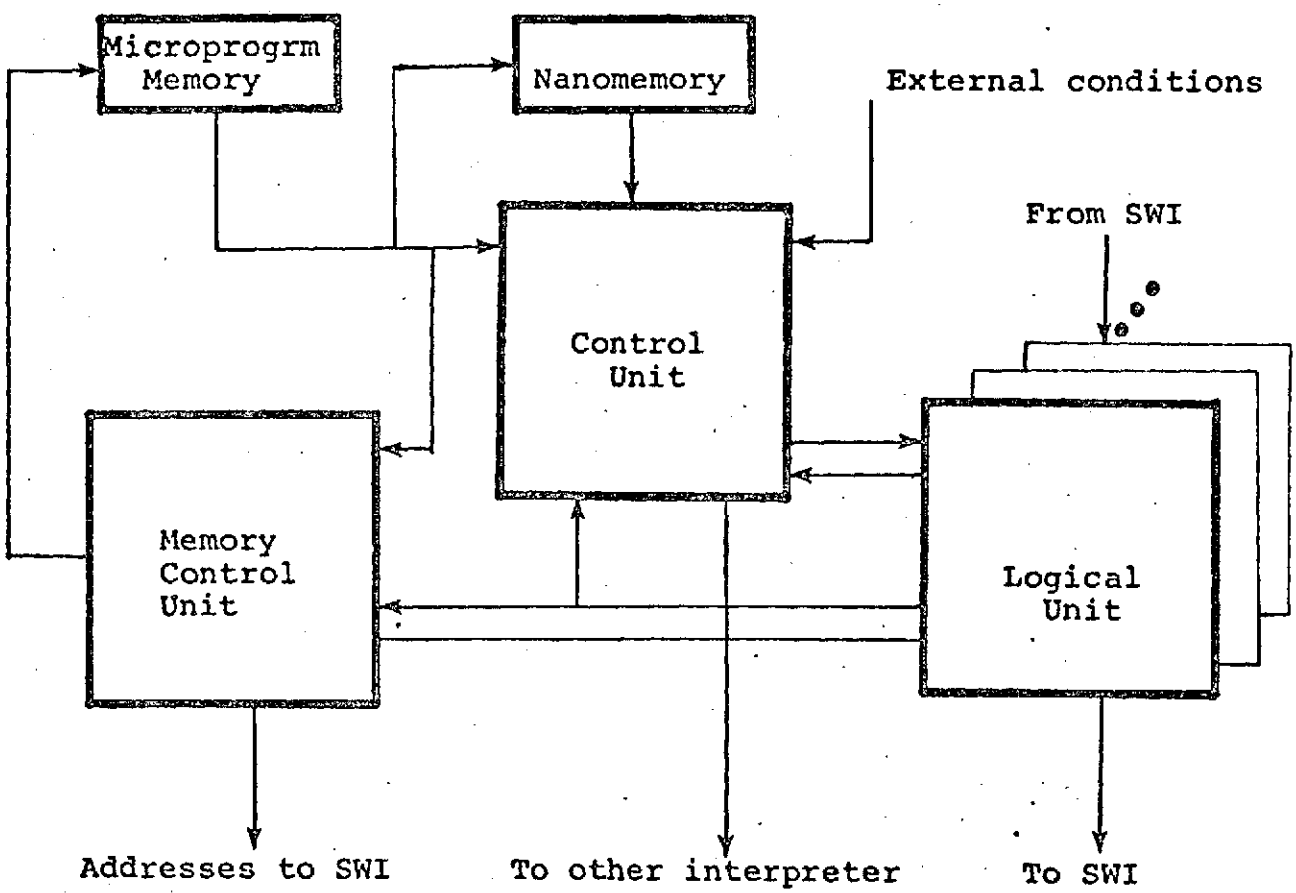
QM-1 ORGANIZATION AND DATA FLOW

Figure 4.4.3-1 [Nc 71]

4.4.3.1 The Nano Data QM-1 [Nc 71]. The QM-1 offers an exceptional degree of flexibility in a processor unit. Control is effected by double level emulation with a micro-control store driving a nano-control store. The micro memory is a writeable control store. The data width is 18 bits. One of the major features of the machine is the variety within the memory hierarchy. This includes main memory up to 512K bytes of 750 ns core, a local store of thirty-two 18 bit registers, external register consisting of thirty-two 18 bit registers, control store of up to 32K 18 bit words, and a nano store up to 1K 360 bit wide. This hierarchy of storage with the extremely wide nano memory, and potentially large degree of processing parallelism would certainly prove quite satisfactory for implementing the proposed instruction set.

One important shortcoming of the machine is that the word length is fixed to 18 bits. While this is not a handicap for developmental work, it would penalize its execution for the standard aerospace units of 32 bits in actual operation.

The generalized structure of the QM-1 appears to be ideal for emulation, which indeed is what it was designed for. The reason that the QM-1 can not currently be considered is that it is not easily available for usage, and thus is currently an unrealistic choice for HALM development. A study of its structure, however, proves very fruitful in comparing micro-processor designs (Figure 4.4.3-1).



Burrough's D-Machine
Interpreter Block Diagram

Figure 4.4.3-2

4.4.3.2 Burrough's D-Machine [Bi 70]. The Burrough's D-Machine is an unusually modular and flexible architectural design, which is capable of application to a wide variety of problem areas. In its basic multi-processor configuration, it consists of three major building blocks: interpreters, switch interlock, and memories. The interpreter is a micro-programmed processor and is used to perform both arithmetic/logical computation and I/O device control. The switch interlock is the communication network which links interpreters, operating memory, and I/O devices.

The D-machine interpreter is constructed from five functional parts: memory control unit (MCU), control unit (CU), logic unit (LU), micro program memory (MPM), and nano memory (NM), (Figure 4.4.3-2). The word length of the interpreter depends only upon the logic unit, which is modular in 8-bit blocks, from 16 bits to 64 bits. The use of micro programming enables the control logic to be quite regular in structure, resulting in economy of manufacturing. Additionally, different micro programs may be used with the same hardware to implement different instruction sets for different applications. Furthermore, if a read-write rather than read-only micro programmable memory is attached, the system can reload this memory dynamically to run programs written in different machine languages at different times.

To save storage, the micro program structure of the interpreter has been divided into two logical sections: micro and nano. The control of functional operations within the interpreter is dictated by the contents of a location in nano memory. Each of the 56 bits corresponds to a control line for the elements of the LU, CU, and MCU. A given nanoword is selected under control of a micro word which specifies the nanoword's address in nano memory. As a result, nanowords may be referred to by many micro words; hence, the bit saving.

Burroughs is producing both a commercial and a military version of the interpreter-based system. The commercial version is being used for disk controllers and for other applications not yet announced.

A major shortcoming of the D-machine appears to be the fact that there is little local storage associated with an interpreter. However, Burroughs has indicated that a memory unit could be attached to a device port, which would serve the function.

The D-machine would be a good candidate for a micro-processor implementation of HAL. But as with most micro-processors, it has a definite byte and word orientation. The data units would have to be choosen to be some multiple of 8 bits. This structuring of sizes varies greatly in philosophy from the bit orientation and non forced structuring of the B1700. This, in conjunction with the easier access to the B1700, removed the D-machine from active consideration.

4.4.3.3 The IBM AP-101 [Va 72]. The IBM AP-101 is a micro-processor oriented to the execution of a modified IBM 360 type of instruction architecture. Intermetrics has in the past examined the capabilities of this processor under contract first to IBM and latter within the Space Shuttle program under contract to Rockwell International.

The data width of the AP-101 is 32 bits. It contains a single 32 bit ALU and a register file containing 32 32-bit registers. The instruction decoding is specifically oriented towards the current AP-101 instruction architecture. While it is always possible to emulate any particular instruction architecture, the AP-101 was not designed for this purpose and any such use would become very inefficient. The micro instruction addressing capability is basically oriented towards a limit of 4K by 44 bit micro words. The physical implementation is actually less than that limit.

Since the AP-101 is the computer to be used for the Space Shuttle, it was of interest to see how it would be able to support a HAL machine design. However, its specific instruction format orientation and micro addressing structure make it unfeasible to consider it as a design tool. Further, it would be impossible to have access to it in order to develop an implementation, since, for example, the micro store itself is not writtable.

A study of the AP-101 micro processor design is interesting in the fact that it has taken a very pragmatic engineering approach for a cost effective implementation of its instruction architecture [Pa 70].

4.4.3.4 The Burrough's B1700. The B1700's design objective was "to give 100 percent variability, or the appearance of no inherent structure". [Wi 72a]. It was designed to be an essentially unbiased emulation facility, able to adopt to any instruction architecture used to support the language being emulated. The general structure, philosophy, and usage of the B1700 has been published in a series of three papers by W.T. Wilner in 1972 [Wi 72a, Wi 72b, Wi 72c].

The basic qualities of the B1700 indicated in these papers include:

- Bit Addressible Memories

In order to be free of structure restrictions, there are no mandatory byte or word boundries inherent in the processors architecture. The hardware supports the memory access in such a way that there is no penalty for addressing and particular bit address (even though physical memory is eight bit units).

- Field Widths are Free to Vary

Besides having bit addressible memory, the field width accessed and processed are free to vary for 1 to 65K bits. The internal bussing and ALU are capable of automatically handling information in units of from 1 to 24 bits. If larger units than 24 bits are to be processed, this would require further memory accessed (access is in 24 bit quantums), but the processing can be performed without the involvement of the user.

- Good Bit Testing and Field Manipulation

As a corallary to the bit addressing capability, the B1700 provides for efficient manipulating, and sequencing upon 4 bit units while being able to easily manipulate and extract 1 to 24 bit units.

- Writable Micro Memory

The system was designed to support a multi emulator capability. The micro instruction executes out of main memory, but may be buffered by fast circuits. The ability to modify and develop an emulator is inherent in the design and its philosophy.

- Designed for Multi Emulators

The B1700 was intended to operate in a multi-emulator mode. Thus, the facility for this form of development explicitly exists. Similarly, the problems of common executive and I/O interfaces has been resolved. The interface to each emulation is standardized and the I/O and other executive functions supported. Thus, the development of a new emulator can principally concentrate on the instruction architecture under development.

- Micro Code Facilitates Modularity

The addressing structure of the micro code is such that micro-procedures may be defined both re-entrantly and recursively. The micro-processor hardware supports a 32-deep hardware stack. This then enables clean modular design with minimal penalty.

The B1700 is a commercial machine which is fairly accessible for usage. Upon the request of Intermetrics for work under this contract, the Burrough's corporation allowed access to further information upon the B1700 micro-processor and for its actual usage in the development of a HALM emulator. The details of the B1700 micro-processor design has not yet (it is believed) appeared in general publication, and are currently considered propriatory by the Burrough's Corporation. The availability of this information has greatly helped the pursuit of the HALM development in this contract.

4.5 Implementation

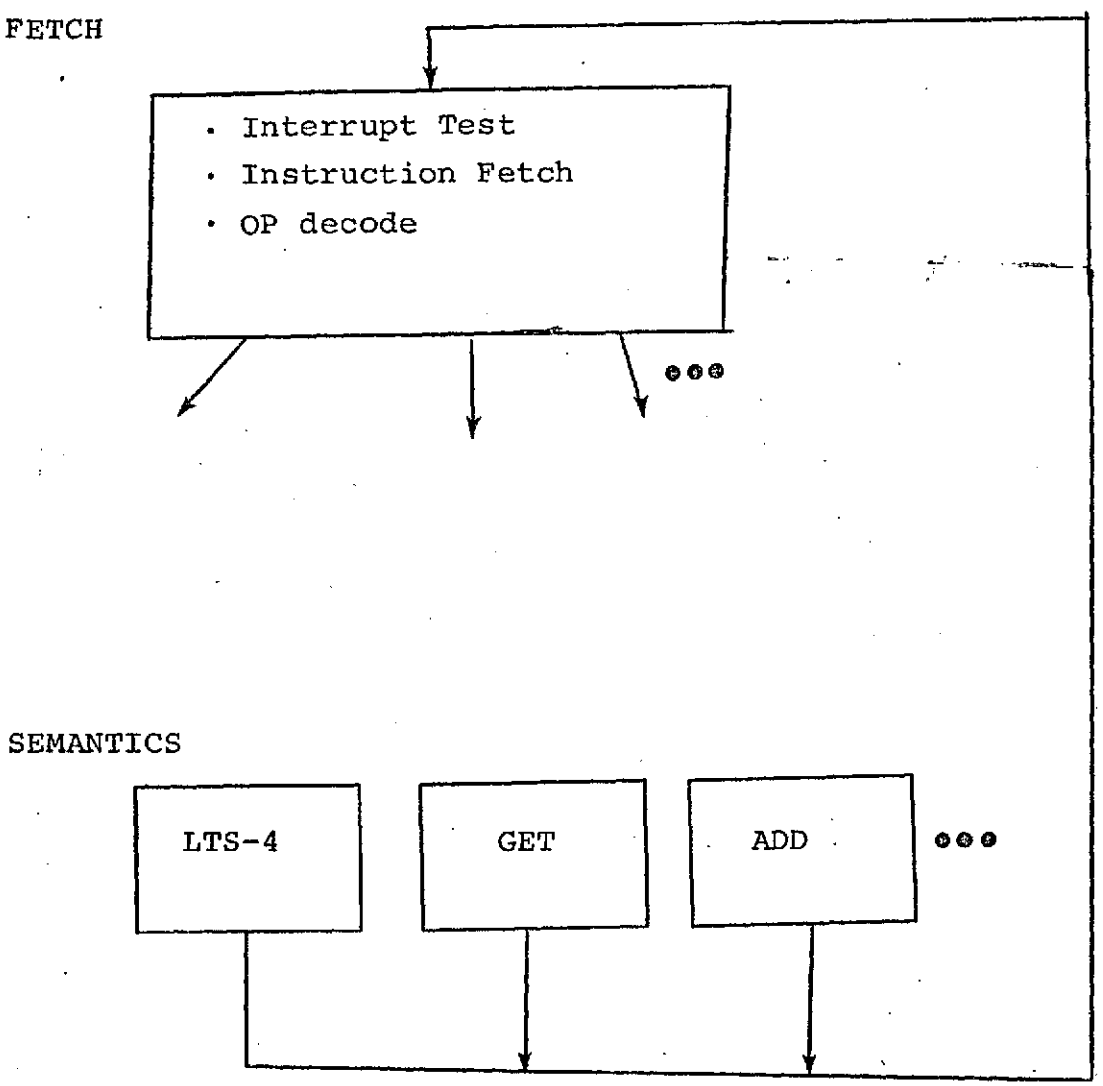
In order to investigate the implementation of a HALM, the B1700 was choosen to be the host processor for the baseline MP instruction architecture. The baseline MP instruction architecture is more highly structured (byte and word oriented) than is required by the B1700. While modification of the design could have improved the efficiency of the HALM design, the limited time available for this task made this prohibitive. Requirements for, and possible modifications to, the HALM addressing structure were, however, investigated in parallel to this implementation task (Section 4.3), thus providing a basis for future improved implementation.

The two main results of this task have been the detailed investigation and analysis of the B1700 capabilities and limitations for the implementation of emulators; and the design and partial implementation of a modified MP instruction architecture.

The remainder of this section will discuss the programming environment and conventions provided by the B1700 for HOL emulators; the high level design of the MP instruction architecture implementation; and examples of instruction architectures encoding. Section 4.6 will discuss the limitations and possible modifications to the B1700 (also applicable to other micro-processors) for improved HALM execution.

4.5.1 B1700 Emulator Environment

The Burrough's B1700 was designed as an emulation vehicle. It does not have any preference for a particular instruction architecture or format sizes, and various HOLs can be supported in their own fashion. This amorphousness is inherent in its design philosophy. The B1700 was designed for a multi emulator environment. From this decision arises the requirement that there be a standardized interface to the operating system and for I/O processing. It is by convention that the various emulators interface in a particular way. This is not inherent in the processor's design itself. Being but a gentleman's agreement, it is the responsibility of each emulator to test for any required interrupt servicing at convenient times (normally the start of each new HOL instruction cycle) and then to return to the operating system having saved one's own environment.



Instruction Execution

Figure 4.5.2-1

I/O, for example [Wi 72c], is handled by sending or receiving a string of bits. A pointer to a buffer area along with a device code is passed to the operating system. The system is designed so that the HOL emulators may assume perfect I/O transmission. I/O devices, from the HOL emulator point of view, can be assumed to be present and ready, and the results obtained to be error free. This philosophy is consistent with a top down design. Responsibility for I/O preparation is not placed upon each HOL emulator, but rather upon the next level of service, in this case the operating system. The emulators do, however, have the responsibility of periodically checking to see if there is a high priority I/O process waiting to be performed; that is, the micro-program must check for I/O interrupts.

Other operating system functions for multi-programming are handled in a similar fashion.

This establishment of a standardized operating system and I/O handling greatly facilitate the use of the B1700 as a design tool for the development of instruction architectures. Concentration can thus be placed upon the development and refinement of the instruction architecture language structures.

4.5.2 Implementation Structure

The basic flow of all instruction set implementations follows the basic pattern:

- instruction fetch
- op decode
- semantic routines

Instruction execution begins by obtaining the next instruction. The opcode of this instruction is then decoded. This decoding is used to indicate the meaning of the instruction: what function is to be performed. Control is transferred to the appropriate routine and the semantics of the instruction is performed. Figure 4.5.2-1 indicates this flow with respect to B1700 usage.

