AN ARITHMETIC LOGIC UHI7 FOR AH ISP SIMULATOR

Mark *tfirsch*

December, ",932

DflC-O1-12-32

AN ARITHMETIC LOGIC UNIT
FOR AN ISP SIPJ1ULATOR

M.S. Project Report

MarkHirsch

Department of Electrical Engineering
Carnegie Mellon University

13 August 1982

Table of Contents

List of Figures

List of Tables

1. Abstract

The execution time for simulating an ISP description can be improved a maximum of 5 to 20 percent when the ALU portion of the simulator is implemented in hardware. Factors which affect the degree of improvement are the level of abstraction of the model, the complexity of the ISP description, and programming style. Two hardware implementations are examined in detail. The first is based on the AMD2900 family of bit slice microprocessors and the second is based on a VLSI microprocessor chip designed at Carnegie Mellon University specifically for the Register Transfer Machine (RTM).

2. Introduction

ISPS is a hardware description language in which a large class of computers and other digital systems can be described. ISPS has its origin in the Instruction Set Processor (ISP) notation [Bell and Newell 71] and is oriented toward the description of digital computers.

Simulation of ISPS descriptions has a variety of applications. First, it would be particularly useful in the hardware design phase of a digital computer. If the target machine could be simulated, problems associated with its design could be uncovered and corrected before construction begins, thus saving time and money. Second, insight into new architectures could be gained without the expense of construction. Third, software could be developed and tested for the target machine before a machine is actually available.

To date, work at Carnegie Mellon University has focused on the design and development of a Register Transfer Machine (RTM), specifically designed to simulate ISP descriptions [Barbacci 81]. This machine is currently implemented entirely in software [Barbacci 80a]. There are certain problems, however, with such an implementation. This software implementation is slow because it adds an extra level of interpretation for each ISPS instruction. Small programs can be satisfactorily simulated, but programs of moderate size take a long time to execute. For example, a program running on a simulation of a microprocessor such as the Intel 8080 would take about five thousand times longer than it would on the real machine. And for simulations of larger machines, the execution speed ratio becomes even greater. It would take about ten thousand times longer for the same program to run on a simulation of a PDP11 as opposed to the real machine [Barbacci 82].

In an effort to reduce the inefficiency inherent in the simulator, a project was recently undertaken at Carnegie Mellon University whose goal is to implement an ISP emulator in hardware. The project is divided into two parts: the ALU and the rest of the machine. The ALU includes all functions necessary to execute the arithmetic and logical instructions of the RTM while the rest of the machine supplies the user with the capability to interactively monitor the behavior of the target machine.

The intent of this project is to evaluate the issues involved in the ALU portion of a machine that executes the code generated from ISPS descriptions. First, the software simulator will be examined to determine where most of its time is spent and what the potential speedups are that could be realized from a hardware ALU implementation. This analysis consists of two parts. The first is a determination of the percentage of Arithmetic/Logic operations that are executed by the RTM. The second is a determination of the amount of time it takes to execute the various RTM operations.

Second, features peculiar to the RTM ALU will be discussed as well as algorithms for implementing these features. The ALU can perform arithmetic in either two's complement, one's complement, signed magnitude or unsigned magnitude number systems. It can perform alignment of operands and extract fields from within a larger field. In addition, the ALU can operate on operands of any word length.

Third, two implementations for the hardware ALU are studied. They are as follows: one based on the AMD2900 family of bit slice microprocessors and one based on a VLSI microprocessor chip designed at Carnegie Mellon University specifically for the RTM.

3. Simulating ISP Descriptions

ISPS is a hardware description language and can be used to describe virtually any computer. The ability to simulate a target machine once it has been described in ISPS greatly enhances the usefulness of the language as a design tool. Such a facility has been developed at Carnegie Mellon University and has been given the name, "the ISPS simulator".

The simulator provides the user with the capability to interactively monitor the behavior of a target machine. This is extremely useful in debugging both the hardware and software of the target machine. For example, the user can instruct the simulator to set break points at key statements in the target machine program and halt the simulation in order to examine various target machine parameters (i.e., register contents, memory contents). The ISPS simulator manual contains complete user oriented documentation for the simulator features [Barbacci 80a].

The simulator is currently implemented in software. Versions in several different languages and for several different machines exist and other versions are being developed at this time. The heart of the simulator is the Register Transfer Machine (RTM). (The specifics of the RTM will be described in later sections.) ISP descriptions are translated into code for the RTM and are linked to the code which interprets the RTM. Figure 3-1 shows the steps taken by a user performing a simulation. First, the architecture of the target machine is described in ISPS. The description is then compiled and translated into various tables containing the information from the ISP description. The tables, known as Global Data Base tables (GDB), are organized by the compiler in such a manner that the information they contain regarding the target machine is easily accessible to the simulator. The final step before the simulator is ready to run is to link the target machine tables to the simulator code.
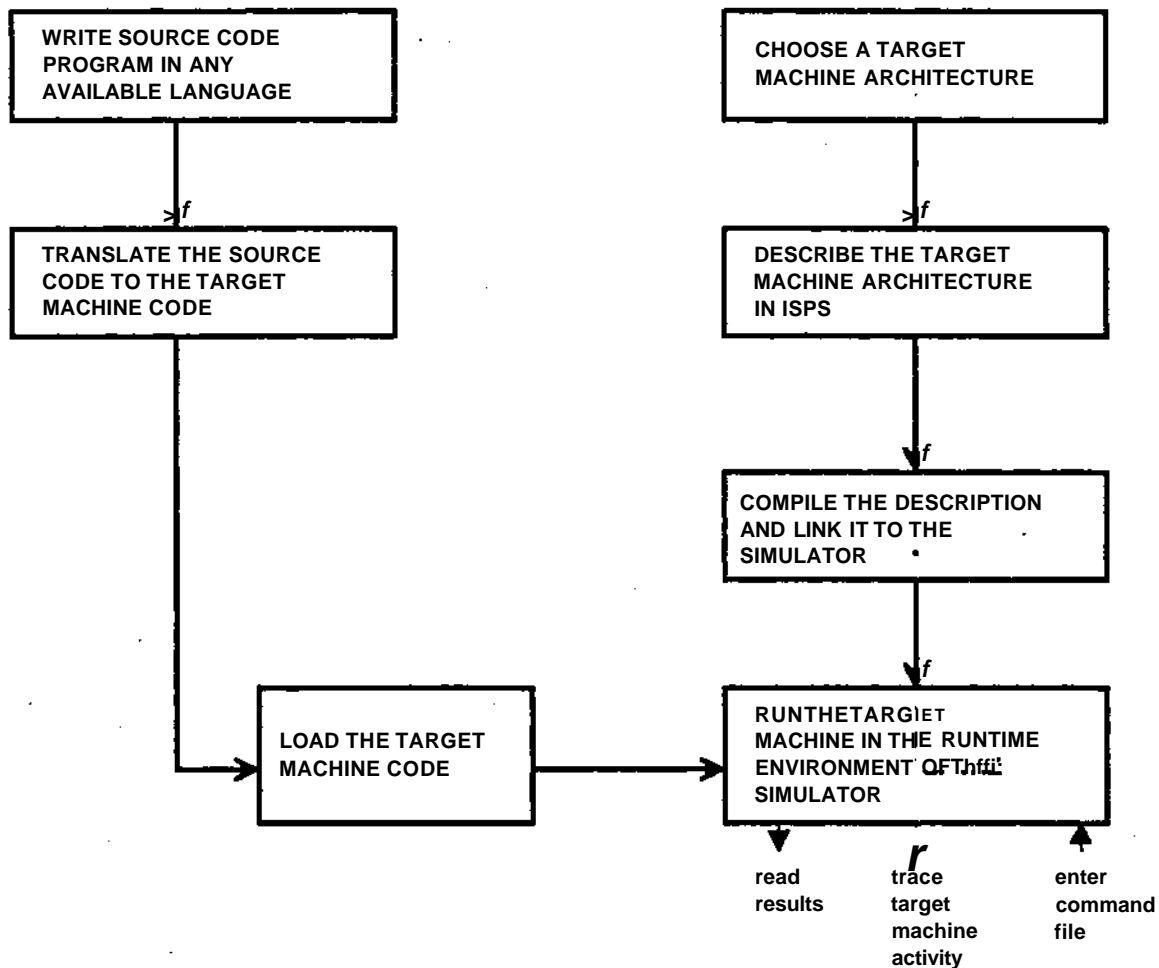
Figure 3-1: Sequence of Steps Necessary to Perform a Simulation

3.0.1 The GDB Tables

The information for simulating an arbitrary machine is contained in the GDB tables. These tables are generated during the ISPS compilation sequence previously described and contain target machine dependent RTM code. The GDB is composed primarily of two tables: the symbol table and the statement table.

The symbol table contains information regarding each entity declared in the ISPS description.