- HALM to B1700 Interface Routines
  - SET-UP-HALM — initial entry set up
  - SWAP — return interface
  - RUN-TIME-ERROR — error handling
- HALM Instruction Architecture Requirements
  - PUSH-STACK — register to memory
  - POP-STACK — memory to register
  - REGISTER-FILL — fill the top of stack
  - GEA — calculate physical address
  - FORM-DESCRIPTOR — fill in descriptor fields
- Common Semantic Subroutines
  - GET-2-OPERANDS — set up stack for dyadic operator
  - GET-1-OPERANDS — set up stack for monadic operator
  - PUT-RESULT — set up stack with operator's result
  - MULTIPLY-16-16 — fixed point multiply service routine
  - Floating Point Support Routines

Working Subroutines

Figure 4.5.2-2

In the micro-instruction set of the B1700 is an instruction for directly reading from the main memory. With this instruction it is possible to simultaneously execute the read and increment/decrement the address pointer while simultaneously incrementing/decrementing an associated counter. This allows for very efficient memory referencing since the bookkeeping and maintenance of pointers and counters are simultaneously provided for. This allows for modification of the standard instruction flow to only two basic steps:

- instruction FETCH and DECODE
- SEMANTIC routines

In the description of the B1700 micro-processor (Section 4.4.3.4), it was indicated that there was great facility in the manipulation and testing of four bit fields. This allows for the decoding of the opcodes by four bits at a time. It is possible to do an effective 16 bit do case by "or"ing a four bit field to the micro-instruction program counter. Thus, op code decoding occurs in steps of four bits at a time.

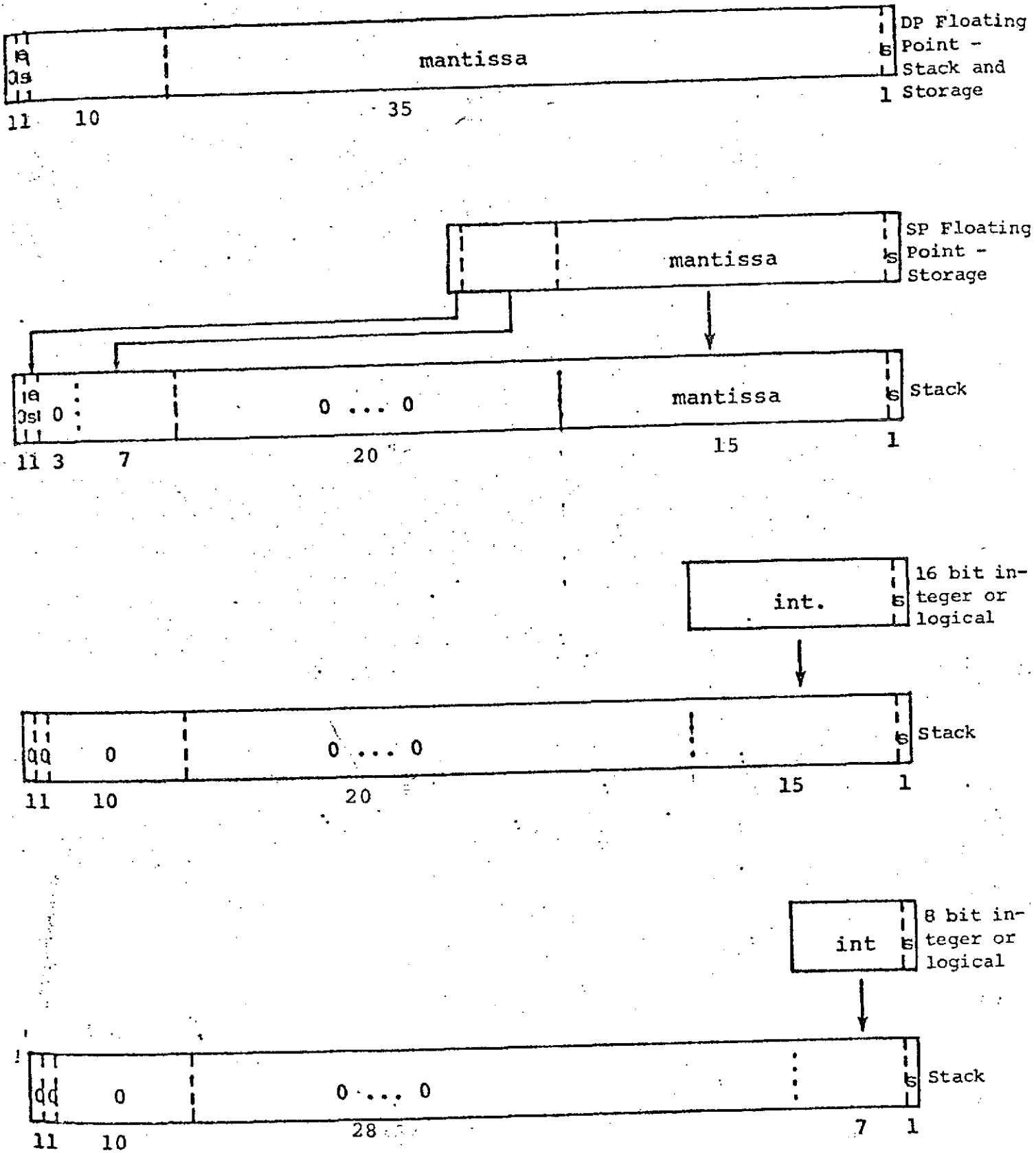
The semantic routines then perform their appropriate functions as defined in the instruction architecture. These are the basic routines that actually execute the instruction function such as ADD, COPY, Store, ...

The B1700 was designed with a micro-instruction level stack mechanism which allows for reëntrant and recursive routines. This design modularity allows each of the semantic routines to call upon a series of service routines for common functions. These functions can either be reflective of the bookkeeping required for the instruction architecture, e.g. stack Push or Pop, calculate effective address, ...; or bookkeeping required by B1700 conventions, e.g. operating system and I/O interfaces; or they can be a common function of two or more the semantic routines, e.g. floating point normalize.

Figure 4.5.2-2 gives a summary of the basic service routines associated with an implementation of the MP instruction architecture. Other service routines would also exist because of the desire for modularity and clean design. The B1700 allows each of these routines to be encoded in a fashion similar to normal machine instructions.

This section has presented the basic structure of the HALM implementation consisting of three parts: 1) the FETCH routine for obtaining and decoding the instruction, 2) the semantic routines for their interpretation, and 3) various support routines.

Arithmetic forms have been reduced to 24 and 48 bit units.



Modified MP Instruction Architecture
Arithmetic Type Formats and Mapping to Stack

Figure 4.5.3-1

4.5.3 Implementation Examples

A partial implementation of the MP instruction architecture was made during this study. During this implementation, several modifications to the baseline MP instruction architecture were made. In particular, since this was an investigation and analysis task, modifications were made to the arithmetic types as described in the baseline. This was caused by the fact that the internal data width of the B1700 is 24 bits. Thus, it is more aimiable to manipulations of quantum either less than this size or multiples of it. In particular, it was decided that instead of supporting a stack of 64 bits width, it would support a stack of 48 bits of width. This does not directly effect the other portions of the MP architecture, but only changes its data types, descriptors, and special words. The change, however, facilitates the implementation on the B1700. While the 64 bit format can be supported by the B1700, it required more care in details and bookkeeping. Figure 4.5.3-1 shows the modifications to the arithmetic formats for the modified MP architecture. Similar minor modifications also were required for the descriptor and special word formats.

Three of the implemented routines are now given as examples. These are the FETCH routine, and the two semantic routines, LTS4 and LOR.

```
FETCH                              * ROUTINE LABEL
  MOVE 24 TO CP                    * SET FIELD CONTROL TO FULL WIDTH
  MOVE FETCH-ADDRESS TO TAS        * RESET UP THIS ROUTINES ADDRESS
  IF ANY-INTERRUPT THEN            * TEST FOR HI-PRIO INTERRUPT
  BEGIN                            * IF THERE IS ONE:
    MOVE CHECK-FOR-INTERRUPT-CODE TO X
                                   * WHICH CODES TO TEST FOR
    CALL SWAP                      * SEE IF SHOULD RETURN TO O.S.
    MOVE L TO Y                    * HAVE SUCCESSFULLY COMPLETED
    IF Y NEQ O THEN CALL RUN-TIME-ERROR
                                   * ERROR HAS OCCURRED
  END
  MOVE NEXT-INST-PTR-TO FA         * PLACE PC FOR MEMORY FETCH
  READ 24 BIT TO T INC FA          * READ THE NEXT 24 BITS
  EXTRACT 4 BITS FROM T(0) TO L
                                   * OBTAIN THE FIRST 4 BITS
  MOVE L TO M                      * OR IT TO THE MICRO INSTRUCTION
  JUMP FORWARD                     * JUMP UPON THE 4 BITS
    GO TO EIGHT-BIT-OPS            * 0000
    GO TO EIGHT-BIT-OPS            * 0001
    GO TO EIGHT-BIT-OPS            * 0010
    GO TO EIGHT-BIT-OPS            * 0011
    GO TO LTS4                     * 0100
    GO TO LTS4                     * 0101
    GO TO LT-OPS                   * 0110
    GO TO LTLD-LTLDX               * 0111
    GO TO COPY                     * 1000
    GO TO COPY                     * 1001
    GO TO GET                      * 1010
    GO TO GET                      * 1011
    GO TO ADR                      * 1100
    GO TO ADR                      * 1101
    GO TO ADRE                     * 1110
    GO TO ADRE                     * 1111
```

The Initial Op Decode of Four Bits

Figure 4.5.3-2

4.5.3.1 FETCH Routine. Section 4.5.2 indicated the function of the FETCH routine in the HALM implementation. It is responsible for checking for any interrupt, for obtaining the next instruction from memory, and for the actual opcode decoding process. Figure 4.5.3-2 shows this routine as written for the modified MP instruction architecture. Figures 4.5.3-3 through 4.5.3-5 show the MP instruction architecture encodings as given in Mi 72, (errors being corrected). These are the encodings that have been implemented in the FETCH routine.

Going through the FETCH routine, the following is seen:

- The data width to be used within the processor is set to 24 bits. By setting the CP to a value of 1 to 24, the ALU will act accordingly on that bit width.

- The address of the FETCH routine itself is now placed upon the micro instruction stack. This allows the semantic routines, when they are finished, to do an EXIT (e.g. a GOTO the address indicated by the value on the top of the micro-instruction stack).

- The interrupt flags are tested to set if there is an interrupt present. If there is an interrupt, the mask of those of interest is passed to the SWAP routine. If control must be returned to the operating system, this will be done so by the SWAP routine after appropriately saving the emulators environment (i.e. registers). Upon return from the SWAP routine, it is checked to see if all is satisfied or if there is an error. If there is an error related to the process, control is given to a routine to handle it.

- After any interrupt processing has been handled, if present, the program counter (PC) is placed into the memory address register. Twenty four bits of memory is now read, and the PC is incremented by this 24.

- The first four bits from this memory read are now extracted from the 24 bits.

- The extracted 4 bits, the opcode, are now "ored" into the next micro-instruction. This effectively modified the next micro-instruction's address field.

- This next instruction is a branch. The low four bits of address have been modified by the four bits of opcode which have been extracted. Therefore, a 16 way branch can now occur.

- A comparison of the sixteen GO TOs with the encoding presented in Figures 4.5.3-3 through 4.5.3-5 show that each of these branches now go to the appropriate semantic routine.

  Bits 0000 to 0010 need further decoding and thus now go to an EIGHT-BIT-OPS decode routine which does another appropriate fan-out.

  Bits 0100 and 0101 both go the the LTS4 instruction routine.

  Bits 0110 must be further decoded to discover which literal operator is present. Hence, this branch goes to a LIT-OPS routine for decoding.

  Bits 0111 are either a LTLD or a LTLDX instruction. It thus goes to a routine which will perform the appropriate semantics.

  Bits 1000 through 1111 are similarly decoded and go appropriately to either the COPY, GET, ADR, or ADRE instruction routines.

Control returns to the FETCH routine when the appropriate instruction semantics are completed.

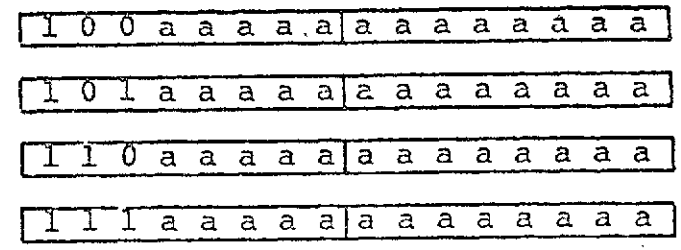
It is interesting to note that the preference of the B1700 for 4 bit fields has resulted in a 16 way fan out for this routine. Often, other forms of opcode decoding are possible. Either random logic or fields of a larger width can be tested. These methods pay the penalty of either being special logic, non modular, or if the field size is larger than 4 bits, it will cost more memory in its usage (8 bits implies 2^8 fan out or 256 fan out: but the literals and operands are encoded in 3 of 4 bits!).

OPERATOR	BYTE LENGTH	FORM
a) All Instructions (except those below)	1	0 0 v v v v v v — v..v provides codes
b) Exceptional Instructions		
BST m,n	3	0 0p p p p p p p | 0 0 m m m m m m | 0 0 n n n n n n
BLD m,n	3	0 0p p p p p p p | 1 1 m m m m m m | 0 0 n n n n n n
BOUT m,n	3	0 0p p p p p p p | 1 0 m m m m m m | 0 0 n n n n n n
BIN m,n	3	0 0p p p p p p p | 0 1 m m m m m m | 0 0 n n n n n n
		m ... m bit field length n ... n starting bit position
BSETL n	2	0 0q q q q q q | 0 0 n n n n n n
BRSTL n	2	0 0q q q q q q | 0 1 n n n n n n
BCHGL n	2	0 0q q q q q q | 1 0 n n n n n n
BTSTL n	2	0 0q q q q q q | 1 1 n n n n n n
		n ... n bit position
JCC m	2	0 0r r r r r r | 0 0 0 0 m m m m

MP Instruction Encodings (1)
Figure 4.5.3-3

c) Operand Meta-Operators

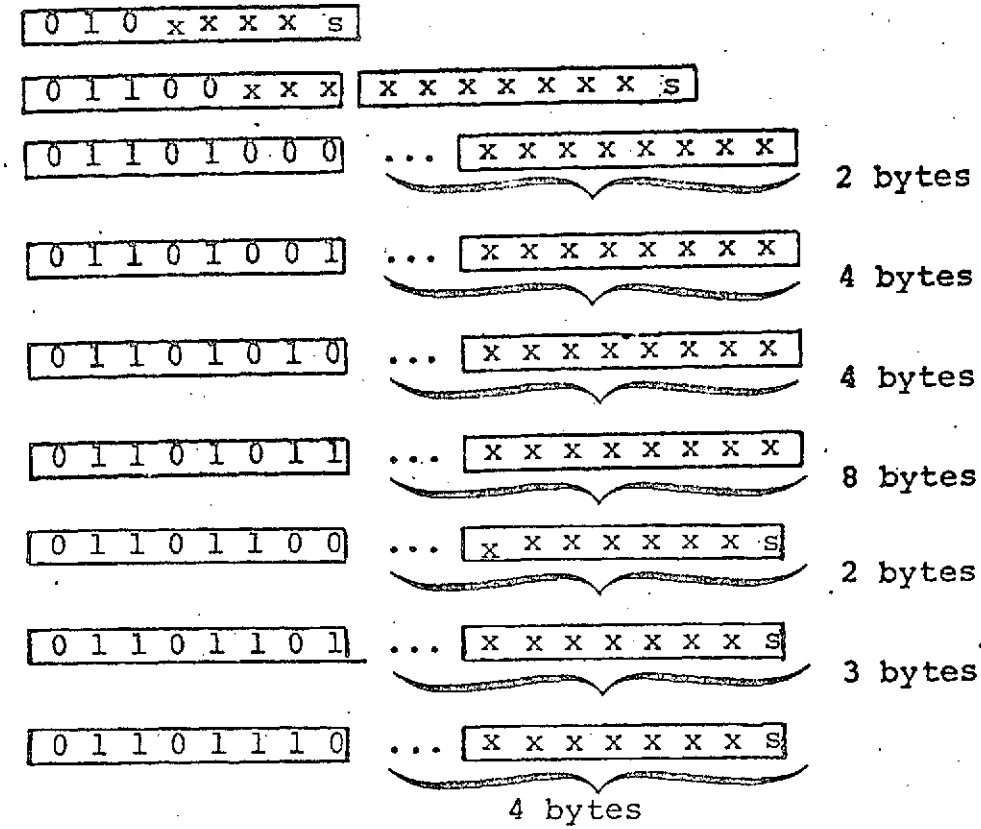
COPY	2
GET	2
ADR	2
ADRE	2



a...a lexical level, displacement

d) Literals

LTS4	1
LTS10	2
LTS15	3
LT32	5
LT32F	5
LT64	9
LTS7M	3
	4
	5



Legend: s - sign bit
x...x - numerical value

MP Instruction Encodings (3)
Figure 4.5.3-4

OPERATOR	BYTE LENGTH	FORM
LTLD	1	0 1 1 1 0 0 N N
	or 3	0 1 1 1 0 1 N N ... i i i i i i i i — 2 bytes
LTLDX	1	0 1 1 1 1 0 N N
	or 3	0 1 1 1 1 1 N N ... i i i i i i i i — 2 bytes

NN: literal to be loaded
- 00 signed 7 bit
- 01 signed 15 bit
- 10 32 bit flt. pt.
- 11 64 bit value

i...i: literal table address

MP Instruction Encodings (3)

Figure 4.5.3-5

```
LTS4

    COUNT FA DOWN BY 16          * FIX PC ADDRESS CORRECTLY
    EXTRACT 5 BITS FROM T(3) TO Y
                                 * GET THE 5 LITERAL BITS
    CLEAR X                      * ZERO THE X REGISTER
    CALL PUT-RESULT              * PLACE RESULT: X,Y INTO STACK
    EXIT                         * RETURN TO FETCH ROUTINE
```

(Note: This rotuine assumed a 48 bit arithmetic format versus the baseline MP 64 bit arithmetic format; refer to Figure 4.5.3-1).