Examples of valid declaration types are register, memory, procedure, label, constant, etc. The bit count (length in bits) and word count (number of words) are stored in the symbol table for those entities for which this information is meaningful (i.e., register, memory, constant).

The statement table contains an entry for each RTM operation generated from the ISPS description. There is a field in each entry for the opcode of the RTM instruction as well as fields for pointing to entries in the symbol table which corresponds to the two source operands and the one destination operand.

3.1 The RTM

The RTM employs three addresses and is designed specifically for simulating ISP descriptions. The instruction set of the RTM can be divided into three general categories: data movement; arithmetic/logical; and control. Figure 3-2 shows the instructions that fit into each of these classifications.

3.1.1 Data Movement

This grouping includes all instructions which move data from one place to another. This includes such instructions as MOVE, READ and WRITE. Also included are instructions that transform the information contained in one or more data words to a new data word. Examples of instructions of this type are the following: CONCATENATE, COUNTONE, FIRSTONE, LASTONE and MKMASK. The CONCATENATE instruction connects the two source operands as shown in Figure 3-3. The bit length of the destination is the sum of the lengths of the two sources. The result of a COUNTONE operation is the number of bits in the source operand that is set to a 1. Similarly, FIRSTONE and LASTONE find the bit position within the source operand where either the first or last one is found. MKMASK generates a mask in the destination based on the values of the two source operands. A detailed description of these and other instructions is found in [Barbacci 81].

MOVE	LASTONE	PARITY
READ	MKMASK	RBYTE
WRITE	CLEAR	WBYTE
CONCATENATE	INCREMENT	MASKLEFT
COUNTONE	DECREMENT	MASKRIGHT
FIRSTONE	CONNECT	

DATA MANAGEMENT OPERATIONS

ADD	OR	EQL	SL0	SRD
SUBTRACT	AND	LEQ	SL1	SLI
MULTIPLY	XOR	GEQ	SLR	SRI
DIVIDE	NAND	NEQ	SLD	
NEGATE	NOR	GTR	SR0	
MOD	NOT	LSS	SR1	
	EQU	TEST	SRR	

ARITHMETIC/LOGICAL OPERATIONS

IF	JOIN	PEND	TIMEWAIT	ISRUNNING
BRANCH	LEAVE	BSELECT	SMERGE	
CALL	RESTART	DIVERGE	STOP	
START	SELECT	PMERGE	DELAY	
RESUME	TERMINATE	PJOIN	WAIT	
SJOIN	PBEGIN	UNDEFINED	UNPREDICTABLE	

CONTROL OPERATIONS

Figure 3-2: Classification of the RTM Instructions

SOURCE1

1	1	0	1

SOURCE2

0	1	1	0

DESTINATION

1	1	0	1	0	1	1	0
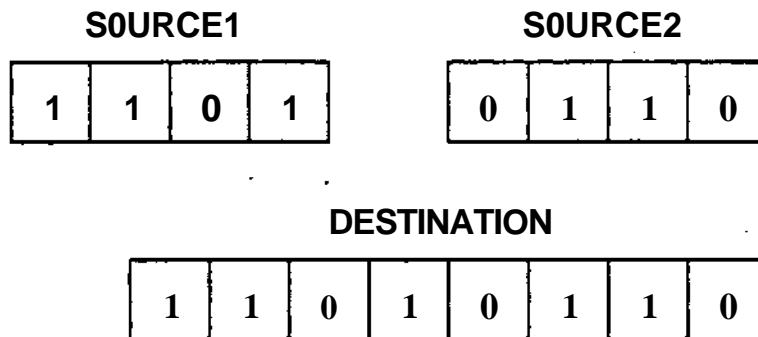
Figure 3-3: The Concatenate Instruction

3.1.2 Arithmetic/Logical

Instructions which fall into this category are those which control the ALU. Examples of such instructions are ADD, SUBTRACT, MULTIPLY and DIVIDE. In addition, logical operations such as OR, AND, XOR as well as instructions which compare two operands (EQU LEQ, etc.) would be included in this group. There is also a rich set of instructions for shifting and rotating data.

There is a one-to-one correspondence between the RTM Arithmetic/Logical instructions and the Arithmetic/Logical operations permitted in an ISPS description. All instructions in this group can be specified in any of four numbering systems: two's complement, one's complement, signed magnitude and unsigned.

3.1.3 Control Instructions

Instructions that affect the sequence of RTM execution belong in the control category. Examples of these are: IF, BRANCH, CALL, START, RESUME, etc.

3.2 Implementation Issues

There are several features of the RTM which should be noted because they are peculiar to this machine.

1. The capability to perform arithmetic and logical operations in four number systems (two's complement, one's complement, signed magnitude and unsigned).

2. Operands can be of any length. The same operation can have operands each of which is a different length.

3. Alignment/field extraction. The operands can be anywhere within the host memory word or even across word boundaries. See Appendix A for further explanation.

The RTM is currently implemented entirely in software. This implementation deals with the first issue of doing arithmetic in any of four number systems with separate high level language subroutines for each arithmetic instruction for each number system.

The software RTty has both a fast and a slow execution sequence for each instruction. If the semantics of Bliss coincide with the semantics of the RTM operation; then the fast sequence will execute. Otherwise, the simulator must do extra housekeeping associated with the operation. For example, if the operands are smaller than 36 bits (the word size of the PDP 10 on which the simulator is implemented), then the fast sequence will execute for some RTM operations. However, if the operands are larger than 36 bits, then the simulator must perform the operation on more data words while keeping track of things (such as the carry bit across word boundaries in an add operation), thus slowing down the execution speed for that instruction. Figure 3-4 shows the Bliss code for executing either the fast or slow sequence for the RTM CONCATENATE instruction. The slow sequence is contained in a separate subroutine.

The operands for the various RTM operations are packed in the PDP 10 memory so that they take up a minimum amount of memory space. For this reason, it is not unusual to find operands which lie across word boundaries. Also, ISPS allows the declaring of entities within other entities. Thus the RTM must be able to extract fields from within larger fields. This is a very common operation of the RTM since most instructions involve packed operands.

```
%conc% BEGIN
    IF fast
    THEN    mpdest[mpsize-1]«- (.mpsrd[mpsize-1] • 12) OR .mpsrc2[mpsize-1]
    ELSE    mpconc(mpdest<0,0>₁mpsrc1<0,0>,11₁mpsrc2<0,0>,12);
    tallvbop
    END;
```

**Figu re 3-4:   Bliss Code for Executing Either the Fast or Slow Sequence
for the RTM CONCATENATE Instruction**

4. Analysis of the Software Simulator

A detailed analysis of the software simulator is necessary to determine what performance gain (if any) can be acquired by implementation of the arithmetic/logical portion of the simulator's instruction set in hardware, assuming the rest of the simulator is eventually implemented in hardware.

There are several questions of particular interest in this analysis. Where does the simulator spend its time? What percentage of the time is it executing instructions from the Arithmetic/Logical group? How long does it take to execute the various RTM instructions? What is the penalty for doing RTM operations on long operands (greater than 36 bits)? And, finally, how often are long operands encountered?

4.1 Static and Dynamic Instruction Frequency Data

Instruction usage data was very useful in this analysis. By knowing how often the various RTM operations are actually used and how long it takes to execute each particular RTM operation, insight is gained into the potential advantage of a hardware ALU.

Data was collected for this analysis in two different ways:

1. a static instruction frequency count

2. a dynamic instruction frequency count

The static count is obtained by analyzing the statement table of the RTM files for the machine under study. The dynamic count is obtained by counting instructions as they are executed by the RTM. The motivation for performing both a static and a dynamic frequency count is that their results tend to complement each other. The static analysis by itself looks at all features of the target machine but fails to realize that most programs use certain features of a target machine more than others. On the other hand, the dynamic analysis looks at the features of the target machine that are used in the benchmark programs but overlooks those that are not. By analyzing the data collected with both methods, more can be learned about the potential speed up of the simulator with a hardware ALU.

Several shortcomings of this analysis should be noted at the outset. The results of either the static or the dynamic count experiments are highly influenced by the target machine, the ISPS description of the target machine, and the benchmark program for the target machine. The static analysis was performed on the ISPS descriptions in the CMU-A ISP library. Although the library contains descriptions for a cross section of machines, from bit-slice microprocessors (AMD2901) to large computers (IBM 360), it is far from exhaustive as to the types of machines that could be described in ISPS. Also, it is possible to describe the same machine with two or more different ISP descriptions. However, it is the best approximation until more complete data becomes available.

The same problem exists for the dynamic instruction frequency count experiments. The scope of data to be collected here is even greater than for the static case since the results of a single experiment are also dependent on the benchmark program that the experimenter chooses. For the dynamic instruction frequency count, some benchmark programs were taken from the library on the CMU-A and others were written by the experimenter.

The instruction usage data from the static analysis is summarized in Table 4-1. The data for this analysis was derived from the ISP descriptions of fourteen different machines, consisting of a total of approximately 2300 lines of ISPS code. Each machine was analyzed separately (Table 4-2). Table 4-3 summarizes the statistics from this analysis.