Semantic Routine: LTS4 Implementation Load Signed 4 Bit Literal Into Stack

Figure 4.5.3-6

4.5.3.2 LTS4 Semantic Routines. The LTS4 operator places a signed five bit quantity into the top of stack. Figure 4.5.3-6 shows the B1700 implementation of this instruction.

- Since the FETCH routine counted up the PC by 24 bits, but the LTS4 is only 8 bits in length, this routine must now decrement the PC back down by 16 bits. (The FETCH routine incremented the PC by 24 bits since it read the maximum amount that it efficiently could, and the increment of the address pointer can be done at the same time).

- Referring to the LTS4 format in Figure 4.5.3-4, shows the literal information in bits 3 to 8 of the operator byte. Thus, these five bits are extracted from the instruction register (T) which still contains the 24 bits of information obtained by FETCH. These five bits are placed into the Y register and then the X register is zeroed.

- By convention, the X and Y registers contain the high and low portions of a resultant value of an operation (in this implementation). Referring to Figure 4.5.3-1, it is seen that the literal is indeed in the correct format for placing into the stack.

- The routine PUT-RESULT is now called which will take the 48 bit XY value and place it in the top of stack. The PUT-RESULT routine worries about the bookkeeping of the stack: whether the A register is currently filled, whether the top of stack must be pushed to memory, etc.

- Finally, control is returned to the FETCH routine in order to process the next instruction. EXIT is a return using the address on the top of the

This routine is representative of the bit extraction capability of the B1700 and the ease of the code generation for it.

```
LOR
    COUNT FA DOWN BY 16          * FIX PC ADDRESS CORRECTLY
    CALL GET-2-OPERANDS          * SET UP TWO TOP OF STACK REGISTERS
    MOVE B-REG-2 TO X            * SET UP TO "OR" LOW
    MOVE A-REG-2 TO Y            * 24 BITS OF STACK
    MOVE  XORY TO L              * REGISTERS, SAVE IN TEMPORARY
    MOVE B-REG-1 TO X            * SET UP TO "OR" HIGH
    MOVE A-REG-1 TO Y            * 24 BITS OF STACK
    MOVE  XORY TO X              * REGISTERS, LEAVE IN X
    MOVE L TO Y                  * LOW 24 BITS TO Y
    CALL PUT-RESULT              * PLACE RESULT INTO STACK
    EXIT                         * RETURN TO FETCH ROUTINE
```

Semantic Routine: LOR Implementation Perform Logical Or Upon Two Top of Stack Registers

Figure 4.5.3-7

4.5.3.3 LOR Semantic Routine. The LOR operator performs a logical OR upon the top two operands upon the stack. The result is the left on the top of the stack. Figure 4.5.3-7 shows the B1700 implementation of this instruction.

- As with the LTS4 instruction, the PC must be decremented by 16 bits since the operator is only a syllable of 8 bits in length.
- The routine now calls the GET-2-OPERANDS routine. This routine makes sure that the two top of stack registers, A and B, contain values. This may require reading operands from memory or interpreting an address.
- The LOR routine then takes the low 24 bits of the A and B registers and places them as inputs to the 24 bit ALU (i.e. the X and Y registers).
- The "logical or" of these values is temporarily saved.
- The routine then does the same for the high 24 bits of the A and B registers. The "logical or" being placed into the X register.
- The low order 24 bits are now placed into the Y register, thus,forming the desired 48 bit result.
- Now, as with the LTS4 routine, the 48 bit result in the XY register is placed into the stack by the PUT-RESULT routine.
- Finally, control is returned to the FETCH routine for processing the next instruction.

This routine discloses the preference for 24 bits in the B1700 architecture. To process 48 bits took two steps through the ALU.

4.5.3.4 Routine Implementation. The previous three examples have shown how code is generated for the B1700. The process is basically straight forward with the ability to manipulate various bit fields as desired. The ALU itself provides the standard types of results such as "and", "or", "not", "exclusive or", "masking", "complementation", "addition", and "subtraction".

These three examples are sufficient to show how the B1700 is used and its possibilities. In the next section, several limitations and desired modification to its structure will be discussed.

4.6 HALM and B1700 Mutual Reflections

Implementation of the MP instruction architecture upon the B1700 highlights the assumption made during their individual developments. The free, basically structureless form of the B1700, indicates how HAL/S has presummed the necessity of rigid data formats. The ability of the B1700 to perform almost any form emulation, on the other hand, often results in time penalties when a specialized function is required. A proper design process consists of refinement, with feedback to the previous level, as artificial restrictions are discovered or pragmatic ones required.

4.6.1 HAL/S

The process of implementing the MP instruction architecture highlights the ease of implementation with the use of the B1700. But it also indicates where HAL/S has either general or complex capabilities whose requirement for a micro-implementation is debatable. Either because they are used very infrequently, or because they could consume large amounts of time, thus adversly interacting with real time process and I/O handling.

The B1700 also indicated areas where more generality for the HAL/S language does not involve efficiency penalties.

4.6.1.1 Ability to Implement a HALM. As previously discussed, the implementation of an instruction architecture can be viewed with respect to four separate categories: control sequencing, data addressing, functional transformations, and data representation.

The B1700 has a very clean modular structure. It is fairly easy to write a micro routine for any particular instruction. The control sequencing as presented in the MP instruction architecture is basically of a straight forward nature. As was mentioned previously, the MP instruction architecture was modified to have a 48 bit

width stack instead of the initial design of 64. This in turn forces modification to the special control words by narrowing their width. This constraints was not mandatory, but rather one of convenience since the internal bussing is of 24 bit width. (It would be possible to use multiples of 32 bits, but this requires more careful bookkeeping and more coding. This was not of importance for the investigation of the implementation aspects of both the instruction architecture and host micro-processor).

Similarly, the implementation of the MP instruction architecture data addressing is well defined. While the B1700 easily emulates this structure, it is to be noted that address manipulation (lexical level-displacement to stack number-offset to physical memory addresses) are performed in a step by step fashion using the general capabilities of the micro-processor. In a specific implementation of such an instruction architecture, it would be profitable to have the specialized capability for some of this, otherwise, sequential manipulation. Of course, in a non-real time or developmental environment, this is no real penalty.

HAL/S has a set of function transformations, semantic operations, more powerful than the conventional scalar arithmetic. These include the ability to do vector and matrix arithmetic along with generalized array processing of the various basic data types. These powerful operators could be encoded either as micro instructions, as are the scalar operations, or they also can be provided as basic instruction level subroutines. The advantages and disadvantages are of course memory size and execution requirements. In particular, the time granularity of response required for process and I/O interaction may make prohibitive the total calculation upon an array or even a large matrix. The question then of implementation depends upon statistics of HAL/S language usage, the capabilities of the micro-processor, and the real time characteristics of the required mission. Within the context of this study, these various complex operators were considered to be a refinement to the basic implementation and non essential for this initial investigation. Thus, they are, in general, assumed to exist as subroutines, (which of course, is how they are implemented, either in line or out of line, in the IBM 360 and AP-101 implementations).

One other set of operators are of interest in HAL/S. These are the real time or executive functions. HAL/S assumes that there exists an executive which is both priority driven and is capable of supporting the HAL/S real time statements. The B1700 was not designed, nor meant to be, a real time processor. It is oriented towards batch processing in the business community. Since the B1700 is also meant to execute in a multi emulator environment, it has already assumed a particular executive interface and its appropriate functions. Within the context of this project, this was the executive interface assumed for HAL/S.

The final area of data representation was also affected by the B1700. Soley for implementation convenience within the context of this project, the data representations were modified from the initial MP instruction architecture format of 64 bits in the stack to 48 bits. It is to be noted, however, the HAL/S language specification does not designate data types other than by the weak attribute of SINGLE or DOUBLE. The B1700 does not directly support floating point arithmetic data types. Rather, this must be encoded via micro subroutines. The only penalty paid, of course, is that of execution time. For the business community to which the B1700 is oriented, this is no problem since most of their arithmetic is in decimal (or binary) format. To efficiently support a scientific application where floating point calculations predominate, it would of course be desirable to have a special floating point capability.

One other reflection of the B1700 is the fact that it is able to support data representations of varying widths. Thus, it actually is practical to support a spectrum of data precisions within higher order language. It is easy to envision the higher order language having a precision attribute specifying the number of decimal digits required, and then having the storage thus allocated and the calculations thus performed.

4.6.1.2 Modifications to HAL/S. The last section indicated two interesting possibilities for the HAL/S language definition: one with respect to data type representation, and the other with respect to the executive interfaces.

The data representation was seen to be one of the four areas of language specification which are basically independent of each other. Further, HAL/S, as with most higher order languages, does not directly specify the arithmetic data representation to be used. Their policy of non-specification is a hedge. Higher order languages are usually implemented on various processors. There is no industry standard upon format representation, the Univac 1108 varying from the IBM 360 from the Singer SKC 2000 from the Burroughs 6700, ... It can, in general, also be legitamately argued that an add is indeed an add. If the precision provided is sufficient for any task, then the algorithm (encoded in the HOL) itself should not care about the data representation.

The variability offered in the B1700 for data widths indicates that perhaps what should be done is that a higher order language should specify the characteristics: precision and range, required for the variable, and thus make this a part of the algorithmic development. It would then be possible to have efficient use of both memory resources and to have an algorithm that would work "correctly" upon different host processors.

Another variant of this idea (in the context of the current generation of software) would be to have the data declared to have a particular data representation (instead of the required attributes such as precision and range) and thus be able to have the execution have the data characteristics of a known architecture: IBM 360, SKC 2000, B6700, ... This attitude, while not ideal from either the top down design or analytical approach, would be useful in the context of software verification, duplication, and reproducibility of results while allowing the introduction of more efficient instruction architecture and hardware implementations.

The other area of interest to HAL/S is the executive interface. While HAL/S goes in great detail in specifying the executive and real time statements, it would be of interest to see, as part of the specification, the other side of this interface. That is, HAL/S should also specify the assumed characteristics of an executive required to properly support HAL/S. In particular, since HAL/S is a real time language, it would be desirable to quarantee that a complex of HAL/S programs will execute identically in an identical real time environment when the processor and/or executive support have been changed. Basically, the specification desired is that equivalent to specification of a "multiply". It is not important how the circuitry is done, but rather that the same result be returned. In the case of HAL/S executive functions, both "time" and "processes" are entities whose interactions need be specified in order to obtain deterministic and reproducible results.

4.6.2 B1700

The B1700 has proved to be an excellent facility for the investigation, implementation and refinement of instruction architectures. The results of this study, of course, indicate several areas in which it is found lacking, highlights useful modifications, and indicates some of the general characteristics desired in any micro-processor used as a support for emulators.

4.6.2.1 Deficiencies. While the B1700 is an extremely efficient emulator for the general case, it has several drawbacks for use with HAL/S. In the aerospace environment, HAL/S is used as a real time process control language and must efficiently execute various scientific calculations.

While it is not impossible to have the B1700 execute in "real time", the amount of calculation that could be performed in such a manner is limited. Again, this is not that for which the B1700 was designed.

The newer aerospace computers have come to support floating point calculations. The advantages have to do with algorithmic specification, programming design and fewer conceptual or run time errors. When floating point is encoded into a micro routine, it of course takes in general quite a few time steps. These can become prohibitive if the floating point is used regularly. From the HAL/S point of view, it would be more than desirable to have direct floating point support, and thus improved

execution time. From the conceptual design point of view, this is of no importance.

The SKC-2000 and the AP-101 both have 32 bit single precision floating point formats (however, of slightly different representations). One drawback of the B1700 is its preference for, and internal bussing of, 24 bit data widths. While conceptually this does not alter the B1700's design, it has the real pragmatic effect of hampering in the efficient emulation of many current 32 bit width machines. It is quite easy to conceive of the B1700 design, but altered to have an internal bussing (and ALU capability, etc.) of 32 bits.

4.6.2.2 Possible Modifications. While the last section contained issues that are thought to be real drawbacks for the use of HALM-B1700 implementation in a real time environment, this section will contain possible modifications to the micro-processor's structure that would aid in execution efficiency.

- Special Opcode Facility

The FETCH routine illustrated the preference of the B1700 for four bit manipulations. The initial fan out in the opcode decoding was 2^4 or 16 ways. While this is an extremely efficient methodology in minimizing the number of steps required versus the amount of memory required, it can be seen that a large part of this routine is consumed with standardized testing and bookkeeping. Since the FETCH routine, by definition, occurs in each instruction execution, it would be reasonable to provide some further hardware support for this function.

This hardware support could take the form of a particular entry point for FETCH (thus no need to set up the FETCH address into the micro-instruction stack), automatic interrupt testing under mask, reading of the next instruction from memory, and a fanout to the specified routines (pragmatically again this would be on the order of the 2^4 or 16 way fanout).

Further sophistication could allow for a specification of encoding of the opcode bits. This would allow the minimum number of words required for the opcode jump table while still being fast by use of hardware support.

- Hardware Support of a Memory Stack

While it is easy to manipulate the register file of the B1700 and to read/write memory, the use of a stack in the higher order language instruction architecture usually requires some degree of bookkeeping. Thus, for example, if two registers are designated to be the "top of stack", A register, and "next to top of stack", B register, it becomes necessary to keep track of which is currently filled and of when it becomes necessary to push one or both of them into the memory portion of the stack, or to fill them from memory.

Since so many higher order language architectures are stack oriented (by the fact of the stacks correspond to both the compiler codes generation and the algorithm's run time execution characteristics), it would be quite reasonable to have two of the registers in the register file be designatable as the A and B registers, and then to provide hardware support for their maintenance and manipulation.

While this might seem to be a minor point, their continual need for maintenance in a stack oriented architecture becomes a sizeable overhead.

- Fixed Point Multiply

When one is multiplying by a multiple of two, this can be accomplished with great efficiency by merely shifting. In the B1700, where fields of any bit width may be used, numbers such as 3, 5 or 23 can usually appear. In the process of addressing elements of an array, for example, the index must be multiplied by the field widths. Thus, if the B1700 is being used efficiently, general multiplications must occur and not mere shifting. The way that this would currently be done is, of course, to call a multiply subroutine. This subroutine uses the adder in the normal repetitive fashion to accomplish the multiplication. Since HAL/S does have both a Vector/Matrix and array capability, the use of indices, either implicit or explicit, must be efficiently supported.

If a hardware fixed point multiple were provided, the manipulation of arrays of varying bit sizes and dimensions of course becomes that much more efficient as does the process of multi ranked entities which also involve general multiplication.

- Condition Code Testing

The B1700 provides both a very good mechanism for bit manipulation and field extraction, and for a large number of condition code testing. However, the bit manipulation and field extraction is not a part of the ALU, while most of the condition code testing is. It would be desirable to be able to have condition code on bits without having to use the ALU which would both consume another instruction for this move, and potentially destroy some useful information.

- Floating Point Support

As has been previously discussed, it would be beneficial from an execution point of view if floating point calculations were directly supported by the B1700 rather than being microprogrammed. In the scientific environment, towards which HAL/S is oriented, this is most important.

- Internal Bussing of 32 Bits

The desire for an internal bussing of 32 bits is the pragmatic desire to be efficiently compatible with a large number of processors currently available. With some loss of efficiency, it is of course possible with the B1700, to emulate a 32 bit architecture. Also, of course, certain applications may not require an arithmetic data representation greater than 24 bits or multiples thereof. Then, of course, the current B1700 is emmiently suitable.

- Descriptor and Addressing Support

While the previous suggested modifications were oriented towards the general support of any emulator, this suggestion presupposes a particular architecture with a particular representation. Once the formats and semantics of the addressing of an instruction architecture becomes known, it is then possible to specify subfunctions for their manipulations. It is these repetitive actions and bookkeeping that become prone to inefficiencies.

When, for example, the descriptor formats are given, specific hardware aids can be envisioned for tearing apart the information and its appropriate manipulation.

Similarly, when the addressing structure is designated, such as base-displacement in the IBM 360, or the lexical level-displacement, stack number-offset, and physical memory addresses, as in the MP instruction architecture, it is obviously more efficient if the hardware is capable of adding in the appropriate translations. That is, the micro-process must decipher the base-displacement form of addressing by:

a) extract the "base" bits ;

b) fetch the indicated register using the bits as an index ;

c) add the displacement bits to register value; and

d) use the resultant value as the memory address.

If hardware aid were available, the extraction of bits and fetch of registers and addition of displacement could all occur in one time step.

This form of aid is seen to be very dependent on the instruction architecture being emulated. But, this specialization in return greatly aids in efficiency.

All of the above modifications were not functional requirements in nature, but were rather related to the question of efficiency: the number of time steps required for the emulation of an instruction architecture.

4.6.2.3 General Micro-Processor Characteristics. From the discussions in the previous section and Section 4.4 on micro-processors, several general charcterizations may be drawn about the features desired in a micro-processor used for a HALM instruction architecture emulation.

- Easy Bit and Field Manipulation

In order to interpret the various formats, bit fields must be able to be manipulated.

- Condition code testing and branching

It must be possible to test bits and bit field and to make a decision upon the result.

- Modularization

In an instruction architecture oriented towards a higher order language, modularization becomes extremely important since there is the need of various common service and common semantic subroutines. Also, this allows for a clean design methodology.

- Special Hardware Support

In order to have an efficient emulation of a higher order language such as HAL/S it is desirable to have hardware support in the following areas:

- floating point support
- automatic top of stack maintainance
- special opcode decoding mechanism
- address decode aids such as fixed point multiply

A combination of generalized bit and field manipulation along with some specialized hardware supports, proves to be a very efficient methodology of supporting a class of higher order language machines.