The dynamic analysis is described in detail in a CMU technical report by Benjamin Atlas [Atlas 82]. Table 4-4 summarizes the results presented in that report

4.1.1 Predicting the Percentage of ALU Instructions

4.1.1.1 Level of Abstraction of Modeling the Target Machine

Several factors influence the percentage of ALU operations contained in the RTM code generated from ISP descriptions. The first one is the level at which the description models the target machine (i.e., level of abstraction of the model) [Northcutt 82]. Examples of level of abstractions would be the gate level, the register transfer level, instruction set level, etc. [Bell and Newell 71].

MACHINE	TOTAL OPERATOR USAGE (N1)	TOTAL OPERAND USAGE (N2)	IMPLEMENTATION LENGTH	N1/N2	% ALU OPS
1) BDX900	841	1435	2276	1.496	23
2) AM2910dn	424	386	810	1.759	22
3) AM2901	382	469	851	1.744	20
4) HP21MX	867	1141	2008	1.988	20
5) AM2910lb	344	178	522	7.167	5
6) CDC 6600	2261	3749	6010	2.468	14
7) PDP8	255	233	488	2.55	13
8) PDP11/70	3316	3717	7033	3.531	10
9) INTEL 8080	888	741	1629	3.496	10
10) MARK1	65	40	105	3.824	9
11) MC6502	1020	783	1903	3.893	9
12) AYK14	4103	4165	8268	4.455	8
13) IBM/370	4412	5262	9674	4.070	8
14) ECLIPSE	4511	5015	9526	5.999	6

Table 4-1: Implementation Lengths of Various Models: Static Analysis

Machine	Total Operator Usage	Data Movement (%)	ALU/ Logical (%)	Control (%)
HP21MX	867	28.7	19.2	52.0
MC6502	1020	21.9	9.0	69.0
ECLIPSE	4511	35.7	5.9	58.4
IBM370	4412	35.5	8.3	56.2
INTEL 8080	888	20.0	10.1	69.8
BDX900	841	33.8	23.1	43.2
AM2910lb	344	13.7	4.9	81.4
AM2910dn	424	14.6	22.2	63.2
PDP8	255	20.4	13.7	65.9
PDP11/70	3316	36.4	10.0	53.6
AYK14	4103	31.1	7.8	61.1
CDC6600	2261	43.5	14.5	42.1
MARK1	65	10.8	9.2	80.0
AM2901	382	28.0	20.7	51.3

Table 4-2: Operator Usage--Static Analysis

database:

2300 lines of ISP code, comprising 14 ISP descriptions

Target Machine Descriptions	Models at the Microcode Level	Models at the ISP Level	Percentage of ALII Operations
1 B0X900	X.		23
2 AM2901dn	X		22
3 AM2901	X		20
4 HP21MX	X		20
5 AM29101b	X		5
6 CDC 6600		X	14
7 PDP8		X	13
8 PDP11/70		X	10
9 INTEL 8080		X	10
10 MARK1		X	9
11 MC6502		X	9
12 AYK14		X	8
13 IBM/370		X	8
14 ECLIPSE		X	6

Summary of statistical analysis for
the above machines

	All Models	Models at the Microcode Level	Models at the ISP Level
mean	12.64	21.25	9.67
standard deviation	6.15	1.5	2.5
variance	35.09	1.6	5

Table 4-3: Statistics of the Static Analysis

The descriptions studied for this analysis modeled target machines at one of two levels: **the** register transfer level (microcode) and the instruction set level (ISP). Table 4-3 separates the data into groups according to the level of abstraction of the model and shows the statistics for each group.

TARGET MACHINE	PROGRAM DESCRIPTION	DATA MOVEMENT (percent)	ALU/ LOGICAL (percent)	CONTROL (percent)
PDP11/70	DEC diagnostic for branch instructions	25	8.2	65.5
PDP11/70	DEC diagnostic for unary instructions	26	9.4	63
PDP11/70	DEC diagnostic for floating point instructions	30	10	58
PDP11/70	DEC diagnostic for floating point instructions	30	10	59
PDP11/70	DEC diagnostic for floating point instructions	29	9.6	60
PDP11/70	DEC diagnostic for arithmetic instructions	26	9.9	63
INTEL 8080	matrix multiplication	21.4	15.1	63.4

Table 4-4: Summary of the Dynamic Instruction Usage Analysis

An inspection of the various descriptions reveals that machines modeled at the ISP level do not model the interpretation of the target machine instructions in line with the decoding of the instructions, but rather call subroutines to do the interpretation. The more levels of indirection in the interpretation cycle, the more overhead is incurred in terms of transferring control from one subroutine to another. An examination of Table 4-2 will verify this argument. This table shows that the machines modeled at the ISP level (see Table 4-3) also have a larger percentage of control instructions, indicating a larger use of subroutines in the descriptions. When there is a larger percentage of control instructions, the percentage of ALU instructions naturally drops. An examination of the descriptions modeled at the ISP level of abstraction shows fewer levels of indirection in the instruction interpretation cycle for those machines with the higher percentage of ALU instructions.

4.1.1.2 Measuring ISP Complexity

The various ISP descriptions were classified according to their complexities. The goal of this portion of the analysis was to relate the percentage of ALU operations in the RTM code to the complexity of the ISP description of the target machine. One measure of complexity (which is the one chosen for analysis) is the implementation length (N) of the description. The basis for the validity of this measure is described in *Elements of Software Science* by Maurice Halstead [Halstead 77]. Halstead defines implementation length as:

$$N = H_x + N_2$$
where:

N = implementation length

H_x = total usage of all operators appearing in that implementation

N_2 = total usage of all operands appearing in that implementation

Table 4-1 shows the implementation lengths for the various ISP descriptions studied in this analysis. Figure 4-1 describes the relationship between implementation length and the percentage of ALU operations in the RTM code for the target machine.

Each point on the plot in Rgure 4-1 represents one of the machines analyzed in the static instruction frequency usage experiments of Table 4-1. In Table 4-1, the machines are numbered 1 through 14. These numbers are used to encode the various machines in the plot of Figure 4-1. A circle around a point indicates data derived from the dynamic analysis. When more than one dynamic instruction usage observation was available, the point furthest from the statically derived data was plotted. An arrow indicates the difference between the two measurements, thereby giving an indication as to the range in percentage of ALU operations encountered for a particular model.

Machine	% ALU Ops	Length
MICROCODE LEVEL DESCRIPTIONS		
1 BDX900	23	2276
2 AM2910dn	22	810
3 AM2901	20	851
4 HP21MX	20	2008
5 AM2910lb	5	522

Machine	% ALU Ops	Length
ISP LEVEL DESCRIPTIONS		
6 CDC 6600	14	6010
7 PDP8	13	488
8 POP11/70	10	7033
9 INTEL 8080	10	1629
10 MARK1	9	105
11 MC6502	9	1903
12 AYK14	8	8268
13 IBM/370	8	9674
14 ECLIPSE	6	9526