4.7 Statistical Results

The comparison of instruction architectures requires an understanding of just what a meaningful comparison is, and which measures are useful; it also requires a method for obtaining these measures; and then finally the results obtained by this comparison. While the most desirable comparisons would have been made with the use of HAL/S Space Shuttle usage statistics, these were not available during the time period of this contract. However, the simple method of benchmarks allows for a meaningful, general comparison.

4.7.1 Useful Measures for Comparing

In order to compare various instruction architectures, it is necessary to choose a measure, or quantification, of some aspect of their design. From a realistic point of view, the only important measure of any system development is whether it can perform as needed within the cost and time constraints allowed. But, within this framework there are many different architectures available to a computer system with respect to network design, instruction architecture, implementation of the instructions, and the actual physical circuit design. In the consideration of a higher order language, there would seem to be three measures that could be considered as objective criteria for measurement of an (any) instruction architecture: time, space, and ease of use.

4.7.1.1 Execution Time. While initially time may seem to be a useful criteria, further consideration shows that the execution time of a program is basically independent of the instruction architecture itself, relying instead upon the logic and circuit design of the architecture's implementation. That is, while gross inefficiencies of instruction architecture design could have bad effects, "good" designs (in all the various forms: three, two, one operand or stack-oriented; single accumulator or multiple register; etc.) are largely dependent on the speed of memories, registers and logic, and the degree of parallelism used in the instruction execution. The actual number of fetches from memory can, however, provide a metric which accurately compares efficiency of one architecture to another. A further refinement would be to differentiate the memory accesses for instructions from those for data.

4.7.1.2 Memory Requirements. Space, the amount of main memory required, is however a real objective measurement that can be made. It is possible to separate the "logical" design of an instruction set, from the actual "physical" bit implementation. From an information theoretical point of view, even "logical" designs can be compared regarding efficiency of representing the information content of a given program. Memory is a major factor in system design, since currently it is the most costly physical component within a computer system. Reducing program length (compacting instructions) minimizes both hardware cost and execution speed. Hence, if the instructions fetched from memory have a higher information content, fewer memory fetches and total memory cycles are needed. In the measurement of memory needs, data memory may be differentiated from instruction memory. It is not to be expected that different machine architecture will vary greatly in data memory requirements since data (size) is predicated upon on the precision requirements of the problem under consideration. The instruction memory however, allows for a large memory savings. The design of several architectures for Higher Order Languages have claimed memory reductions from 25% to 75% [Sa 72].

a) Cirad [We 71] has reported that their SPL machine had yielded an overall reduction of 60% in the memory requirements over a traditional single-addressed architecture for implementing the same set of guidance equations and functions. The memory efficiency is reported to be "due to the use of Polish stack with implied addressing, the use of floating point, the number representation used, direct fetch of literals for instructions, built-in array operations and use of one of two byte instructions without word boundary restrictions".

b) Kerner and Gellman [DR 70] have designed a machine which directly executes Fortran statements. Programs written in this language and executed on their machine occupied 75% less memory. These results were atained by comparing the machine code generated by the Fortran compiler for the IBM 7094 with the numbers of words required to represent the instructions for the ILM. The 4:1 compression of memory space for program storage was the result.

c) Sugimoto [Su 69] has studied the direct execution of the PL/1 language and the implementation of his PL/1 reducer. For typical scientific programs, the length of the object code has been reduced by 25% compared to the object code generated by presently available PL/1 compilers. He also found a speed gain of 28% for arithmetic string operations.

d) Higher order language examples have demonstrated that a traditional machine architecture; viz. the IBM 360, uses at least twice as much memory as a specially designed computer, the Burroughs 6500. Distinguishing between the static memory size and the dynamic memory usage allows for a more efficiently compacted information and optimal design of the data.

e) As previously indicated in Section 4.3.3, Wilner [Wi 72a, Wi 72b] has reported program memory savings of from 40% to 70% with usage of the B1700 over current instruction implementations.

4.7.1.3 Ease of Use. The third criteria, "ease of use" is difficult to express quantitatively. It is, however, very real with respect to programmer usage: How easy is it to implement a program? When the system is to be programmed in a higher order language, the question is changed into whether the HOL can be easily and effectively implemented with the instruction architecture. This question of ease of usage also can be useful in examining an existent architecture with respect to what are the common programmer mistakes and errors, what do programmers find irritating and annoying, and what then are useful incremental improvements.

4.7.2 Methods for Quantifying Instruction Architecture Comparisons

Several methods have been suggested to compare proposed instruction architecture and produce quantitative results. The ideal solution would be to continue development on all candidates and measure performance with respect to cost and execution time after they have been built. This method of approach is hardly practical. An attempt to achieve the same results has been made by postulating some mix of instruction types and then evaluating the machine's execution time and memory sizing, based on the assumed mix. This approach, however, is open to question because of the assumptions inherent in the à priori presumed instruction mix. This fault is particularly apparent when comparing two architectures which are basically different, such as an IBM 360 versus a Burrough's B6700. They do not even begin to have the same, or similar, breakdown of instructions.

When dealing with a higher order language, such as HAL, it is more practical to take a different approach. Often, benchmark programs have been devised for comparative testing, but they have the drawback that they are seldom representative. They usually consist of but a relatively simple set of routines that do some well-defined tasks such as matrix multiply, sort, etc. They are inadequate since they ignore the real characteristics of a job's execution. It is most important to know how the machine executes programs in the application environment. Subroutine calling and exiting, saving of special index registers, linking conventions, and addressing are of interest insofar as they are utilized in the execution of actual programs.

In the selection of a computer from a set of already existing candidate machines, the use of benchmarks is often facilitated by the existence of the appropriate HOL compiler (e.g. FORTRAN) on each of these machines. The benchmarks then can be compiled and run and results compared on each of the candidates. The software as well as the hardware is tested in this fashion: it is only the success of the combinations of both that can produce good results and merits the ranking. It can be argued therefore, that fair and reasonable overall conclusions may be obtained. This method is not directly applicable to the development and comparison of new computer architectures, since compilers on these machines, of course, do not yet exist; further it is difficult to project accurately the picture of proposed job usage. However, the use of benchmark programs can still be a useful technique. Note that the code generated must also assume the capabilities of a compiler. A more detailed discussion of this method is given below in Section 4.7.2.1.

Besides the use of benchmark programs, the various language features of a HOL can be separated so that code generation and performance on the candidate machine may be examined. This then allows for a comparison between statement types on each machine as well as the additional ability of separately weighing the relative importance of statement types under discussion. This method of relative comparison was first developed by Wichmann [Wi 69, Wi 70, Wi 71, Wi72] who compared the implementation of different Algol compiler's code execution time. His methods were extended by Wortman [Wo 72] into a tool for the comparison and development of machine architectures. Section 4.7.2.2 will discuss a modified Wichmann approach; Wortman's approach is examined in Section 4.7.2.3.

To a large degree, the actual design of an instruction architecture itself can allow for a near optimal encoding. Once the basic logical instruction architecture is made, a Huffman encoding is then performed with respect to actual usage statistics of the language. However, this method itself has several limitations. 1) There is the assumption that the basic operators and operands have some how been designated, i.e. the logical instruction architecture has been made. But, this logical design can often itself be improved upon such factors as by examining the frequency of two or more instructions following each other. 2) There is the requirements that actual usage language statistics are available. If they are available, how representative are they? 3) Probably the most important decision which effects the encoding is the implementation of the addressing structure. The actual sizing of operand fields will highly effect the efficiency of encoding; but this is dependent upon usage statistics which can be interpreted in many ways depending upon how the addressing is handled. 4) Most important, it is to be noted that this provides for but or static bit encoding. It is not concerned with either execution time or with dynamic encoding. Thus,taken in extreme, it would become very inefficient and prohibitive.

It is thus seen, that while usage statistics can aid in the physical encoding of a logical instruction architecture, it is not in itself a sufficient methodology for the development of the instruction architecture. The instruction architecture is of a logical nature that must reflect the HOL (HAL/S) while taking in consideration current machine capabilities and possibilities, and in particular, the addressing structure must be developed. Even with this design work being done, improvement can be made outside of the architecture. Thus, the methods of Wichmann and Wortman provide a useful tool to highlight both the efficiencies and inefficiences of the instruction architecture allowing for improvement beyond more efficient Huffman encoding.

4.7.2.1 Method of Benchmark Programs. One method of obtaining a comparison between proposed instruction architecture involves encoding a series of benchmark programs for each proposed machine. The approach involves arranging a cursory compilation of representative programs; the resultant code is then examined in terms of both memory and time efficiency. This method eliminates one major source of discrepancy, namely the vagaries of individual compiler writers and their chosen techniques. Since the code generation is being performed by the developer, the compilation techniques remain constant, the results obtained should be a fair measure of each architecture's capabilities.

The application of this approach would consist of the following steps:

a) Selection of a subset of representative HAL programs. This may be based on those developed for the proposed usage if it differs from the general usage.

b) Postulation of a run time environment for each of the proposed architectures. Included would be assumptions concerning the compiler's use of the general register set if present (e.g. bases, indices, accumulators). It is necessary to define in detail the addressing assumptions used, and the method and number of things addressed. Allowance must be made for the number of entities in excess of the basic addressing policy. Also included is the definition of linking conventions: their type, purpose, size restrictions and various specialized formal parameter passage policies.

c) Given the basic run time environment, the mechanical policy for translating the HAL language features is adopted. Modification of the basic policies is allowed only insofar as it is reasonable to assume that a compiler could efficiently detect special cases. It is important to emphasize the global attitude and policies of a compiler versus those of an assembly language programmer. The assembly language programmer in general takes an extremely local contectual view in the generation of code.

d) Using the run time environment and mechanical code generation policies, generate the code for the selected HAL programs.

e) Statistics can now be directly obtained from the generated code. Size data can be gathered by direct examination of the resultant code. Speed information can be inferred (approximately) by counting the instructions to be executed and assuming equivalent hardware implementation for the comparative architectures.

While this method gives a basically sound comparison between various architectures, it does not indicate the relative merits of each architecture. Indeed, the assumption in generating code from benchmark programs is that the benchmark programs are indeed representative of the environment to be encountered in actual usage. While the code generation can be considered fairly accurate, the relative weighing of the various language features may not be so. Secondly, a small subset of a total run time environment does not approach, let alone emphasize, the limitations of a particular architecture. There are the limitations which are inherent in any instruction architecture. These include how many entities can be addressed, the size of a code module, the number of formal parameters which can be passed, and so on. It is important when developing an instruction architecture that these limitations of the architecture are carefully choosen and thus may be assumped to be reasonable for the proposed computer usage. These boundary limits will seldom be highlighted, or even encountered, by benchmark programs.

Nevertheless, it is this method (though not applied in a rigorous fashion) which is most convenient and easiest to apply with the initial investigation and development of differing instruction architectures. While a detailed analysis is often required when architectures vary but in small detail, a short benchmark is often helpful in differentiating architectures that vary greatly (e.g. Von Newmann architecture versus a stack-oriented architecture).

```
x := 1.0
x := 1
x := y
x := y + z
x := y × z
x := y / z
k := 1
k := 1.0
k := 1 + m
k := 1 × m
k := 1 ÷ m
k := 1
x := 1
1 := y
x := y↑2
x := y↑3
x := y ↑ z
e1[1] := 1
e2[1,1] := 1
e3[1,1,1] := 1
1 := e1[1]
begin real a; end
begin array a[ 1:1 ]; end
begin array a[ 1:500 ]; end
begin array a[ 1:1, 1:1 ]; end
begin array a[ 1:1,1:1,1:1 ]; end
begin goto abcd; abcd: end
beginswitch ss:=pq; goto ss[1]; end
x := sin( y )
x := cos( y )
x := abs( y )
x := exp( y )
x := ln( y )
x := sqrt( y )
x := arctan( y )
x := sign( y )
x := entier( y )
p0
p1( x )
p2( x, y )
p3( x, y, z )
```

Wichmann's Langugage Fragments

Figure 4.7.2-1

4.7.2.2 Modified Wichmann Approach [Sa 72]. A second approach to comparative evaluation can be made by extending a method presented by B.A. Wichmann [Wi 69]. Briefly, this method consists of defining a representative set of statements (figure 4.7.2-1) of the HOL (in this case Algol, in our case it would be HAL/S), and making a set of time measuremenst, Tij, for each representative HOL statement i (i=1 to n) on machine j (j=1 to m). Wichmann choose to use 41 representative statement types for his comparisons.

He then models these measurements as:

$$Tij = Fi\ Sj\ Rij \qquad 1 \leq i \leq n$$

$$1 \leq j \leq m$$

where Fi is a measure of statement complexity, Sj is a measure of machine performance, and Rij is a factor related to the machine's relative performance for a particular statement.

The assumption is that the execution time of a statement is somehow directly proportional to the "complexity" of that statement and to the "performance" of the particular machine. The Rij is then a measure of how much the particular Tij measurement varies from the ideal.

After obtaining the Tij measurements, the next step is to use these mn values and to determine the m + n values for the Fi and Sj. This is a valuable approach if the postulated measurements Tij are the only ones obtainable. However, the results are less than satisfying since the relative frequency of dynamic occurrence of the statements of the actual application is not taken into account. An extension of this approach is proposed as a more satisfying view of the problem of determination of statement complexity and machine performance.

Suppose a large sample of software were coded in the HOL. If these programs were executed on a commercial machine, under instrumentation which can observe the relative frequency of dynamic occurrence of each statement type wi, or the relative weights of each statement type is assumed upon aerospace statistics, then a more meaningful measure of machine performance (in this case, slowness: Pj) is given by

$$\sum_{i=1}^{n} wi\ Tij = Pj \qquad 1 \leq j \leq m$$

The P-values are analogous to Wichmann's S-values, but are renamed to avoid confusion. These P-values are computed from the measured statement execution times on the j machines as defined by the matrix Tij adjusted by the statement execution frequency estimation for the proposed application software.

In an analogous manner, the relative measure of the memory utilization can be obtained. Let Mij be the amount of memory needed to represent the HOL statement i, and the machine j. The static distributiion of HOL statements can be obtained for the benchmark by counting the HOL constructs in the code. Define σ_i as the static distribution. Then a relative measure of memory efficiency can be obtained by

$$\sum_{i=1}^{n} \sigma_i\ Mij = Aj$$

The Aj values are relative measures of the memory sufficient for each machine.

Since the Pj have been determined, the statement complexities Ci in the Wichmann equation can be written as:

$$Tij = Ci\ Pj\ Qij \qquad 1 \leq i \leq n \qquad 1 \leq j \leq m \qquad (1)$$

where the Qij and Ci are related to the Rij and Fi of the Wichmann equation. This is mn equations in n(m+1) unknowns. To obtain a "best fit", we chose to minimize the variation of the Qij relative to the Ci, therefore define:

$$E = \sum_{ij} (LQij)^2 = \sum_{ij} (LCi + LPj - LTij)^2$$

where the prefix L on a variable indicates the logarithm of that variable. This leads to

$$LCi = \frac{1}{m} \sum_{j} (LTij - LPj)$$

and the Qij may then be computed from (1).

The interpretation to be placed upon the Qij values is that they reflect the inefficiency of machine j executing statement-type i, relative to how that machine executes other statement-types, independent of the statement-complexity and frequency of execution.

The values Qij then, allow for an understanding of the structure of the machine with respect to the HOL. This would allow insight as to the ability of the machine to carry out particular functions not specifically considered in the weighting of the HOL statements.

In this method, therefore, it is necessary to first develop a set of statement-types to be examined with respect to code memory size and execution time. Further, in order to develop the P, an assumption of their relative weights is made. After this it is possible to develop a meaningful measure which is capable of indicating inefficiencies in the design of the respective architectures.

4.7.2.3 Wortman's Approach. Wortman [Wo 72] enlarged Wichmann's approach both by having static and dynamic metrics, and in his particular choice of metrics. These metrics which Wortman used, included two terms for the description of the static characteristics of programs:

a_1 number of bits required to represent the instructions,

a_2 number of bits required to represent the data,

and four terms for the description of the dynamic characteristics of programs:

a_3 the number of memory references required to fetch instructions during program execution,

a_4 the number of memory references required to access (fetch or store) data during program execution,

a_5 the number of bits of instruction fetched during program execution,

a_6 the number of bits of data accessed during program execution.

Using each of these attributes with each of the associated language fragments (statement-types) and then experimentally obtaining:

s_j static frequency of the language fragment,

and

d_j dynamic frequency of the language fragment f_j for computer p.

This allows for the development of cost measurement with respect to either measure, or assumed weighing functions

$$\omega_i^{(p)}$$

for the statement fragments f_i for the machine p. This then leads to a total cost formula of:

$$M^{(p)} = \sum_{i=1}^{k} \omega_i^{(p)} \left(\sum_{j=1}^{m} q_{ij}^{(p)} s_j \right) + \sum_{i=k+1}^{n} \omega_i^{(p)} \left(\sum_{j=1}^{m} q_{ij}^{(p)} d_j^{(p)} \right)$$

In actual practice it is generally sufficient to be able to calculate:

$$\sum_{j=1}^{m} q_{ij}^{(p)} s_j \qquad i = 1,2$$

$$\sum_{j=1}^{m} q_{ij}^{(p)} d_j^{(p)} \qquad i = 3,4,5,6$$

without obtaining the $\omega_i(p)$ in detail. This is true since the difference between various architectures is great enough to see the difference in most language fragments. Further, in using this information for incremental design improvement, the relative changes in each fragment can be clearly seen.

While Wichmann limited himself to 41 statement-types, Wortman performed his comparison upon 284 statement-types [Wo 72]. The statement fragments as presented by Wortman, modified by the additions and subtractions of features, would be quite appropriate for HAL. These added features would primarily concern the real time features of HAL; the primitive arithmetic types in HAL of vector and matrix and their associated operators; the HAL TASK blocking; general HAL flow control statements including DO CASE, EXIT and LOOP; and the use of the HAL sub-array capability. The deletions would include the ALLOCATE and FREE statements.