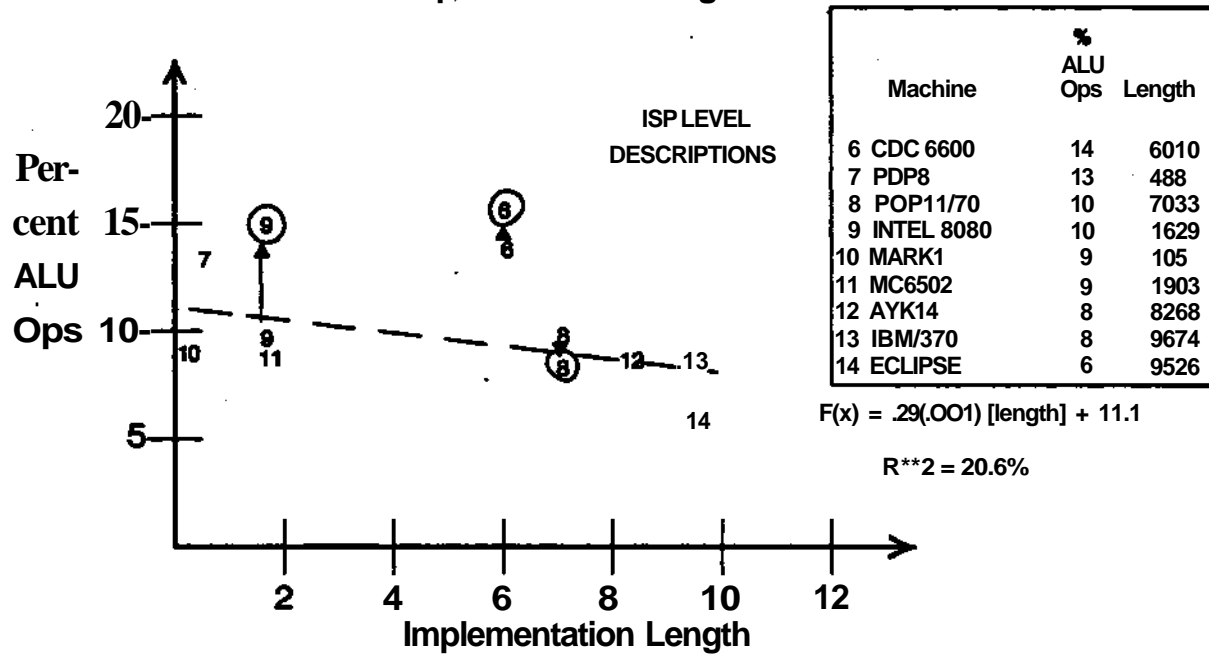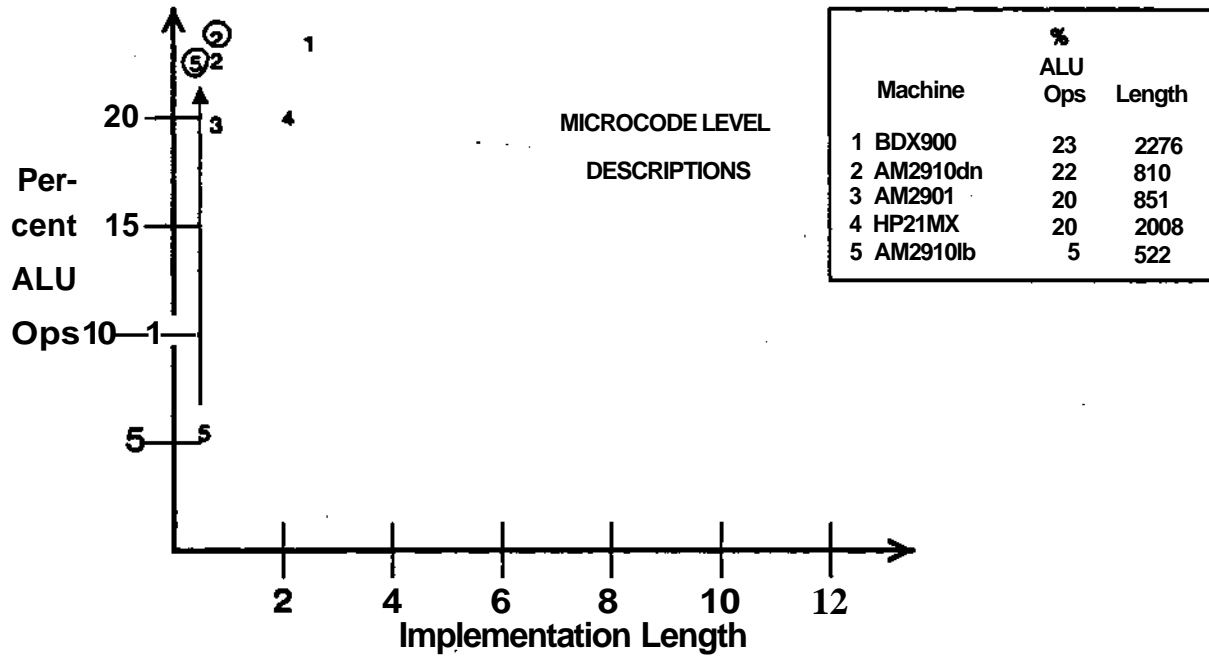$F(x) = .29(.001) [length] + 11.1$

$R**2 = 20.6\%$

Figure 4-1: Relationship between Implementation Length of the Target Machine Description and the Percentage of Generated ALU RTM Operations

4.1.1.3 Programming Style

The two different descriptions for the AM2910 (points 2 and 5 in Figure 4-1) demonstrate the affects description style have on the static count of the percentage of ALU operations. The higher percentage of ALU operations for point 2 is due to the description stating ALU operations common to all selections of a decode explicitly rather than stating them before decode as in description 5 [Northcutt 82]. This is verified in table 4-2 by noting that the additional operations for the AM2910dn over the AM2910lb are arithmetic/logical operations and associated overhead. It should be noted that the difference in percentage of ALU operations generated by these two ISP descriptions lies at the extremes for the data collected.

The dynamic instruction usage analysis reveals a different result. Both descriptions for the AM2910 executed approximately the same percentage of arithmetic/logical operations. This can be explained by the above argument The code that was factored out and stated before the decode in description AM2901lb is executed before the decoding of each instruction. The AM2910dn executes similar code during the interpretation of each machine instruction. Appendix B contains portions of the two ISP descriptions and points out differences in styles.

4.1.2 Timing

How much time does it take to execute the various RTM instructions with the current software implementation? This information, combined with the instruction usage analysis, is necessary to evaluate the potential speedup for a particular ISP description.

Execution speed data for several instructions from each category of the RTM instruction set were taken. Multiplying the relative execution speed of the ALU operations (with respect to the other instruction categories) by the relative frequency usage of the ALU operations gives the expected speedup.

The methodology and collection of the data for the timing analysis is described in a CMU technical report by Benjamin Atlas [Atlas 82]. Table 4-5 shows the execution speed for various RTM

operations. The table shows a large difference in execution speed for performing certain ALU operations over others. For example, two's complement and sign magnitude addition take approximately six times longer to execute than do other operations in the table. Table 4-6 shows the frequency of use of those ALU instructions having the higher execution speeds.

OPCODE	MNEMONIC	SLOW SPEED	FAST SPEED	DIFFERENCE
44	SRO	60 to 120 depending on number of shifts		
60	NOT	40	10	30
61	AND	40	10	30
62	OR	50	20	30
65	XOR	40	10	30
101	2CAD0	230	200	
111	EQL2C		20	
141	SMADD	220 to 320 depending on signs		
161	USADD	40	10	30
162	USSUB		20	
171	USEQL	40	10	30
172	USNEQ	40	10	30
176	USGTR	.40	10	30

Table 4-5: Execution Speed of RTM Operations-all times in microseconds

A dynamic timing analysis shows a strong relationship between the percentage of instructions executed from each category of RTM operations and the percentage of time spent by the simulator executing the instructions from that catagory. Table 4-7 demonstrates this assertion and is typical of the data derived from the dynamic timing analysts. The test program for this case was a crossection of twenty instructions from the Intel 8080 instruction set.

Machine	Frequency of Instructions With High Execution Times (percent)
HP21MX	2.5
MC6502	4.3
ECLIPSE	.19
IBM/370	.1
INTEL 8080	4.2
BDX900	13.6
AM29101b	0
AM2901dn	.26
PDP8	5.1
PDP11/70	.63
AYK14	1.6
CDC6600	0
MARK1	5.1
AM2901	0

Table 4-6: Frequency of Use for ALU Operations with High Execution Speed

	PERCENTAGE OF TOTAL INSTRUCTIONS		PERCENTAGE OF TOTAL EXECUTION TIME	
	ARITHMETIC	LOGICAL	ARITHMETIC	LOGICAL
DYNAMIC MEASUREMENT (8080 TEST PROGRAM)	6	9	6	4
STATIC MEASUREMENT (8080 RTM TABLES)	6	4		

Table 4-7: Relationship between the static and dynamic instruction frequency usage and the percentage of total execution time spent by the simulator performing RTM ALU operations

Table 4-5 gives the execution speed for RTM instructions operating on both long and short

operands whenever data was available. A long operand is one that is larger than 36 bits (the word size of the PDP 10 on which the simulator analysis was performed), and a short operand is one that is less than 36 bits. The table indicates an execution speed penalty of approximately 30 microseconds for doing arithmetic/logical RTM operations on long words with the exception of shift operations. The penalty for these operations varies according to the number of shifts. Data was collected to determine the relative frequency of performing ALU operations on long operands versus short operands. Both static and dynamic operand bit length frequency counts were taken. The data for the static count was collected as follows. The symbol table of the GDB for each description was scanned. The bit length for each entry was entered into an array. Then the statement table was scanned. Each time an operand was pointed to as a source or destination for an ALU operation, a counter associated with the operand was incremented. The counters for all operands having the same bit length were added. Figure 4-2 summarizes the results of this analysis with a histogram of the operand word length. The histogram, which is a composite of all descriptions studied, shows that most operands are less than 32-36 bits long. An examination of the tables together with an analysis of the dynamic operand length provides additional insight as to the effects of operand size on ALU performance.

With the exception of the CDC 6600, descriptions studied were for machines with word lengths of 36 bits or less. Although the histogram of Figure 4-2 shows a significant number of long operands, in many cases, the ALU RTM instructions are actually working on subfields of these operands and these subfields are smaller than 36 bits. This was demonstrated in the dynamic frequency analysis using the description for the PDP11/70 on the CMU-A ISP library as an example. A benchmark program was written [Atlas 82] with floating point instructions which referenced long operands. However, for the 3,746 ALU RTM operations that were executed, only 44 of them or 1.2 percent operated on long operands.

A benchmark program was also written for the CDC6600 (60 bit length machine) containing only ALU operations [Atlas 82]. Counting the number of RTM ALU operations on long operands while executing this program on the simulator should give some indication as to the worst case frequency of encountering them.

NUMBER OF
TIMES USED

HISTOGRAM OF OPERAND WORD LENGTHS

(STATIC MEASUREMENTS)

3500

3000

2500

2000

1500

1000

500

1-4   5-8   9-12   13-16   17-20   21-24   25-28   29-32   33-36   37-48   49-60   61-80   81-144

OPERAND BIT LENGTH
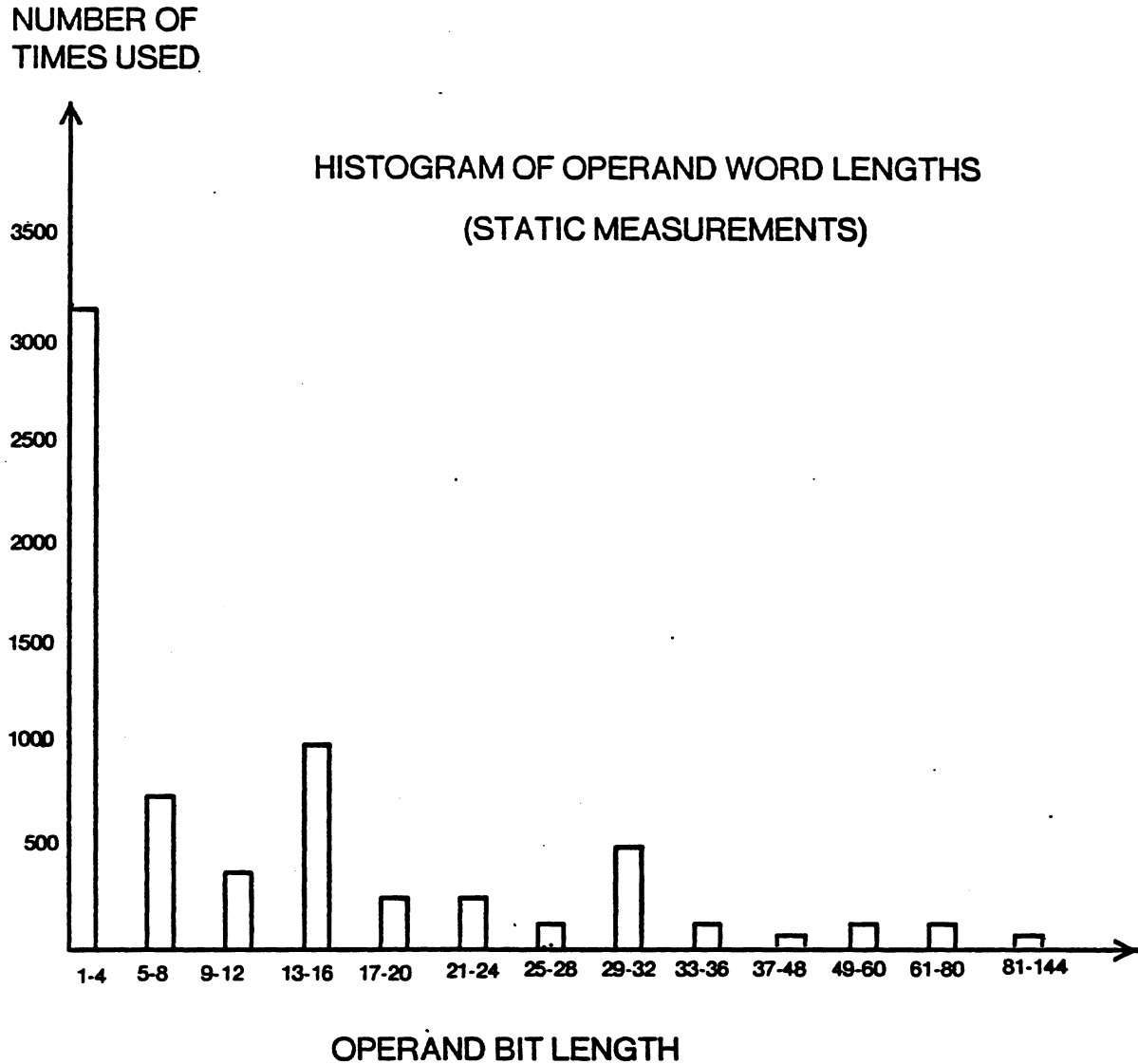
Figure 4-2: Histogram of Long vs. Short Operands--Static Measurement

The results show a relatively small percentage of long operands (1-2 percent). This is explained by the fact that there is an overhead associated with the execution of each target machine instruction in terms of additional RTM operations. These extra operations are necessary for houskeeping functions of the RTM. Because additional operations are on short operands, they minimize the relative frequencies of encountering long versus short operands.

This analysis shows that a hardware ALU word size of 32-36 bits would be sufficient since the worst case frequency of encountering larger operands is small. Furthermore, this analysis indicates that the ALU operations account for about 5-20 percent of the total execution time-the maximum potential speedup of a hardware ALL).

5. Hardware Implementations for the ALU

The hardware implementation for the simulator can be divided into two modules: the host and the ALU. Figure 5-1 shows how the two modules fit together.

5.1 The Host

The host computer sequences through the RTM code as it interprets the statement table of the GDB. The host also takes care of all overhead associated with the simulation of the target machine. This includes operations such as the operator interface, setting breakpoints and monitoring the execution of RTM code.

5.2 The ALU

The ALU executes the RTM instructions. It performs arithmetic, logical, shifting and other operations on data from the host The results of the operations are sent back to the host

As shown in Figure 5-1, the ALU can be divided into two modules: the data path portion and the control portion. The control portion receives instructions from the host and coordinates the activity of the data path portion. All communication of status and control information between the host and the ALU is via the control portion, and all data is transferred between the host and ALU via the data path portion.

5.2.1 Specifications for the ALU

5.2.1.1 Inputs

Inputs to the ALU are instructions and data. The instructions contain the opcode of the RTM operation to be executed. The length (number of bits) as well as the bit offset within the host memory word of each operand is included as part of the instruction.
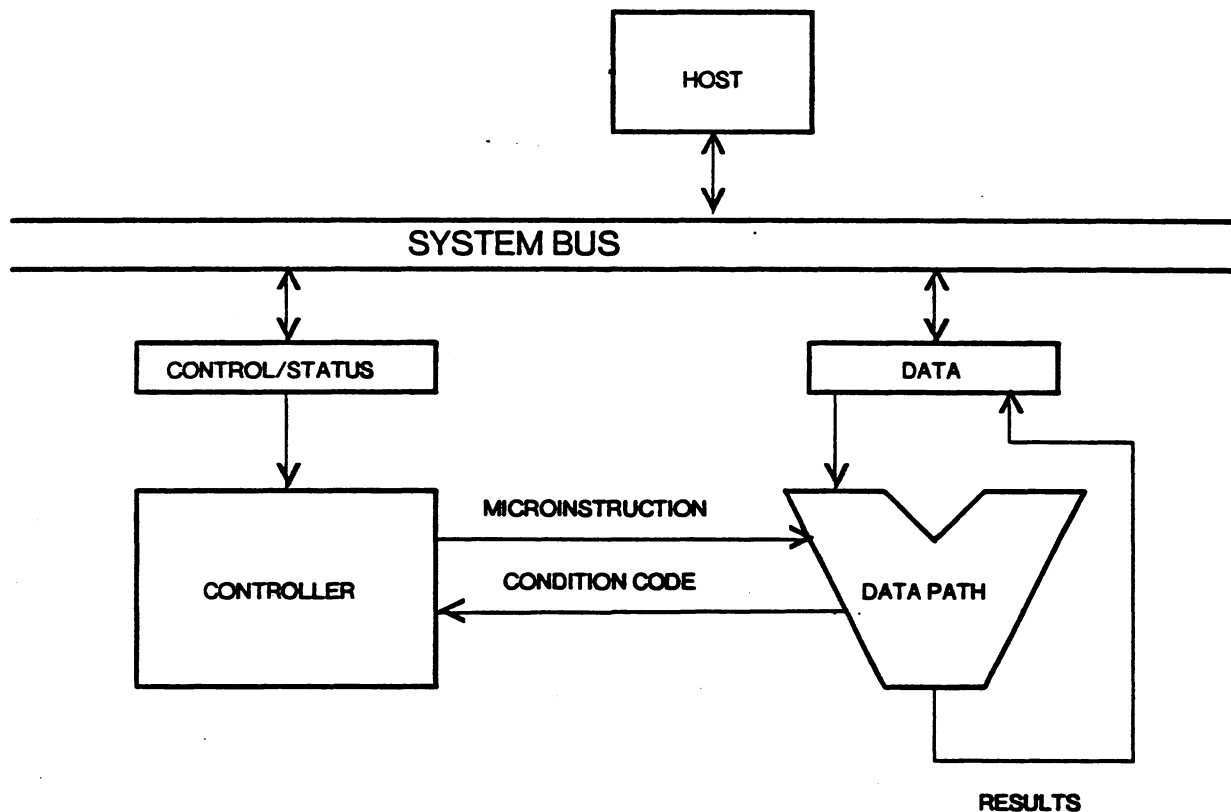
Figure 5-1: The ALU

5.2.1.2 Outputs

There are two outputs from the ALU which are returned to the host computer: the results of the RTM operation and processor status information.