For a detailed comparison of various instruction architectures, this is the method which is most beneficial. While it is possible to make a basic statement of the efficiency of one form of instruction architecture as compared with another by using benchmark programs, this is but a gross measure which fails to indicate, in detail, where the efficiencies and inefficiencies lie. By examining each language feature, and thereby producing statement fragments, it is possible to find the inefficiencies and hence to allow incremental improvements in the design.

However, this level of analysis which is necessary for the fine tunning, the incremental improvements, of an instruction architecture was neither required nor feasible within the context of this study. The investigation of stack-oriented architectures and various addressing possibly in themselves greatly reduced memory requirements when compared to Von Neumann architectures, and thus a benchmark form of comparison suffices. The applications of the Wichmann/Wortman approaches requires actual usage statistics when used as a design methodology. But it is to be noted that during the time period of the performance of this work, actual HAL/S Shuttle usage statistics had not yet become available.

4.7.3 Comparisons of Architectures

Section 4.3 discussed the importance of addressing and provided a comparison between the IBM 360 and AP-101 code generation of the HAL/S compilers. (The actual code generation and HAL/S program are contained in Appendix 1). For the purposes of the implementation on the B1700, a modified MP instruction architecture was adopted. Appendix 2 shows the encoding of the same HAL/S program, CUBES, using this instruction architecture. There are several interesting things to note about the resultant comparison.

While the AP-101 reduced the IBM 360 code size by 32.6%, the MP reduced the IBM 360 code size by 42.5%. This is by 10% more (Figure 4.4.2-2). The MP instruction architecture also managed to reduce the address field portion to 56.6% of the bits used (Figure 4.4.2-3) versus 76.5% of the IBM 360, or 68.7% of the AP-101 (Figure 4.3.1-3). Only 901 bits were required for addressing with the MP versus 1298 for the AP-101 and 2144 by the IBM 360. Yet, the AP-101 only required 590 opcode bits while the MP required 691 bits. The reason for this discrepancy in favor of the AP-101 is simply that the initial MP instruction architecture design was byte oriented with the majority of operators requiring 8 bits, while the AP-101 was able to obtain a large number of 5 bit of opcodes. Even with this advantage for the AP-101, the total result showed more efficiency for the MP architecture.

Any final (next) physical mapping of the MP instruction architecture would be a great improvement on the current good results. 1) Actual usage statistics will become available and allow for an efficient Huffman encoding of the opcodes (thus, by definition be as compact as possible) and 2) when using a basically format free micro-processor such as the B1700 there is no requirement to have "syllable" operators, but rather 5 or 6 bits, or what ever operator is most informationally efficient may be used. It is hoped that in the next bit mapping another 20 to 40% reduction in space may occur.

4.8 Supra-HAL/S Usages

It has been seen that a micro-processor allows for a relatively efficient implementation of a higher order language instruction architecture. There are other possibilities for the use of a micro-processor other than just HOL implementation. Certain features of a language may be too complex and thus, prohibitive for implementation; but other features, which normally give rise to difficulty, e.g. error handling, may be easily implemented with the aid of micro code. Looking beyond the HOL language instructions, it is seen that whole routines may be written in the micro code if their simplicity and frequency of usage warrant it. Besides features related to a particular HOL and its usage, there is the whole area of executive support which can be greatly enhanced by use of micro code.

4.8.1 Language Features and Routines

It was previously indicated that certain of the HAL/S semantics may be too complex than to be worthwhile to implement in the micro code. These would include the general array and matrix processing. Besides requiring excessive micro memory for implementation, they require a large amount of processing time, perhaps more than would be allowed for the real time processing. But, it is also possible that certain language functions (which are usually defined to be a function in the language specification and treated as such during implementation) may be of frequent enough occurrence and simple enough nature to be effectively implemented in the micro code.

These routines might consist of some of the trigonometric package as has often been suggested [Pa 70]. These, however, are not usually of a very high occurrence in actual practice. Another possibility following the same line of thought would be to implement some basis for the generation of the various trigonometric functions, thus aiding in all of their implementations. (Conceptually, for example, implement $e^{i\theta}$ in micro code).

Recognizing that a HOL implementation normally functions with a support package of routines including the trignometric, vector and matrix, and other arithmetic functions, it would seem very reasonable to carefully consider their linkages. In particular, these "system" routines are both well defined and completely known by the compiler performing code generation. Whether a particular routine is to be in micro code or to be implemented with the normal instruction set, could be determined by statistical usage or execution requirements. In either case, the executive environment required by these system routines is very limited and defined. Thus, it is possible to generate linkages which take particular advantage of this fact and need not set up the general environment. An example of this concept of linkages can be found in Va 73a pertaining to proposed modification for the AP-101 for the Space Shuttle computer.

Generalizing this special interest taken in the functions and routines defined to be part of the language (SIN(X); ...), it would of course be possible to actually encode other routines written in HAL/S into micro code. This would be done either because the timing characteristics of the routine are so critical that they must be made more efficient, or else the frequency of usage of this routine is so high that a dramatic saving in throughput is to be gained by such an implementation.

While it is possible to envision an automatic mechanism for either generating standard HALM code or actual micro code for a particular routine during compilation, the need for this complex and difficult code generation capability would not in general be warranted. By the definition of the routine which is candidate for such an encoding, it is an exceptional case.

4.8.2 Executive Usage of Micro Code

The operating system of a particular computer is not necessarily directly encoded in, or even related to, the higher order language used by the application programmers.

Two promising areas for executive/micro code interaction are in the data structure required by the executive and in the interfaces to the higher order language programs.

Executives require certain general forms of data structures which are not directly supported by scientifically oriented HOLs. These data structures would include queues, stacks, and various linked lists structure. Often, these basic structures are "built" by specifying an array or structure in the executive's implementation language. Then, a few basic routines are written to treat the "built" data structure in the appropriate manner. These routines would indicate such things as ENQUE and DEQUE elements for queue data structures, and ENTER, REMOVE and SEARCH for the link list data structures. It is obvious, that if the executives implementation language were to have these data types as primitives and their manipulative routines as language primitives, then a micro code implementation would greatly improve its execution efficiency.

Besides general data structures, any particular executive has specific data structures which are basic to its operations. These would include such things as the Process Control Block (PCB), or a Time Queue element. It then becomes possible to define operators upon, for example, the PCB, which do exactly the appropriate manipulations. These could include the state transaction operations such as READY, WAIT, ACTIVE, ... Being identified as primitives, they too could then be implemented in micro code. It should be noted, that these forms of data structure manipulation, in general, are not complicated but consist of searching and bit manipulation, and these types of functions of course are very efficient in micro code.

A third set of data structures which concern the executive are those various synchronization primitives that are finding their way in higher order languages and which are required for real time processing. These primitives include, for example, Dijkstra's PV primitives, and Events and locks as used in HAL/S. Here is a case where there is an interaction between the HOL implementation and the executive. While the data structure is defined in the HOL, by its nature of being more global than a particular process, it must be handled by the executive. Again, a micro code implementation can make the implementation very efficient, and in this case it can also lend authority to the integrity of the operators by guaranteeing their uninterrupted execution.

Besides data structures, another area is the specification of interfaces from the applications program to the executive. The executive can be considered to be a series of routines that act upon the process state of the system. It allows changes in the states of processes. The executive also handles the interfaces to the outside world: interrupts and I/O processing.

If some of the HOL executive interfaces are simple executive routines (e.g. UPDATE PRIORITY) then it is possible that the whole function had become a single instruction, a micro routine. In this case, the interface indeed consists of executing one instruction which is the appropriate executive routine.

It is also possible to develop special executive HOL interfaces in order to minimize the amount of overhead required. This is possible, just as with the other language service routines (SIN(X),..) since all of the interfaces are known and well defined.

As with all routines, the decision of the encoding of an executive routine must depend upon its complexity, critical time requirements, and frequency of usage. With a refined definition of the required executive environment for a real time HAL/s, it would be possible to investigate those routines candidate for micro program encoding. The method for determining the best candidates is to instrumentate the actual execution of the system, and to determine the bottle necks. Perhaps in the future, this will become possible, with for example, the development of the Space Shuttle environment.

4.9 Conclusions and Recommendations

As a result of this implementation study, it is clear that a HALM is fairly simple to realize. A modified version of the MP instruction architecture [Mi 72] was investigated in detail and partially implemented on the B1700. While the B1700 is not designed to be a real time process control computer, its internal structure allows for convenient implementation of varying instruction architectures, and with the help of some specialized hardware, e.g. floating point unit, it would prove to be efficient in time as well as it is in space.

Further results of this study are the emphasis upon the importance of the instruction architecture addressing methodology; the requirements for actual HAL/S user statistics in order to both properly encode the instruction architecture operators and in order to help determine the most appropriate addressing mechanisms; and, an appreciation of the possibilities of being able to address any bit width without penalty, e.g. true precision specification in the HAL/S language itself.

While the results of this short study have been affirmitive and reassuring, it is desirable that several of the areas of investigation be developed further. Areas which can be considered to be of particular importance are as follows:

- HAL/S User Statistics

In order to both compare current instruction architectures and to develop future ones, it is necessary to know exactly how a language is used. Both Section 4.3 on addressing and Section 4.7 on statistics emphasized the requirements for usage statistics. It is only by this means that compact encoding of a logical instruction architecture into a physical representation of the instruction architecture may occur. Further, by knowing both the forms of operands and their characteristics distribution, it becomes possible to develop the appropriate, and most efficient addressing structure. User statistics also enable incremental improvements to the instruction architecture itself. Not only can encoding be made better, but appropriate operators can

be specified to optimize upon the correlation of actual occurance of several basic operators (e.g. A = A + 1;).

As the Space Shuttle program continues, statistics for HAL/S usage should become available. It is hoped that they will be used.

- Investigation of Various Address Structures

A thorough investigation of the various addressing structures available (absolute, indirect, lexical level-displacement, stack number-offset, base-displacement, sectors, banks, descriptors, ...) should be performed. In particular, it is of interest to know the time and space tradeoffs with respect to implementation complexity. In the aerospace environment, in particular, appropriate addressing would greatly decrease memory requirements.

- Develop Standardized Basic Operating System

It would be useful to have a virtual operating system specification which would define not only the HAL/S interfaces, but would indicate the allowable process interactions and time constraints. Such a specification would allow for deterministic and reproducible results of a complex of HAL/S programs regardless of the specific executive implementation or support processor.

- Variations and Stability in User Statistics and Resultant Design

It would be useful to determine how well a particular physical HALM realization acted with different sets of user statistics. Had the design been so tuned, that with a different set of usage characteristics, it became inefficient? Or, is it a relatively stabile design that varies but reasonably? This task would require both an analysis of how the design varies as statistics vary, and the actual gathering of several sets of statistics which do vary. Both the analytical and practical treatment of this task can be considered of interest.

- Full Implementation of a HALM

It would be desirable to actually complete a HALM implementation. This would afford assurance of actual design integrity and provide a facility for statistics validation. While it is relatively easy to develop memory size comparisons in abstraction, the actual execution of a HALM provides valid timing statistics and the micro routines provide the basis for the understanding of the timing. Actual execution on a microprocessor enables the determination of the timing bottle necks of an instruction architecture design.

The efficient hardware implementation of higher order languages is no longer in question. It is possible to orient the instruction architecture for the language which they are to execute, and to do so in an efficient manner. The principal issue for computing systems should be the development of languages which are truly oriented towards the problems to be solved.

4.10 Bibliography and References

Al72 Alexander, W.G., "How A Programming Language is Used", Technical Report CSRG-10, February 1972, Computer Systems Research Group, University of Toronto, Canada.

Bi70 Bingham, H.W., et al, "Microprogramming Manual for Interpreter Based Systems", TR 70-8, Burroughs Corporation, Paoli, Pennsylvania, November 1970.

Bo73 Boehm, B.W., "Software and Its Impact: A Quantitative Assessment", Datamation, May 1973.

Ca68 Carey, L.J., and Sturn, W.A., "Space Software at the Crossroads", Space and Aeronautics, December 1968.

Ch64 Chen, T.C., "The Overlap Design of the IBM System/360 Model 92 Central Processing Unit", FJCC, Part 2, 1964.

Co68 Corbato, F.J., "Sensitive Issues in the Design of Multi-Use Systems", MAC-M-383, Project MAC, December 12, 1968.

Co69 Corbata, F.J., "PL/I as a Tool for System Programming", Datamation, May 1969.

Co72 Colen, P., "Space Programming Language Machine Architecture Study", 2 Volumnes, prepared by CIRAD Corporation for USAF SAMSO, TR 72-117, 15 May 1972.

GR70 Graham, R.M., "Use of Higher Level Languages for Systems Programming", Technical Memorandum 13, Project MAC, September 1970, AD 711 965.

Hu70 Husson, S.S., "Microprogramming: Principles and Practices, Prentice Hall, 1970.

II71 Intermetrics, Inc., "HALMAT: An Intermediate Language of the First HAL Compiler", Revised, Intermetrics, Inc., 27 October 1971.

Ke70 Keeler, F.S., et al, Computer Architecture Study, USAF SAMSO, Report TR-240, October 1970. AD 720-798.

Kr70 Kerner, H., and Gellman, L., "Memory Reduction Through High Level Language Hardware", AIAA Journal, December 1970, pp. 2258-2264.

Kn70 Knuth, D.E., "An Empirical Study of Fortran Programs", Stanford University, Computer Science Department, Report No. CS-186, 1970, AD 715 513.

Lu72 Lutz, M.J., and Munthey, M.J., "A Micropogrammed Implementation of a Block Structured Architecture", Department Report 33-72-MU, Department of Computer Science, SUNY at Buffalo.

Mi72 Miller, J.S., Vandever, W.H., Stanten, S.F., Avakian, A.E., and Kosmala, A.L., "Engineering Study for the Functional Design of a Multiprocessor System", Final Report, Contract NAS 9-11745, Intermetrics, Inc., Cambridge, Mass., September 1972.

Nc71 Nanodata Corporation, "QM-1: Preliminary System Description", Nanodata Corporation, Williamsville, New York, October 1971.

Ni72 Nielsen, W.C., et al, "Aerospace HOL Computer", prepared by Logicon, Inc. for USAF AFAL, TR-72-292, 4 Volumnes, October 1972.

Pa70 Patzer, W.J., et al, "Aerospace System Implications of Microprogramming", in Air and Spaceborne Computers, Technicians Services, Slough, England, April 1970.

Ra69 Rakoczi, L.L., "The Computer with a Computer A Fourth Generation Concept", Computer Group News, March 1969.

Ro71 Rosin, R., et al, "An Environment for Research in Microprogramming and Emulation", State University of New York at Buffalo, Department of Computer Science, Dept. Report 5-71-M, 1971.

Sa72 Saponaro, J.A., et al, "Advanced Software Techniques for Data Management Systems", Volume 1, Final Report, Contract NAS 9-11778, Intermetrics, Inc., Cambridge, Mass., February 1972.

Su69 Sugimoto, M., "PL/I Reducer and Direct Processor", Proc. 4th National Conference, ACM, 1969.

Va71 Vandever, W.H., "Uncorrelated Notes from the 4th Annual Workshop on Microprogramming", Multi-processor Memo #07-71, Intermetrics, Inc., 20 September 1971.

Va72 Vandever, W.H., "AP-101 Micro Processor Description", AP-101 Memo #01-72, Intermetrics, Inc., 13 October 1972.

Va73 Vandever, W.H., "B1700 Micro Processor Description", B1700 Memo #01-73, Intermetrics, Inc., 3 December 1973.

Va73a Vandever, W.H., "AP-101 Instruction Set Modifications", Shuttle Memo #35-73, Intermetrics, Inc., 17 September 1973.

We71 Wersan, S.J., et al, Architectural Study for Advanced Guidance Computers, Part 2, prepared by CIRAD Corporation for USAF SAMSO, TR 71-6, February 5, 1971, AD 723 669.

Wi51 Wilkes, M.V., "The Best Way to Design an Automatic Calculating Machine", Manchester University, Computer Inaugural Conference, 1951, p. 8.

Wi69 Wichmann, B.A., "A Comparison of Algol 60 Execution Speeds", National Physical Laboratory Report CCU-3, January 1969.

Wi69a Wilkes, M.V., "The Growth of Interest in Micro-programming; A Literature Survey", Computing Surveys, Vol. 1, No. 3, September 1969.

Wi70 Wichmann, B.A., "Some Statistics from Algol Programs", National Physical Laboratory Report CCU-11, August 1970.

Wi71 Wichmann, B.A., "The Performance of Some Algol Systems", Proceedings of the IFIP Congress 1971.

Wi72 Wichmann, B.A., "Five Algol Compilers", Computer Journal, V. 15, No. 1, 1972.

Wi72a Wilner, W.T., "Design of the Burroughs B1700", FJCC, 1972, pp. 489-497.

Wi72b Wilner, W.T., "Burroughs B1700 Memory Utilization", FJCC, 1972, pp. 579-586.

Wi72c Wilner, W.T., "Microprogramming Environment in the Burroughs B1700", Compcon 72, Digest of Papers, pp. 103-106.

Wo72 Wortman, D.B., "A Study of Language Directed Computer Design", Technical Report CSRG-20, December 1972, Computer Systems Research Group, University of Toronto, Canada.

Chapter 4

Appendix 1

HAL Programming Example

Included in this appendix is an example of the code generated for HAL/S on both the IBM 360 and the IBM AP-101.

This program is representative of the size reduction which occurs when in a program code generated for the AP-101 versus the IBM 360. Several comments need be made in order to determine the relative sizes of the address fields and the opcode fields.