5.2.1.3 Functions

The ALU will emulate RTM instructions from the Arithmetic/Logical instruction category discussed in Section 3.1.2. The instruction set includes operations from several number systems: one's complement, two's complement, signed magnitude and unsigned. The operands can be any specified length.

5.2.2 Variable V/ord Length

One of two problems can occur when manipulating data with an arbitrary word length on a fixed word length ALU. The first case is when the data word is smaller than the word size of the ALU. This situation can be dealt with by right justification and sign extension of the data. The other situation is when the operand is larger than the ALU word size. In this case, the data operation must be done in steps with intermediate parameters peculiar to the operation (i.e., carry, borrow, etc.) saved between steps. An ALU slice is the width of the data path of the ALU. The control portion of the ALU can keep track of how many ALU slices to fetch for each operand, as well as other housekeeping data.

5.2.3 Alignment/Field Extraction

The host will pass the ALU the necessary information to extract fields from within larger data words and perform alignment of operands. The information required is as follows:

- the bit offset of the field within the larger word

- the length of the field

An algorithm for doing alignment is shown in Figure 5-2. The algorithm will right justify the operand and zero out the unused high order bits.

5.2.4 Arithmetic Operations in Four Number Systems

Arithmetic is done with full adders with the proper correction terms added depending on the particular number system of the RTM instruction. Figure 5-3 show examples of algorithms for performing addition and subtraction with full adders in two's complement, one's complement, signed magnitude and unsigned number systems.

5.2.5 Implementations

Two hardware implementations for the ALU were analyzed. The first is based on the Advanced Micro Devices AMD2900 family of bit slice microprocessors, and the second is based on a microprocessor chip designed at Carnegie Mellon University.

GIVEN:

the length of the operand in bits (Length)
the bit offset of the operand with the word (Bitoff)
the ALU slice width in bits (ALUSLICE)
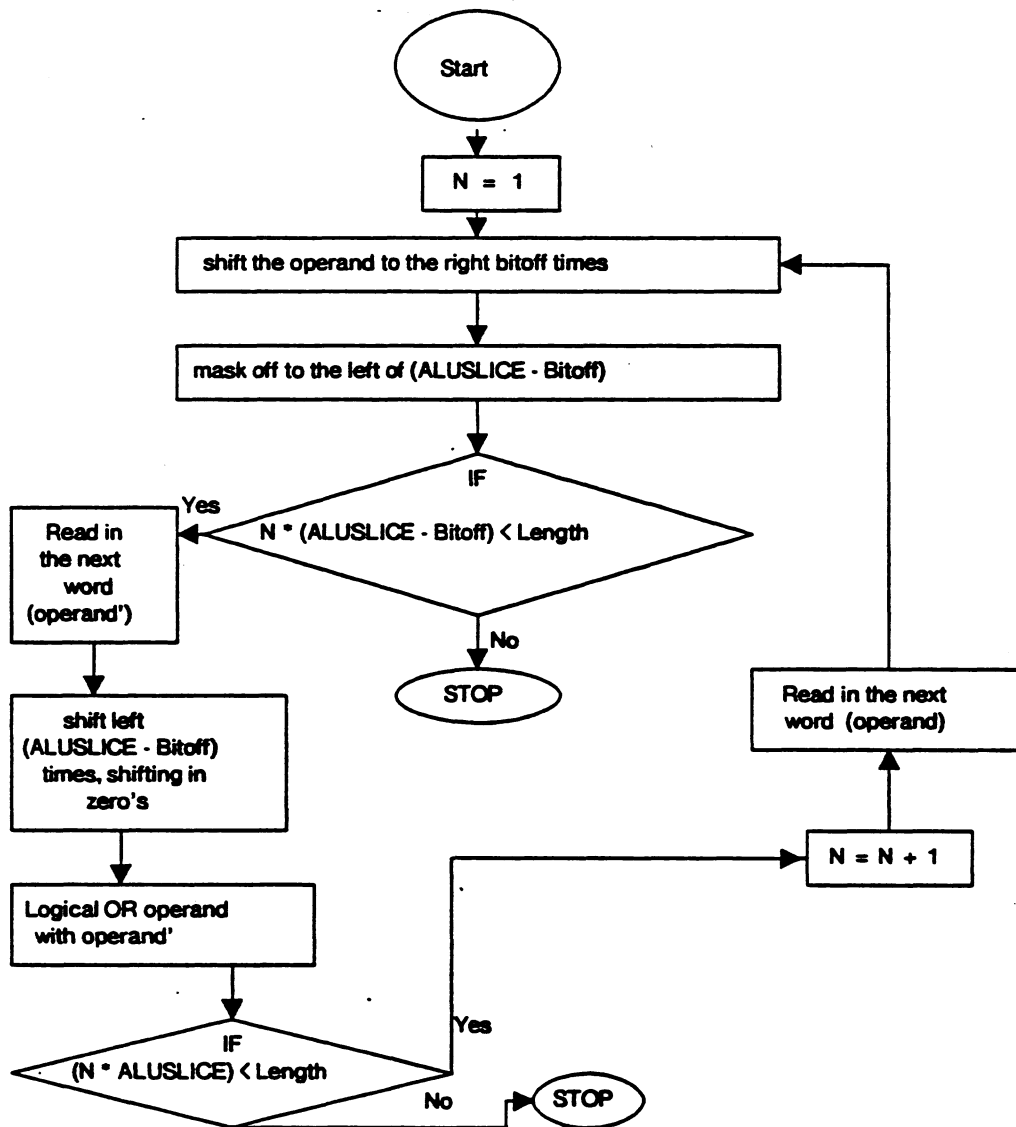
Figure 5-2: An Algorithm for Doing Alignment and Field Extraction

SIGN and MAGNITUDE

Subtraction:
—change the sign of the b operand and proceed as for addition

Addition:
—exclusive OR the signs of the two operands

IF FALSE (0)
—then ADD the two operands and give the sign of either

ELSE
— INVERT the b operand and add the two operands (except
the sign bit)
--check for carry out of MSB

IF TRUE
—then ADD 1 to the LSB of the sum
—the sign is that of the a operand

ELSE
—then ADD 1 to the LSB of the sum
—the sign is that of the b operand

TWO'S COMPLEMENT

Addition:
—the a operand is added to the b operand (including sign bit)

Subtraction:
—the b operand is inverted and added to the a operand
(including sign bit)
—a 1 is added to the least significant bit of the sum

ONE'S COMPLEMENT

Addition:
—the a operand is added to the b operand (including sign bit)
—a 1 is added to the LSB of the sum if a carry out of the sign
bit occurs

Subtraction:
—the b operand is inverted and added to the a operand
—a 1 is added to the LSB of the sum if a carry out of the sign
bit occurs

Figure 5-3: Algorithms for Addition and Subtraction in Four Number Systems

5.2.5.1 AMD2900 Bit Slice

Figure 5-4 shows a block diagram of the four bitwide data path of the AM2901 [AMD 81]. Several 2901's can be connected together in series to provide an ALU slice width of any multiple of four bits. *The AMD Microprocessor Logic and Interface Data Book [AMD* 81] and *Bit Slice Microprocessor Design* [Mick and Brick 80] describe the 2900 family of chips in detail and give numerous examples of how to use them.

Figure 5-5 gives the microcode for the RTM OR instruction and the number of machine cycles it would take to execute for the block diagram of Figure 5-6.

5.2.5.2 VLSI

A second implementation is based on a microprocessor chip designed in-house specifically for the RTM instruction set. Because the chip is non-standard and very little documentation exists, it will be described in detail.

The name of the chip is P.mbs which stands for Processor.multiple byte slice. The layout for P.mbs was performed by two Carnegie Mellon University Electrical Engineering students, William Birmingham and Jamie Saulnier. Production and testing of the chip is planned. P.mbs is a microprogrammed arithmetic logic unit and is designed to be the ALU data path portion of an ISP RTM. Figure 5-7 shows the data paths and functional blocks of P.mbs. It is a bit slice machine where each slice is 8 bits wide. Slices can be cascaded to form an ALU of any size multiple of 8 bits (1 byte).

Timing

The chip is docked with two phases per machine cycle. During the first phase, data is moved onto the buses and latched at their various destinations where they will be worked on during phase two. The chip can be clocked at a rate of 1.96 MHZ [Birmingham 82]. The clocking rate is limited in clock phase two by the ALU module.