In the code generated for the IBM 360, there are inserted into the listings, several constants which are not directly needed for the execution, but are rather used by the Functional Simulator, SDL, and for debugging. These constants have been ignored in the total size count. But there are also some constants which are required in order to both set up the addressing environment for the routine and in order to bind it to other routines. These have been included in the instruction count as contributing to the address and total bit sizes. Figure 4.A1-1 shows a summary of the sizing as indicated in the listings. This sizing is broken down into the address field and opcode field portions of the total.

The listing for the AP-101 code generation does not break the instruction summary into the various formats, SRS and RS. Figure 4.A1-2 provides analysis of this break down, and then summarizes the program sizing. This again is broken down into the address field and opcode field portions of the total.

IBM 360 Code Generation Field Breakdown

- Constants to be counted in program size

LOCCTR =	000004	DC A	used for initial addressibility
=	000178	DC A	used to set up addressi-
through	000190	DC A	bility registers

= 6 "4 byte" Address Constants

There are 7 BALR instructions requiring address constants in order to link to the indicated routine

= 7 "4 byte" address constants

All other constants are assumed to be not relevant to the programs algorithm, and are for the Functional Simulator, SDL, or other usage.

- The break down of instruction count is therefore as follows:

	RR	(RX,RS,SI)	SS	DC
number of instructions	15	67	0	13

- Weighing these as indicated by Figure 4.3.1-1:

	RR	(RX,RS,SI)	SS	DC	Total
Address Field Bits	120	1608	0	416	2144
Opcode Field Bits	120	536	0	0	656
Total Bits	240	2144	0	416	2800

Figure 4.A1-1

AP-101 Code Generation Field Breakdown

- The output listing of the AP-101 code generation does not provide for a breakdown between the SRS and RS formats. This breakdown can be found by counting the instructions as given in the listing. The results of such an examination are here presented:

INSTRUCTURE	RR	SRS	RS	Total as Given in Listing
AH	0	0	1	1
AHI	0	0	5	5
BAL	0	0	8	8
BC	0	6	2	8
LH	0	1	4	5
LA	0	2	0	2
LH	0	18	7	25
LHI	0	0	7	7
STH	0	13	5	18
TOTAL	0	40	39	79

- Weighing these as indicated by Figure 4.3.1-2:

	RR	SRS	RS	TOTAL
Address field bits	0	440	858	1298
Opcode field bits	0	200	390	590
TOTAL BITS	0	640	1248	1888

Figure 4.A1-2

```
SPN STMT                                SOURCE                                  CURRENT_SCOPE
000000    1 M| CUBES:                                                           | CUBES
000000    1 M| PROGRAM;                                                        | CUBES
000000    2 M|    DECLARE INTEGER INITIAL(1),                                  | CUBES
000000    2 M|                I, IL, MINIM;                                     | CUBES
000000    3 M|    DECLARE IH INTEGER INITIAL(2);                               | CUBES
000000    4 M|    DECLARE INTEGER,                                             | CUBES
000000    4 M|                A, B, K;                                          | CUBES
000000    5 M|    DECLARE J ARRAY(12) INTEGER INITIAL(2#1, *);                 | CUBES
000000    6 M|    DECLARE S ARRAY(12) INTEGER INITIAL(2, 5, *);                | CUBES
000000    7 M|    DECLARE P ARRAY(12) INTEGER INITIAL(1, 8, *);                | CUBES
000000    8 M|    DO WHILE MINIM ¬= S ;                                        | CUBES
                S|                    I                                      |
000000    9 M|       A = I;                                                    | CUBES
000000   10 M|       MINIM = S ;                                               | CUBES
                S|             I                                            |
000001   11 M|       B = J ;                                                   | CUBES
                S|           I                                              |
000001   12 M|       IF J = I THEN                                            | CUBES
                S|          I                                               |
000001   13 M|          IL = IL + 1;                                          | CUBES
000001   14 M|       ELSE                                                     | CUBES
000001   14 M|          DO;                                                   | CUBES
000001   15 M|             IF J = 1 THEN                                      | CUBES
                S|                I                                         |
000001   16 M|             DO;                                               | CUBES
000001   17 M|                IH = IH + 1;                                    | CUBES
                E|                      3                                  |
000001   18 M|                P  = IH ;                                       | CUBES
                S|                 IH                                      |
000001   19 M|                J  = 1;                                         | CUBES
                S|                 IH                                      |
```

```
SRN STMT                                SOURCE                                  CURRENT SCOPE
000001   20 M|                  S  = P  + 1;                                    | CUBES
            S|                   IH   IH                                       |
000001   21 M|                END;                                             | CUBES
000002   22 M|             J = J + 1;                                           | CUBES
            S|              I   I                                             |
000002   23 M|             S = P + P  ;                                        | CUBES
            S|              I   I   J                                         |
            S|                      I                                         |
000002   24 M|          END;                                                   | CUBES
000002   25 M|       I = IL;                                                   | CUBES
000002   26 M|       K = I;                                                    | CUBES
000002   27 M|       DO WHILE K < IH;                                          | CUBES
000002   28 M|          K = K + 1;                                            | CUBES
000002   29 M|          IF S < S THEN                                         | CUBES
            S|             K   I                                             |
000002   30 M|             I = K;                                             | CUBES
000002   31 M|       END;                                                      | CUBES
000003   32 M|    END;                                                        | CUBES
000003   33 M|    WRITE(6) MINIM, A, B, I, J ;                                | CUBES
            S|                          I                                     |
         34 M| CLOSE CUBES;                                                   | CUBES
```

```
LOC  TR  CODE      LABEL     INSN   OPERANDS                SYMBOLIC OPERAND

000046           ST#1      EQU   *       TIME = 49
000000           $0CUBES   CSECT         ESDID= 0001
000000           CUBES     EQU   *
000000 47F0F010            BC    15,16(0,15)
000004 00000150            DC    A'00000150'
000008 0050                DC    X'0050'
000008 050C3C4C2           DC    X'050C3C4C2'
000000 C5C2                DC    X'C5C2'
000010 5830F004            L     11,4(0,15)
000014 9B6AB028            LM    6,10,40(11)
000018 92010000            MVI   0(13),1
00001C           ST#2      EQU   *       TIME = 0
000000           #BCUBES   CSECT         ESDID= 0003
000000 0001                DC    X'0001'
000002           ST#2      EQU   *       TIME = 0
000002 0001                DC    X'0001'
000004           ST#2      EQU   *       TIME = 0
000004 0001                DC    X'0001'
000006           ST#3      EQU   *       TIME = 0
000006 0002                DC    X'0002'
000008           ST#4      EQU   *       TIME = 0
000008           ST#4      EQU   *       TIME = 0
000008           ST#5      EQU   *       TIME = 0
00000C 0001                DC    X'0001'
000010 0001                DC    X'0001'
000012           ST#6      EQU   *       TIME = 0
000026 0002                DC    X'0002'
000028 0009                DC    X'0009'
00002A           ST#7      EQU   *       TIME = 0
00003E 0001                DC    X'0001'
000040 0008                DC    X'0008'
000042           ST#8      EQU   *       TIME = 45
00001C           $0CUBES   CSECT         ESDID= 0001
00001C           LBL#2     EQU   *
00001C 4890A000            LH    9,0(0,10)         I
000020 1A99                AR    9,9
000022 4820A004            LH    2,4(0,10)         MINIM
000026 4029C024            CH    2,36(9,10)        S
00002A 478F0104            BC    8,260(15,0)       C00104  LBL#3
00002E           ST#9      EQU   *       TIME = 18
00002E 4830A000            LH    3,0(0,10)         I
000032 4030A008            STH   3,8(0,10)         A
000036           ST#10     EQU   *       TIME = 18
000036 4849C024            LH    4,36(9,10)        S
00003A 4040A004            STH   4,4(0,10)         MINIM
00003E           ST#11     EQU   *       TIME = 18
00003E 4829A00C            LH    2,12(9,10)        J
000042 4020A00A            STH   2,10(0,10)        E
000046           ST#12     EQU   *       TIME = 33
000046 4859A00C            LH    5,12(9,10)        J
00004A 4950A000            CH    5,0(0,10)         I
00004E 476F0062            BC    6,98(15,0)        C00062  LBL#4
000052           ST#13     EQU   *       TIME = 26
000052 4860A002            LH    6,2(0,10)         IL
000056 4A60B044            AH    6,68(0,11)        H'1'
```

```
LOCCTR   CODE      LABEL    INSN   OPERANDS                SYMBOLIC OPERAND

00005A 4060A002             STH    6,2(0,10)               IL
00005E          ST#14       EQU    *      TIME = 17
00005E 47F00CC0             BC     15,192(15,0)            CCCCC0  LBL#5
000062          LBL#4       EQU    *
000062          ST#15       EQU    *      TIME = 43
000062 4890A000             LH     9,0(0,10)               I
000066 1A99                 AR     9,9
000068 41200001             LA     2,1(0,0)
00006C 4929A00C             CH     2,12(9,10)              J
000070 476F00A0             BC     6,160(15,0)             0000A0  LBL#6
000074          ST#16       EQU    *      TIME = 0
000074          ST#17       EQU    *      TIME = 26
000074 4830A006             LH     3,6(0,10)               IH
000078 4A30B044             AH     3,68(0,11)              F'1'
00007C 4030A006             STH    3,6(0,10)               IH
000080          ST#18       EQU    *      TIME = 180
000080 1A33                 AR     3,3
000082 41100003             LA     1,3(0,0)
000086 4800A006             LH     0,6(0,10)               IH
00008A 05EC                 BALR   14,12
00008C 00000000             DC     A(00000000)             ITCTHEI
000090 4013A03C             STH    1,60(3,10)              P
000094          ST#19       EQU    *      TIME = 10
000094 4023A00C             STH    2,12(3,10)              J
000098          ST#20       EQU    *      TIME = 18
000098 4A10B044             AH     1,68(0,11)              F'1'
00009C 4013A024             STH    1,36(3,10)              S
0000A0          ST#21       EQU    *      TIME = 0
0000A0          ST#22       EQU    *      TIME = 38
0000A0          LBL#6       EQU    *
0000A0 4890A000             LH     9,0(0,10)               I
0000A4 1A99                 AR     9,9
0000A6 4829A00C             LH     2,12(9,10)              J
0000AA 4A20B044             AH     2,68(0,11)              F'1'
0000AE 4029A00C             STH    2,12(9,10)              J
0000B2          ST#23       EQU    *      TIME = 30
0000B2 1A22                 AR     2,2
0000B4 4839A03C             LH     3,60(9,10)              P
0000B8 4A32A03C             AH     3,60(2,10)              P
0000BC 4039A024             STH    3,36(9,10)              S
0000C0          ST#24       EQU    *      TIME = 0
0000C0          ST#25       EQU    *      TIME = 18
0000C0          LBL#5       EQU    *
0000C0 4820A002             LH     2,2(0,10)               IL
0000C4 4020A000             STH    2,0(0,10)               I
0000C8          ST#26       EQU    *      TIME = 10
0000C8 4020A00C             STH    2,12(0,10)              K
0000CC          ST#27       EQU    *      TIME = 33
0000CC          LBL#7       EQU    *
0000CC 4820A00C             LH     2,12(0,10)              K
0000D0 4920A006             CH     2,6(0,10)               IH
0000D4 47AF0100             BC     10,256(15,0)            000100  LBL#8
0000D8          ST#28       EQU    *      TIME = 18
0000D8 4A20B044             AH     2,68(0,11)              F'1'
0000DC 4020A00C             STH    2,12(0,10)              K
```

```
LOCCTR   CODE      LABEL    INSN   OPERANDS                SYMBOLIC OPERAND

000010             ST#29    EQU    *       TIME = 49
000010 1A22                 AR     2,2
000012 48904000             LH     9,0(0,10)               I
000016 1A99                 AR     9,9
000018 48324024             LH     3,36(2,10)              S
00001C 49394024             CH     3,36(9,10)              S
000020 47AF00FC             BC     10,252(15,0)            0000FC  LBL#9
0000F4             ST#30    EQU    *       TIME = 18
0000F4 48404000C            LH     4,12(0,10)              K
0000F8 40404000             STH    4,0(0,10)               I
0000FC             ST#31    EQU    *       TIME = 10
0000FC             LBL#9    EQU    *
0000FC 47FF00CC             BC     15,204(15,0)            0000CC  LBL#7
000100             LBL#8    EQU    *
000100             ST#32    EQU    *       TIME = 10
000100 47FF001C             BC     15,28(15,0)             00001C  LBL#2
000104             LBL#3    EQU    *
000104             ST#33    EQU    *       TIME = 124
000104 41100006             LA     1,6(0,0)
000108 41000003             LA     0,3(0,0)
00010C 05EC                 BALR   14,12
00010E 00000000             DC     A'00000000'             ICINIT
000112 48004004             LH     0,4(0,10)               MINIM
000116 05EC                 BALR   14,12
000118 00000000             DC     A'00000000'             ICUT
00011C 48004008             LH     0,8(0,10)               A
000120 05EC                 BALR   14,12
000122 00000000             DC     A'00000000'             ICUT
000126 4800400A             LH     0,10(0,10)              B
00012A 05EC                 BALR   14,12
00012C 00000000             DC     A'00000000'             ICUT
000130 48004000             LH     0,0(0,10)               I
000134 05EC                 BALR   14,12
000136 00000000             DC     A'00000000'             ICUT
00013A 48904000             LH     9,0(0,10)               I
00013E 1A99                 AR     9,9
000140 48094000C            LH     0,12(9,10)              J
000144 05EC                 BALR   14,12
000146 00000000             DC     A'00000000'             ICUT
00014A             ST#34    EQU    *       TIME = 10
00014A 47F0C004             BC     15,4(0,12)
000150             #FCUBES  CSECT          ESDID= 0002
000150 47F0C174             BC     15,372(0,12)            STRACE
000154 00000000             DC     A'00000000'
000158 00000000             DC     X'00000000'
00015C 00000000             DC     X'00000000'
000160 00000000             DC     A'00000000'             CUBES
000164 0001                 DC     X'0001'
000166 0022                 DC     X'0022'
000168 00000000             DC     X'00000000'
00016C 00000000             DC     X'00000000'
000170 00000000             DC     X'00000000'
000174 00000000             DC     X'00000000'
000178 00000000             DC     A'00000000'
00017C 00000000             DC     A'00000000'
```

```
LOCCTR   CODE      LABEL   INSN   OPERANDS          SYMBOLIC OPERAND
000180 00000000           DC     A'00000000'
000184 00000000           DC     A'00000000'
000188 00000000           DC     A'00000000'
00018C 00000001           DC     X'00000001'
000190 00000000           DC     A'00000000'       @OCUBES
000194 0001               DC     X'0001'
                          END
```

FLD	POS	REF	FLAG	ADDRESS
0001	0007		08	000147
0001	0007		08	000137
0001	0007		08	000120
0001	0007		08	000123
0001	0007		08	000119
0001	0006		08	00010F
0001	0005		08	000080
0001	0002		08	000005
0002	0004		08	000191
0002	0003		08	000189
0002	0003		08	000185
0002	0003		08	000181
0002	0003		08	000170
0002	0003		08	000179
0002	0001		08	000161
0002	0003		08	000155

LOC	B	DISP	NAME
			UNTER CUBES
000000	10	000	I
000002	10	002	IL
000004	10	004	MINIM
000006	10	006	IH
000008	10	008	A
00000A	10	00A	B
00000C	10	00C	K
00000E	10	00C	J
000020	10	024	S
000022	10	03C	P