The Buses

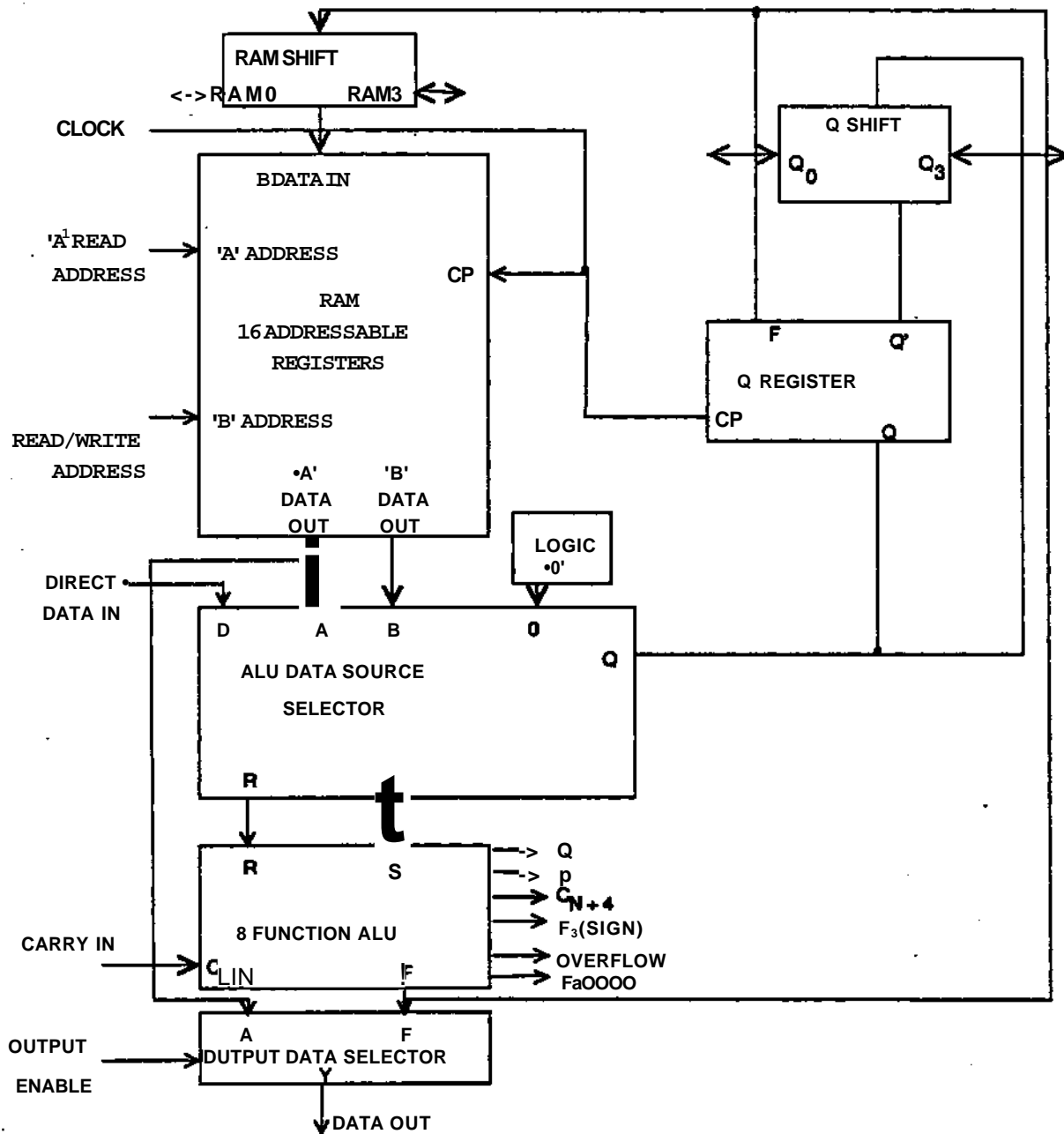**Figure 5-4: The AM2901 4-bit Microprocessor Slice [AMD 81]**

P.mbs has two buses through which data can be transferred among the various elements of the chip. Sources for the A bus are the I/O latch, output latch of the shifter/masker, the RAM A port and the output latch of the ALU. The B bus can select from all of the same sources as the A bus with the exception of the output latch of the ALU.

MICROCYCLE	MICROINSTRUCTION	COMMENT
1	•shift operand A \<bitoff> times \<bitoffa>=4 for this example (this operation takes 1 micro-cycle for each shift)	align operand a
5	•load mask for field extraction of operand into RAM	
6	•load mask for field extraction of operand b into RAM	
7	•shift operand b \<bitoffb> times \<bitoffb>=4 for this example (this operation takes 1 micro-cycle for each shift	align operand b
11	•mask off 4 bit field of operand a by ANDing mask with operand a	perform field extraction for operand a
12	•mask off 4 bit field of operand b by ANDing mask with operand b	perform field extraction for operand b
13	•OR operand a and operand b	

Figure 5-5: Microinstruction Sequence for Implementing the RTM OR Instruction on the AM2901 for Operands Less Than ALUSLICE Bits Wide

The I/O Latch

The I/O latch allows the data pins for the chip to be bidirectional. The two bit opcode for controlling the latch allows the programmer to select the A bus or the data pins for the latch or to put the contents of the latch onto the chip's data pins.

The Shifter/Masker

Data is latched into the shifter from either the A bus or B bus during clock phase one. The shifter is an eight bit barrel shifter capable of shifting up to eight bits across chip boundaries. Data is shifted from the most significant bit (MSB) to the least significant bit (LSB). Data for the shifter from outside the chip is from the higher order slice via the eight bit shifter port. Before performing a shift in which data is being shifted across chip boundaries, the data in the shifter must be put onto the chip's
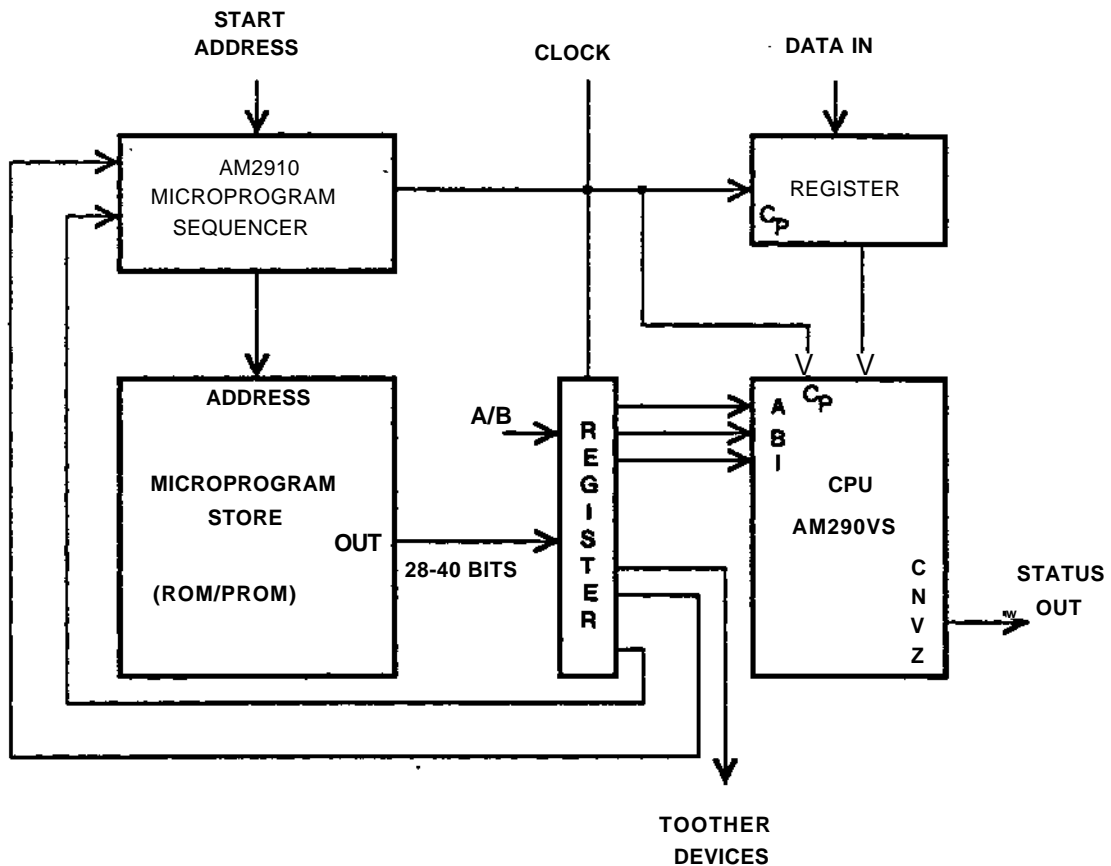
Figure 5-6: Implementation of an ALU Built around the AM2901 [AMD 81]

data I/O pins so that they are available to the lower order chip's shifter port. The masker performs a

logical AND with the shifter output and a specified mask.