INSTRUCTION FREQUENCIES

INSN	COUNT
BALR	7
BR	8
STH	14
LA	4
BC	11
LH	24
CH	5
AH	6
L	1
MVI	1
LM	1

130 HALMAT OPERATORS CONVERTED

406 BYTES OF PROGRAM, 86 BYTES OF DATA

MAX. OPERAND STACK SIZE =4
END OPERAND STACK SIZE =0

```
 LOC    CODE      EFFAD     LABEL   INSN   OPERANDS              SYMBOLIC OPERAND

00000000                   ST#1    EQU    *
000000                     $0CUBES CSECT          ESDID= 0001
000000 0000                        DC     X'0000'
000000                     #DCUBES CSECT          ESDID= 0002
000000 0001                        DC     X'0001'
000001 0000                        DC     X'0000'
000001                     $0CUBES CSECT          ESDID= 0001
000001 000000                      ORG    *-1
00000000                   CUBES   EQU    *
000000 E8F3 0000                   LHI    0,0(3)                @0CUBES
000002 E9F3 0000                   LHI    1,0(3)                #DCUBES
000004 B9C8      00002             STH    1,2(0)
000005 EB50      00014             LA     3,20(0)
000006 B80C      00003             STH    3,3(0)
000007 E801      00000             LA     3,0(1)
000008 B804      00001             STH    3,1(0)
00000009                   ST#2    EQU    *
000002                     #DCUBES CSECT          ESDID= 0002
000002 0001                        DC     X'0001'
00000003                   ST#2    EQU    *
000003 0001                        DC     X'0001'
00000004                   ST#2    EQU    *
000004 0001                        DC     X'0001'
00000005                   ST#3    EQU    *
000005 0002                        DC     X'0002'
00000006                   ST#4    EQU    *
00000006                   ST#4    EQU    *
00000006                   ST#5    EQU    *
000006 000009                      ORG    *+3
000009 0001                        DC     X'0001'
00000A 0001                        DC     X'0001'
00000009                   ST#6    EQU    *
00000B 000015                      ORG    *+10
000015 0002                        DC     X'0002'
000016 0009                        DC     X'0009'
00000017                   ST#7    EQU    *
000017 000021                      ORG    *+10
000021 0001                        DC     X'0001'
000022 0008                        DC     X'0008'
00000023                   ST#8    EQU    *
000009                     $0CUBES CSECT          ESDID= 0001
00000009                   LBL#2   EQU    *
000009 9F09      00002             LH     7,2(1)                I
00000A 9011      00004             LH     5,4(1)                MINIM
00000B 95F5 E014 00014             CH     5,20(7,1)             S
00000D C4F7 004E 0005D             BC     4,78(3)               LBL#3
0000000F                   ST#9    EQU    *
00000F BF19      00006             STH    7,6(1)                A
00000010                   ST#10   EQU    *
000010 9EC9      00002             LH     6,2(1)                I
000011 9CF5 C014 00014             LH     4,20(6,1)             S
000013 BC11      00004             STH    4,4(1)                MINIM
00000014                   ST#11   EQU    *
000014 9CF5 C008 00008             LH     5,8(6,1)              J
000016 BD10      00007             STH    5,7(1)                B
```

```
 LOC   CODE      EFFAD   LABEL   INSN  OPERANDS        SYMBOLIC OPERAND

00000017                ST#12   EQU   *
000017 96F5 C008  00008         CH    6,8(6,1)        J
00001B DB14       0001F         BC    3,5(-1)         LBL#4
0000001A                ST#13   EQU   *
00001A 9F03       00003         LH    7,3(1)          IL
00001E B0F7 0001                AHI   7,1
000020 9F03       00003         STH   7,3(1)          IL
00000023                ST#14   EQU   *
000023 DFAC       0004A         BC    7,43(-1)        LBL#5
0000001F                LBL#4   EQU   *
0000001F                ST#15   EQU   *
00001F 9F09       00002         LH    7,2(1)          I
000020 E0F3 0001                LHI   5,1
000022 95F5 EC08  00008         CH    5,8(7,1)        J
000024 DB60       00030         BC    3,24(-1)        LBL#6
00000025                ST#16   EQU   *
00000025                ST#17   EQU   *
000025 9E15       00005         LH    6,5(1)          IH
000026 B0E6 0001                AHI   6,1
000029 9E15       00005         STH   6,5(1)          IH
00000029                ST#18   EQU   *
000029 9E48       00012         STH   6,18(0)
00002A EFF3 0003                LHI   6,3
00002C 9015       00005         LH    5,5(1)          IH
00002D E4F3 C000                BAL   4,0(3)          HTOTHEH
00002F 9C48       00012         LH    4,18(0)
000030 B0F5 8020  00020         STH   5,32(4,1)       P
00000032                ST#19   EQU   *
000032 9015       00005         LH    5,5(1)          IH
000033 ECF3 C001                LHI   4,1
000035 8CF5 A008  00008         STH   4,8(5,1)        J
00000037                ST#20   EQU   *
000037 9FF5 A020  00020         LH    7,32(5,1)       P
000039 B0F7 C001                AHI   7,1
00003B 8FF5 A014  00014         STH   7,20(5,1)       S
0000003E                ST#21   EQU   *
0000003D                ST#22   EQU   *
0000003D                LBL#6   EQU   *
00003D 9FC9       00002         LH    7,2(1)          I
00003F 90F5 E008  00008         LH    5,8(7,1)        J
000040 B0F5 C001                AHI   5,1
000042 80F5 E008  00008         STH   5,8(7,1)        J
00000044                ST#23   EQU   *
000044 9FF5 E020  00020         LH    6,32(7,1)       P
000046 86F5 A020  00020         AH    6,32(5,1)       P
000048 8FF5 E014  00014         STH   6,20(7,1)       S
0000004A                ST#24   EQU   *
0000004A                ST#25   EQU   *
0000004A                LBL#5   EQU   *
00004A 9D03       00003         LH    5,3(1)          IL
00004C 8DC9       00002         STH   5,2(1)          I
0000004C                ST#26   EQU   *
00004C 8D21       00008         STH   5,8(1)          K
0000004D                ST#27   EQU   *
0000004D                LBL#7   EQU   *
```

```
 LOC    CODE        EFFAD     LABEL   INSN   OPERANDS              SYMBOLIC OPERAND

0000AD 9021        00008             LH     5,8(1)                K
0000AF 9515        00005             CH     5,5(1)                IH
0000B1 D02C        00058             BC     5,11(-1)              LBL#8
00000C5C                     ST#28   EQU    *
0000B2 B0B5 0001                     AHI    5,1
0000B6 9021        00008             STH    5,8(1)                K
00000053                     ST#29   EQU    *
0000B9 9EC9        00002             LH     7,2(1)                I
0000BA 9EF5 A014   00014             LH     6,20(5,1)             S
0000BE 96F5 E014   00014             CH     6,20(7,1)             S
0000C2 C004        0005A             BC     5,1(-1)               LBL#9
00000C5S                     ST#30   EQU    *
0000C4 8009        00002             STH    5,2(1)                I
0000C6                       ST#31   EQU    *
0000005A                     LBL#9   EQU    *
0000C6 DF3A        00040             BC     7,14(-1)              LBL#7
00000C5P                     LBL#8   EQU    *
00000C5B                     ST#32   EQU    *
0000C8 C7F7 C854   00009             BC     7,84(3)               LBL#2
0000005D                     LBL#3   EQU    *
00000C5D                     ST#33   EQU    *
0000CC 5EF3 0006                     LHI    6,6
0000D0 5DF3 0003                     LHI    5,3
0000D4 E4F3 0000                     BAL    4,0(3)                ICINIT
0000D8 9011        00004             LH     5,4(1)                MINIM
0000DA E4F3 0000                     BAL    4,0(3)                HOUT
0000DE 9019        00006             LH     5,6(1)                A
0000E0 E4F3 0000                     BAL    4,0(3)                HOUT
0000E4 9010        00007             LH     5,7(1)                B
0000E6 E4F3 0000                     BAL    4,0(3)                HOUT
0000EA 9009        00002             LH     5,2(1)                I
0000EC E4F3 0000                     BAL    4,0(3)                HOUT
0000F0 9EC9        00002             LH     7,2(1)                I
0000F2 9DF5 E008   00008             LH     5,8(7,1)              J
0000F6 E4F3 0000                     BAL    4,0(3)                HOUT
00000074                     ST#34   EQU    *
0000FA E4F3 0000                     BAL    4,0(3)                STOP
00000023                     #DCUBES CSECT        ESDID= 0002
0000FE 00002D                        ORG    *+10
                                     END
```

```
   LOC    B DISP  NAME

          UNDER CUBES
 000002   1 002   I
 000003   1 003   IL
 000004   1 004   MINIM
 000005   1 005   IH
 000006   1 006   A
 000007   1 007   B
 000008   1 008   K
 000009   1 008   J
 000015   1 014   S
 000021   1 020   P

 INSTRUCTION FREQUENCIES

 INSN  COUNT
 STH     18
 LA       2
 PAL      8
 BC       7
 LH      25
 CH       5
 AH       1
 BC       1
 LHI      7
 AHI      5


 130 HALMAT OPERATORS CONVERTED

 118 HALFWORDS OF PROGRAM, 48 HALFWORDS OF DATA.

 MAX. OPERAND STACK SIZE              =5
 END  OPERAND STACK SIZE              =0
 NUMBER OF STATEMENT LABELS USED      =9
 MAX. STORAGE DESCRIPTOR STACK SIZE   =1
 END  STORAGE DESCRIPTOR STACK SIZE   =0
 NUMBER OF MINOR COMPACTIFIES         =1
 NUMBER OF MAJOR COMPACTIFIES         =0

 END OF HAL/S PHASE 2 JULY 17, 1974.  CLOCK TIME = 19:47:55.78
 TOTAL CPU TIME FOR PHASE 2          0:0:0.80
 CPU TIME FOR PHASE 2 SET UP         0:0:0.02
 CPU TIME FOR PHASE 2 GENERATION     0:0:0.31
 CPU TIME FOR PHASE 2 CLEAN UP       0:0:0.47
```

Chapter 4

Appendix 2

Initial MP Instruction Architecture Coding Example

Included in this appendix is both an example of the MP instruction architecture and a statement for statement comparison with respect to the AP-101 and IBM 360 code generation. The CUBES example given in Appendix 1 has been encoded with MP instruction architecture. The size comparison between the IBM 360, AP-101 and MP is given in Figure 4.A2-1.

It should be noted that in the encoding of the example, the data has been assumed to have been declared statically in order to be equivalent to the IBM 360 and AP-101 coding methodologies. Similarly, the method of executing the WRITE statement (St #33) has been made equivalent to the current IBM 360 and AP-101 methods even though others would be more efficient. Figure 4.A2-2 summarizes the relative code size for the three architectures. It also indicates the relative sizes when the address initialization (St #1) and I/O statement (St#33) are removed. This was done to remove the bias in favor of the MP instruction architecture which has a great deal less overhead in these particular functions.

Figure 4.A2-3 gives the opcode field and address field encoding for the CUBES example usage of the MP instruction architecture. This provides for convenient comparison to Figure 4.3.1-3 of Section 4.3.1 which contains the analogous breakdown for the IBM 360 and AP-101.

	MP	AP-101	IBM 360
ST#1(...7)	0	18	40
ST#8	10	12	18
ST#9	5	2	8
ST#10	7	8	8
ST#11	7	6	8
ST#12	8	6	12
ST#13	5	8	12
ST#14	3	2	4
ST#15	9	10	18
ST#16(17)	7	8	12
ST#18	10	18	20
ST#19	6	10	4
ST#20	11	12	8
ST#21(22)	11	14	18
ST#23	14	12	14
ST#24(25)	5	4	8
ST#26	5	2	4
ST#27	8	6	12
ST#28	7	6	8
ST#29	11	12	20
ST#30	5	2	8
ST#31	3	2	4
ST#32	3	4	4
ST#33	38	46	70
ST#34	1	4	4
TOTAL	199	234	346*

*NOTE: Instruction summary was incorrect in Appendix 1, HAL/S-360 listing. There were but 10 BC, thus, Appendix 1 counted 4 bytes too much.

Comparison of Code Sizes for CUBES (refer to Appendix 1)

Figure 4.A2-1

Relative Program Sizes

CUBES

- Total Program Size Comparison

	IBM 360	AP-101	MP
Total Program Bytes	346	234	199
% Compared to IBM 360	100%	67.4%	57.5%

- Program Size Comparison with I/O and Environment Initialization Removed

	IBM 360	AP-101	MP
Program Bytes Excluding ST#1 & ST#33	226	170	161
% Compared to IBM 360	100%	75.3%	71.3%

Figure 4.A2-2

	Operators	Operands	LTS4	LTS10
Number of Instructions	56	60	11	6

Weighing these as indicated by Figures 4.5.3-3 and 4.5.3-4:

	Operators	Operands	LTS4	LTS10	Total	%
Address field bits	0	780	55	66	901	56.6%
Opcode field bits	448	180	33	30	691	43.4%
Total bits	448	960	88	96	1592	100%

Bit Distribution in the MP Instruction Architecture

Figure 4.A2-3

```
00020      CUBES ENCODED USING THE INITIAL HP INSTRUCTION ARCHITECTURE
00030
00040  ST#1 EQU *
00050  ST#8 EQU *
00060 LBL#2 EQU *
00070       GET I
00080       GET S
00090       GET MINIM
00100       EQUL
00110       LTS10 150       LBL#3
00120       JOT
00130 ST#9  EQU *
00140       GET I
00150       ADR A
00160       STD
00170 ST#10 EQU *
00180       GET I
00190       GET S
00200       ADR MINIM
00210       STD
00220 ST#11 EQU *
00230       GET I
00240       GET J
00250       ADR B
00260       STD
00270 ST#12 EQU *
00280       GET I
00290       DUPL
00300       GET J
00310       EQUL
00320       LTS4 *          LBL#4
00330       JOT
00340 ST#13 EQU *
00350       LTS4 1
00360       ADD
00370       ADR  IL
00380       STD
00390 ST#14 EQU *
00400       LTS10 68        LBL#5
00410       JMP
00420 LBL#4 EQU *
00430 ST#15 EQU *
00440       GET I
00450       GET J
00460       LTS4 I
00470       EQUL
00480       LTS10 34        LBL#6
00490       JOF
00500 ST#16 EQU *
00510 ST#17 EQU *
00520       GET IH
00530       LTS4 1
00540       ADD
00550       ADR IH
00560       STD
00570 ST#18 EQU *
00580       GET IH
00590       DUPL
00600       DUPL
00610       DUPL
00670       MUL
00680       MUL
00681
```

```
00683
00684
00685
00686
00690          ADRE P
00700          STD
00710 ST#19    EQU *
00720          LTS4 1
00730          GET  IH
00740          ADRE J
00750          STD
00760 ST#20    EQU *
00770          GET IH
00780          GET P
00790          LTS4 I
00800          ADD
00810          GET IH
00820          ADRE S
00830          STD
00840 ST#21    EQU *
00850 ST#22    EQU *
00860  LBL#6   EQU *
00870          GET I
00880          GET J
00890          LTS4 1
00900          ADD
00910          GET I
00920          ADRE J
00930          STD
00940 ST#23    EQU *
00950          GET I
00960          DUPL
00970          DUPL
00980          GET J
00990          XCH
01000          GET P
01010          ADD
01020          XCH
01030          ADRE S
01040          STD
01050 ST#24    EQU *
01060 ST#25    EQU *
01070  LBL#5   EQU *
01080          GET IL
01090          ADR I
01100          STD
01110 ST#26    EQU *
01120          GET I
01130          ADR K
01140          STD
01150 ST#27    EQU *
01160  LBL#7   EQU *
01170          GET K
01180          GET IH
01190          GREQ
01200          LTSIO 26          LBL#8
01210          JOT
01220 ST#28    EQU *
01230          GET K
01240          LTS4 1
01250          ADD
01260          ADR K
01270          STD
01280 ST#29    EQU *
```

```
01291
01292
01293
01294
01295
01300          GET S
01310          GET I
01320          GET S
01330          GREQ
01340          LTS4 5              LBL#9
01341          JOT
01350 ST#30    EQU *
01360          GET K
01370          ADR I
01380          STD
01390 ST#31    EQU *
01400  LBL#9   EQU *
01410          LTS10 -34           LBL#7
01420          JMP
01430 LBL#8    EQU *
01440 ST#32    EQU *
01450          LTS10 -160          LBL#2
01460          JMP
01470 LBL#3    EQU *
01480 ST#33    EQU *
01490          MKS
01500          LTS4 3
01510          LTS4 6
01520          ADR IOINIT
01530          ENTR
01540          MKS
01550          GET MINIM
01560          ADR HOUT
01570          ENTR
01580          MKS
01590          GET A
01600          ADR HOUT
01610          ENTR
01620          MKS
01630          GET B
01640          ADR HOUT
01650          ENTR
01660          MKS
01670          GET I
01680          ADR HOUT
01690          ENTR
01700          MKS
01710          GET I
01720          GET J
01730          ADR HOUT
01740          ENTR
01750 ST#34    EQU *
01760          EXIT
```

5. CONCLUSIONS AND RECOMMENDATIONS

5.1 Conclusions - HAL

A. All language features of the HAL language specification were implemented in the 360 version of the compiler. This permitted a thorough evaluation of the language prior to its selection for usage in the Space Shuttle.

B. Many implementation problems were solved and the way was paved for the inclusion of these solutions into Space Shuttle compilers. This permitted the rapid and timely deliver of HAL/S compilers.

C. The investment in compiler implementation and the tailoring of the compiler to the machine architecture produced a number of positive proposals that resulted in adoption into the Space Shuttle F_C instruction repetoire.

D. The method and procedures for rehosting the HAL compiler were demonstrated by the transfer of HAL 360 to the 1108.

E. A system of language control, compiler change control and modification was developed that was to prove useful for Space Shuttle work.

F. In general, the RTOP investment in language and compiler activity provided many returns that are being reaped in the Space Shuttle program.

5.2 Recommendations - HAL

A. The HAL language was developed and evaluated. It has now been adopted for Space Shuttle usage. NASA should make wide usage of the language. This will provide a common language of communication across all levels of NASA software development, it will increase programmer productivity, and provide software transferability.

B. Broad usage of HAL will require a unified method of language control to insure transferability and reduce maintenance and compiler change costs.

C. A unified method of compiler implementation should be studied and the best method adopted by NASA consistent with their objectives of centralization of compiler generation and maintenance, transferability, and language control.

5.3 HALM Recommendations and Conclusions

The recommendations and conclusions resulting from the HAL machine design effort are contained within Chapter 4 in order to provide a section of the final report that can be self contained. These same recommendations and conclusions are repeated here for completeness.

As a result of this implementation study, it is clear that a HALM is fairly simple to realize. A modified version of the MP instruction architecture [Mi 72] was investigated in detail and partially implemented on the Bl700. While the Bl700 is not designed to be a real time process control computer, its internal structure allows for convenient implementation of varying instruction architectures, and with the help of some specialized hardware, e.g. floating point unit, it would prove to be efficient in time as well as it is in space.

Further results of this study are the emphasis upon the importance of the instruction architecture addressing methodology; the requirements for actual HAL/S user statistics in order to both properly encode the instruction architecture operators and in order to help determine the most appropriate addressing mechanisms; and, an appreciation of the possibilities of being able to address any bit width without penalty, e.g. true precision specification in the HAL/S language itself.

While the results of this short study have been affirmitive and reassuring, it is desirable that several of the areas of investigation be developed further. Areas which can be considered to be of particular importance are as follows:

A. HAL/S User Statistics

In order to both compare current instruction architectures and to develop future ones, it is necessary to know exactly how a language is used. Both Section 4.3 on addressing and Section 4.7 on statistics emphasized the requirements for usage statistics. It is only by this means that compact encoding of a logical instruction architecture into a physical representation of the instruction architecture may occur. Further, by knowing both the forms of operands and their characteristics distribution, it becomes possible to develop the appropriate, and most efficient addressing structure. User statistics also enable incremental improvements to the instruction architecture itself. Not only can encoding be made better, but appropriate operators can be specified to optimize upon the correlation of actual occurance of several basic operators (e.g. A = A + 1;).

As the Space Shuttle program continues, statistics for HAL/S usage should become available. It is hoped that they will be used.

B. Investigation of Various Address Structures

A thorough investigation of the various addressing structures available (absolute, indirect, lexical level-displacement, stack number-offset, base-displacement, sectors, banks, descriptors, ...) should be performed. In particular, it is of interest to know the time and space tradeoffs with respect to implementation complexity. In the aerospace environment, in particular, appropriate addressing would greatly decrease memory requirements.

C. Develop Standardized Basic Operating System

It would be useful to have a virtual operating system specification which would define not only the HAL/S interfaces, but would indicate the allowable process interactions and time constraints. Such a specification would allow for deterministic and reproducible results of a complex of HAL/S programs regardless of the specific executive implementation or support processor.

D. Variations and Stability in User Statistics and Resultant Design

It would be useful to determine how well a particular physical HALM realization acted with different sets of user statistics. Had the design been so tuned, that with a different set of usage characteristics, it became inefficient? Or, it is a relatively stabile design that varies but reasonably? This task would require both an analysis of how the design varies as statistics vary, and the actual gathering of several sets of statistics which do vary. Both the analytical and practical treatment of this task can be considered of interest.

E. Full Implementation of a HALM

It would be desirable to actually complete a HALM implementation. This would afford assurance of actual design integrity and provide a facility for statistics validation. While it is relatively easy to develop memory size comparisons in abstraction, the actual execution of a HALM provides valid timing statistics and the micro routines provide the basis for the understanding of the timing. Actual execution on a microprocessor enables the determination of the timing bottle necks of an instruction architecture design.

The efficient hardware implementation of higher order languages is no longer in question. It is possible to orient the instruction architecture for the language which they are to execute, and to do so in an efficient manner. The principal issue for computing systems should be the development of languages which are truly oriented towards the problems to be solved.

Appendix A

Selected HAL Memos Describing HAL Compiler Releases

10/71	Operation Status of HAL on the IBM 360
03/72	HAL Specification Change Notice #1 (HAL I/O)
15/72	Operational Status of HAL/360 Version 360-6
19/72	Release of HAL Version 360-7

HAL USER'S MEMO #10-71

TO: Jack Garman

FROM: Dan Lickly

DATE: 19 October 1971

SUBJECT: Operation Status of HAL on the IBM 360

The utility of the HAL compiler on the 360 is constantly increasing as more capabilities are added to the system and previous flaws corrected. Consequently, any discussion of operational features must reference a version or point-in-time. Two versions are of interest at this time; first, the one made in early September and in use at MSC and second, the one that will be installed at MSC at the next opportunity - approximately 1 November 1971. The characteristics of both are described below.

A. HAL-360 (1 Nov. 1971)

The following items have not yet been implemented in the HAL compiler.

Pass 1

1. The linear array functions: MAX, MIN, SUM, PROD, POLY.
2. Compiler directive cards; viz., INCLUDE.
3. The character constant, CHAR.
4. Output listing cosmetics; e.g., stars, bars, and brackets are incomplete.
5. Real-time control statements are recognized, but not processed further.

Pass 2

1. Update blocks or tasks
2. Locking
3. Precision modifiers
4. READALL

5. Structure assignment, comparison, and parameter passing.

6. The following built-in functions: INDEX, LJUST, RJUST, SIGNUM, ARCCOSH, ARCSINH, ARCTANH, ADJ, MAX, MIN, SUM PROD, POLY, MOD

7. The following shaping functions: BIT, BIT$_{@}$ CHAR, CHAR$_{@}$, SUBBIT.

8. Advanced bit string features; e.g., bit user functions, bit conditionals, and arrayed bit arguments to procedures or functions.

9. No shaping functions with arrayness, nor shaping functions with arrayed arguments.

10. File operations

11. Run-time checks of subscripts & other out-of-limit violations.

B. HAL-360 (10 September 1970) All of the above plus the following:

Pass 1

1. Real-time control statements are not recognized and the key word not reserved; e.g., SCHEDULE, WAIT, UNTIL, SIGNAL, etc.

2. The reserved bit constants: FALSE, ON, OFF.

3. The CHARACTER conversion function is spelled CHAR.

4. DO FOR loops with negative increments.

5. Nested repeat expressions in INITIAL lists.

6. Bit constants may not have repeat numbers.

7. EOF is a key word denoting end-of-file.

8. The optional comma separating the factored attribute list from the first variable name in a factored DECLARE statement is not optional, but mandatory.

Pass 2

1. All bit string operations

2. Multiple invocation of the same function at different levels of nested functions.

3. The built-in function, LENGTH.

4. Arguments of procedures for user function may not be expressions.

5. DO groups operate unreliably under certain circumstances.

C. Implementation Dependent Restrictions

The following limitations are imposed on the current implementation of HAL on the 360.

1. Vectors limited to a length of 32 elements.

2. Matrices limited to 16-by-16.

3. Integers are 32-bit two's complement numbers.

4. Bit strings limited to 32 bits.

5. Varying character strings limited to 255 characters.

6. The number of calls to any one procedure or user function is limited to 50.

7. The number of cases in a 'do case' is limited to 40.

8. The number of groups in a grouped DO FOR is limited to 40.

9. The READ statement only handles 80 column input, through one channel only.

10. The WRITE statement only handles 133 columns output through one channel only.

11. Arguments of procedures or user functions which are arrayed expressions are not allowed.

12. No precision or type conversions are made on arguments of procedures or user functions, nor upon the returns of user functions.

13. If a HAL program is to be called by another then only the first 5 characters of the name are used. The underscore (_) may not be used in the first 5 characters of a program name under any circumstances.

14. The REPLACE statement has a size limitation; the string replacing the identifier may not be more than 256 characters long, nor, in the case of nested REPLACE's may the sum of the string to be added and the part of the old string to the right of the insertion be more than 256 characters.

HAL USER MEMO #03-72

TO: Distribution

FROM: P. M. Newbold

DATE: 1 February 1972

SUBJECT: HAL Specification Change Notice #1 (HAL I/O)

The following changes to and clarifications of the HAL I/O specifications are hereby made. They are implemented in the current version of the HAL 360 compiler. They cover the following areas:

a. characterization of stream-oriented (sequential) storage devices

b. commanding the movement of their read- or write-mechanisms

c. structure of the input data stream

d. effect of the READALL statement

e. type conversion during READ statements.

1. Storage devices are divided into two classes, paged and unpaged. A paged device may be visualized as a book, control functions being used to move the device-mechanism from page to page as well as to position the device-mechanism on the page. An unpaged device may be visualized as a long strip of teletype paper; control functions being used to position the device-mechanism anywhere on the strip.

2. The device-mechanism of any paged device may be commanded by the following control functions, whether they occur in, READ, READALL or WRITE statements:

SKIP(<p>)	TAB(<p>)
LINE(<p>)	COLUMN(<p>)
PAGE(<p>)	

where <p> is an integer or scalar expression (the latter being rounded). The device-mechanism of any unpaged device may be commanded by any of the above control func-

tions except PAGE(<p>) which is meaningless. The operation of a physical device may impose bounds on the acceptable values of <p>.

3. The data fields in the input data stream may be delimited by 1 or more spaces, by a comma, or by a comma and 1 or more spaces. No delimiter is required to data field if one of them is a character data field (i.e. enclosed in quotes). A semicolon, as well as delimiting data fields, also serves to terminate the read operation. If n commas appear between 2 data fields, or if n-1 commas appear between a data field and a trailing semicolon, then n-1 null data fields are said to exist at that place in the data stream. The null field has the effect of leaving the value of the READ list element being processed at that point unchanged.

   Example:

```
X = 0.5;
READ(CARDS) Y,X,Z;
```

   input:

```
0.753,  ,  0.0157
```

   X is left at 0.5.

4. The READALL statement causes different actions to take place, depending on whether the character string list elements have the fixed or varying attribute. If the character string is fixed, it will be completely filled from the input data stream, as many lines being traversed by the device-mechanism as required. If the character string is varying then one of two courses of action are taken. If the maximum length of the character string is greater than the (remaining) length of the current line, then the character string takes on that length, and is filled with the remainder of the line. Otherwise the character takes on its maximum length and is filled from the input stream as if it were a fixed character string.

5. A run-time error message is given if a data field enclosed in quotes is read on input to a scalar, integer or vector/matrix variable, or if a data field not enclosed in quotes is read on input to a character variable. If a scalar data field with a fractional part is read into an integer or bitstring, rounding will occur.

Extra-Lingual Features

In the current implementation of HAL-360 the association of actual I/O devices with HAL I/O channels is made by the use of the DEVICE compiler directive [1] in conjunction with OS360 JCL. Two types of device are supposed to exist:

1. PRINT device: for output only; not input compatible; paged.

2. non-PRINT device: for output and input, possibly in the same program; unpaged.

An expanded explanation of the way in which physical I/O devices are allocated will appear shortly in a future memo.

---

[1] HAL USER MEMO #01-72, The HAL DEVICE directive, R.E. Kole.

HAL User Memo # 15

TO: Distribution

FROM: HAL Staff

DATE: 28 April 1972

SUBJECT: Operational Status of HAL/360 Version 360-6

The purpose of this memo is to describe the HAL/360 compiler release number 6 as it is currently being installed at the RTCC at MSC. This release represents a snapshot of the HAL system as of March 15, 1972. Topics covered in this memo include:

- Functional Restrictions of the HAL Language Specification
- Compiler implementation dependent restrictions
- Summary of new features

With the exception of the Functional Restrictions, all improvements and extensions of the compiler's capabilities mentioned in previous memoranda also apply to the new release.

1. Functional Restrictions of the HAL Language Specification

The following restrictions on the use of HAL's full specification remain in the current release. They are divided into two categories based upon the pass of the compiler which pertains to the restriction.

A. Phase 1 Restrictions

1. The linear array functions MAX, MIN, SUM, PROD, and POLY are not recognized.
2. The INCLUDE compiler directive and corresponding library facilities have not been implemented.
3. The character constant form of CHAR' ... ' has not been implemented --- and may be dropped from the language specification.
4. Houston/MSC only: Lack of a "TN" print chain in RTCC forces the output writer to use an "up-arrow" (↑) to replace brackets ([ ]) in the annotation

of arrays, and "integral signs" ($\int$) in place of braces ({ }) to yield annotation for structures.

B. Phase 2 Restrictions

1. Update blocks and the control of data sharing among programs via the LOCKTYPE attribute have not been implemented.

2. Structure operations of assignment, comparison and parameter passing should not be attempted.

3. The following list of built-in functions:

INDEX	LJUST	RJUST	SIGNUM
ARCCOSH	ARCSINH	ARCTANH	ADJ
MAX	MIN	SUM	PROD
POLY	MOD		

4. The following bit and character string shaping functions

BIT	$BIT_{@}$	SUBBIT
CHAR	$CHAR_{@}$	

5. Certain rules regarding the use of Shaping Functions have now been defined. Refer to HAL USER MEMO #8-72 for details of these rules.

6. With two exceptions, there are no run time checks of limit condutions connected with program control. The exceptions are the detection of compool size discrepancies and the situation of program control flowing to a FUNCTION procedure's CLOSE statement. Various run time error conditions relating to data integrity and I/O operations are detected.

7. The optional comma separating the factored attribute list from the first variable name in a factored DECLARE will produce a warning message if omitted; however, omission will not affect the validity of the compiled program.

2. Compiler Implementation Dependent Restrictions

The implementation restrictions summarized in the previous release memo, HAL Houston User Memo #02-72 still apply to the present compiler without modification.

3. Summary of New Features

A. Output Writer: The HAL output writer feature has been upgraded to its full specification in the current release. The improvements to this routine in the Phase 1 program of the compiler are as follows:

1. Automatic indentation algorithms have been implemented. As a result, a standard format which is quite readable now is created by the output writer. The block structure and logical organization of such language features as DO statements and IF statements is now quite recognizable in the standard form produced.

2. The previous deficiency of ignoring embedded PL/1 form comments ("/* ... */") has been corrected. All embedded comments which occur prior to the semicolon which terminates a statement (beginning with the first comment in the statement if any) are collected, stripped of the /*...*/ delimiters, catenated together and turned into a single comment which starts with the "/*" delimiter and ends with the "*/" delimiter and is placed following the terminating semicolon of the statement. A word of caution: comments which are embedded in E or S lines of a statement are still ignored.

3. Certain cosmetic features have been added or improved in the listings of the compiled program. The principal example is a much improved LISTING2 input image format.

B. REAL Time Facility: The current release of the HAL system supports a simulated real time environment. In this simulation, an external file of events (stimuli) is maintained, which is used together with application program-internal scheduling of events to run examples of real time systems. New features of the Phase 2 code generation support the following statements:

SCHEDULE...	WAIT...
TASK...	TERMINATE...
SIGNAL...	

Additionally, the EVENT variables used in a real time situation may now be declared in the HAL language DECLARE statement using the keyword EVENT as an attribute. For a disucssion of the details of this new feature, consult HAL User Memo #9-72.

C. The use and applicability of shaping functions for conversion between data types in the HAL language have been extended in several ways. Refer to HAL User Memo #8-72 for a discussion of the current ground rules of shaping function use.

D. Precision modifiers have now been implemented.

E. Literals: The literal processing of Phase 1 now implements an improved algorithm for maintaining a list of literals used in a given program. There is still a limit of 100 unique numeric literals in any program, but this limit restricts only literals occurring in executable statements; i.e., literals in declare contexts are no longer considered in checking the literal limit of 100.

F. Initialization: In the previous versions of the compiler, there was a stacking limitation on the number of literal values which could be coded in the literal lists of HAL DECLARE statements. This limitation typically was in the 50-70 range depending upon the context of the program in being compiled. The current release of the compiler employs a new strategy which alleviates this restriction to a great extent. The limitation now is that no more than 450 arithmetic values may be initialized within the initial lists of a single HAL statement. This limitation is independent of program context.

G. Dump and trace facilities: This new release of the compiler incorporates a termination dump which may be used on program failure in the execution of non-real time jobs, and a trace facility which is usable in both real-time and non-real time situations. The dump may be used at termination or at selected points in the program, giving a formatted listing of the user defined variables and identifiers. The trace may apply in general to a whole program, or may be specified for a specified range of statements in a program --- in either case, the trace consists of a formatted message notifying the user of the current position in the program. In a future release, facility will be incorporated for both TRACE and DUMP user aids in either real time or static modes of operation of a program. HAL User Memo #12-72 gives a full description of the DUMP and TRACE facilities as they now stand.

H. The following Specification Change Proposals which were detailed in the specified Intermetrics HAL User Memos have been implemented. These memos may now be considered as updates to the HAL specifications:

1)	User Memo #3	HAL I/O
2)	User Memo #6	The TIME keyword
3)	User Memo #7	Real Time Control
4)	User Memo #9	Real Time Control
5)	User Memo #11	Compool Initialization

HAL USER MEMO #19-72

TO: Distribution

FROM: R. E. Kole

DATE: 12 June 1972

SUBJECT: Release of HAL Version 360-7

This memo describes the status of the HAL/360 compiler as of the release of Version 360-7 to M.I.T. on June 13, 1972.

1. Functional Restrictions of the HAL Language Specification

The following restrictions on the use of the full HAL language remain in Version 360-7.

A. Phase 1 Restrictions

(1) The linear array functions MAX, MIN, SUM, PROD and POLY are not implemented.

(2) The character constant form of CHAR'...' is not implemented.

(3) The EXCLUSIVE attribute of procedures is not implemented.

(4) ACCESS rights for control of COMPOOL data are not implemented.

B. Phase 2 Restrictions

(1) Update blocks and control of shared data are not implemented.

(2) Structure operations are undefined.

(3) The following built-in functions are not implemented:

INDEX	LJUST	RJUST	SIGNUM
ARCCOSH	ARCSINH	ARCTANH	ADJ
MAX	MIN	SUM	PROD
POLY	MOD		

(4) The following bit and character string shaping functions are not implemented.

BIT BIT@ SUBBIT

CHAR CHAR@

(5) HAL User Memo #8-72 is still applicable (defines rules for use of shaping functions).

(6) Run time limit checks are only made for the following situations:

(a) Attempted execution of CLOSE of a function.

(b) Mismatching of COMPOOL sizes of programs that invoke each other.

2. Compiler Implementation Dependent Restrictions

No additions or deletions to the list of implementation restrictions have been made. Therefore, those restrictions summarized in the previous release memo still apply.

3. Summary of New Features

A. Output Writer

Small improvements have been made to the output writer portion of Phase 1. These improvements include the correction of errors in the expansion of single line input to multi-line output and the correction of improper indenting in some forms of the DECLARE statement.

Also, REPLACE items are underlined in the listing to make their use clear.

B.

The INCLUDE compiler directive has been implemented. The form of the directive is that proposed in HAL User Memo #13-72. Use of the INCLUDE directive causes source code to be read from a data set defined on a JCL card of the following form:

```
//INCLUDE DD DISP=SHR, DSN=name,...
```

where name defines a partitioned data set. The partitioned data set directory is searched for the member name specified in the INCLUDE directive and that member is included if it is found. All other situations result in an error message.

C. User Aids

Extensive changes and improvements have been made to the listing produced by the compiler. These changes involve the production of a block summary at the CLOSE of each block of a program and a completely reordered symbol table listing. These improvements are fully explained in HAL User Memo #17-72.

D. Error Recovery

Full error recovery facilities are now available in non-REALTIME including use of the ON ERROR and SEND ERROR statements.

Also, HAL error messages are now produced for all errors and HAL error summary is given at termination of a run.

The full functional description of the HAL Error Processor is given in HAL User Memo #18-72.

The form of the SEND ERROR statement has been changed to correspond to the form defined in HAL User Memo #16-72.