ALU

The ALU performs one of five arithmetic/logical functions as specified by an opcode. The five

functions are as follows: AND, OR, XOR, ADD WITH CARRY and INVERT. Data come from the A bus

and/or the B bus. Four flags indicate the status of the previous result and can be useful for

implementing a mechanism for conditional branching. These are N (negative), C (carry), V (overflow)

and Z (zero). Results can be placed on the A bus.

Carry generate and carry propagate signals are produced so that full carry lookahead can be

implemented across chip boundaries.

OFF CHIP SIGNALS
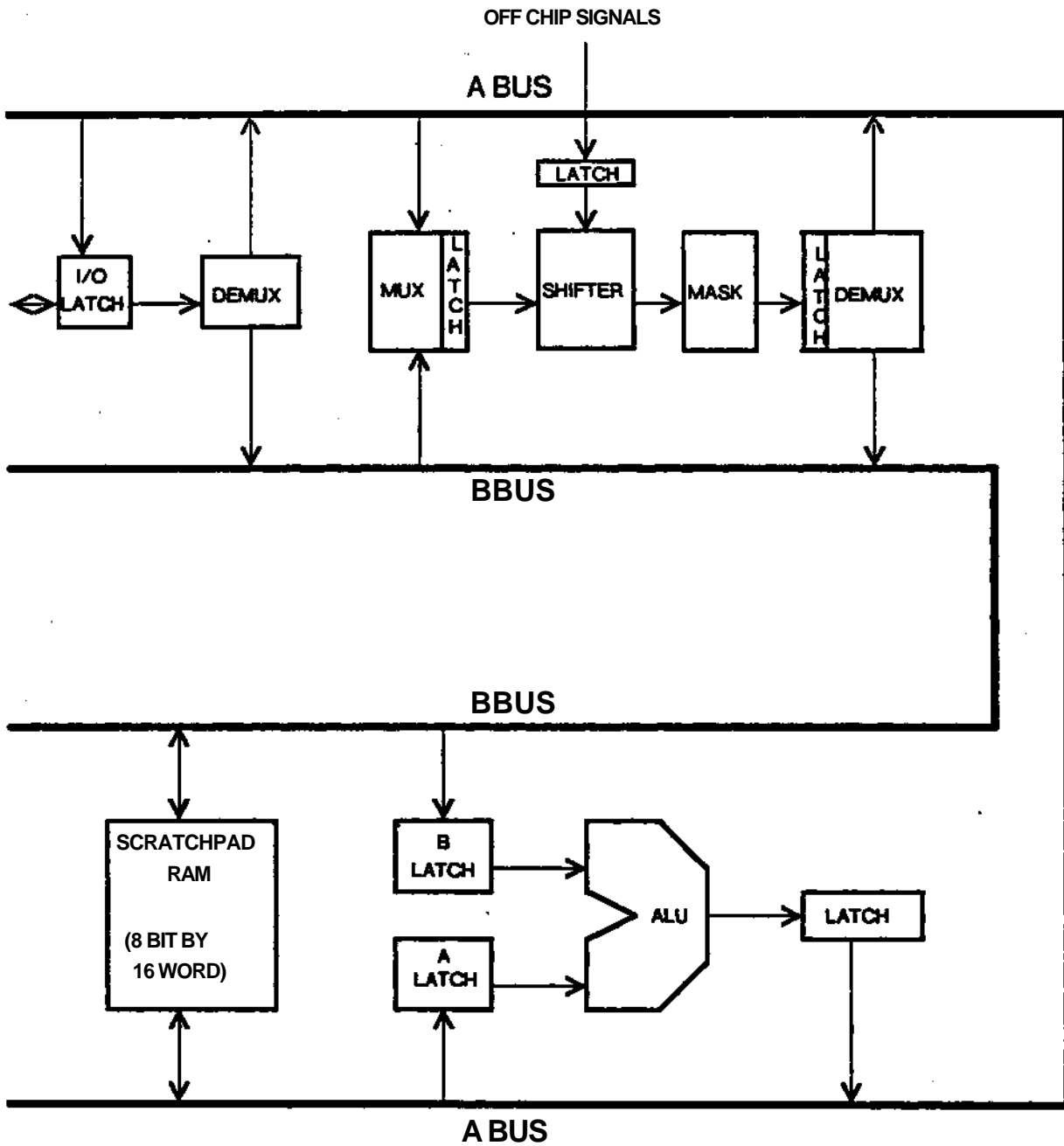
A BUS

BBUS

BBUS

A BUS

Figure 5-7:   P.mbs Functional Block Diagram

RAM

A sixteen word by eight bit RAM is provided to buffer intermediate results. RAM is two ported.

Any cell can be accessed for reading or writing at the same time: one by the A bus and the other by

the B bus.

5.2.5.3 Programming P.mbs for the RTM Instruction Set

To gain some idea as to the performance of P.mbs with regard to the RTM instruction set, several RTM instructions were microprogrammed. The sequence of events is similar for most of the instructions.

- the source 1 operand is latched from the I/O pins to the I/O latch

- the A bus is loaded from the I/O latch; also the source 2 operand is now loaded into the I/O latch

- the shifter is loaded from the A bus, the data is shifted and the source 1 operand is masked; also the B bus is loaded with the source 2 operand from the I/O latch

- the shifter is loaded from the B bus, the data is shifted and the source 2 operand is masked; also the ALU is loaded with the A operand

- the ALU is loaded with the B operand

- the arithmetic/logical operation is performed

- the results are written to the I/O pins through the I/O latch

Figure 5-8 shows the microinstructions necessary to add two operands in two's complement when each of the operands is less than 32 bits.

```
MACHINE
 CYCLE         INSTRUCTION                       COMMENT


1)       IO.LAT          <--IO.PINS  READ OPERAND A FROM THE PINS
                                     INTO THE CHIP


2)       A. BUS          <-IO.LAT    LOAD THE A-BUS WITH OPERAND A

         IO.LAT          <--10.PINS  READ OPERAND B INTO THE CHIP

         SHIFT.MUX       <-A.BUS     LOAD THE SHIFTER WITH OPERAND A
                                     FOR ALIGNMENT AND FIELD EXTRACTION
         SHIFT<BITOFF(A)>PLACES
         TO THE RIGHT                BITOFF IS THE BIT OFFSET OF THE
                                     WITHIN THE ALU SLICE

         MASK OUT THE <BITOFF(A)>
         MSB'S


3)       B.BUS           <-IO.LAT    LOAD THE B-BUS WITH OPERAND B

         A.BUS           <--MASK.OUT LOAD OPERAND A FROM THE MASKER
                                     TO THE A-BUS

         SHIFT.MUX       <--B.BUS    LOAD THE SHIFTER WITH OPERAND B
                                     FOR ALIGNMENT AND FIELD EXTRACTION

         SHIFT<BITOFF(B)>PLACES
         TO THE RIGHT                BITOFF IS THE BIT OFFSET OF THE
                                     FIELD WITHIN THE ALU SLICE

         MASK OUT THE <BITOFF(B)>
         MSB'S

         ALUA            <--A.BUS    LOAD THE ALU WITH OPERAND A


4)       B.BUS           <--MASK.OUT PUT OPERAND B ONTO THE B-BUS

         ALUB            <--B.BUS    AND LOAD INTO THE ALU

         ADD, OR, AND, XOR           PERFORM THE DESIRED OPERATION


5)       A.BUS           <-ALU.LAT   PUT RESULT ON THE A-BUS

         IO.LAT          <-A.BUS     AND INTO THE I/O LATCH


6)       10.PINS         <-IO.LAT    PUT THE RESULTS OF THE OPERATION
                                     ONTO THE PINS
```

Figure 5-8: P.mbs: Microprogramming for Operands Less Than the ALU Slice Width

6. Conclusions

6.1 Summary

This project presented an analysis of a software simulator in order to determine the advantages of a hardware ALU. The results indicate that only a relatively small portion (5 to 20 percent) of the RTM operations would be handled by a hardware ALU. Therefore, the advantage of converting only the ALU portion of the simulator from software to hardware is minimal in terms of expected overall performance improvement. However, when the rest of the ISP simulator is implemented in hardware, the ALU portion should be converted to hardware because the percentage of time spent by the RTM on arithmetic and logical operations would be considerably greater than 5 to 20 percent

Static and dynamic instruction usage data were evaluated in the analysis of the simulator. Data collected by these methods showed a similar mix of RTM instructions and both were useful.

The AMD 2900 implementation does poorly when it comes to aligning operands and performing field extractions. This is due to the fact that it can only shift one bit position per one microcycle. Generating the proper mask for field extraction is also a nontrivial task eating up additional microcycles. The VLSI implementation deals with these problems by providing an on-chip 8-bit barrel shifter capable of shifting across chip boundaries and an on-chip masker which provides up to sixteen masks. The layout for the VLSI implementation has been completed, and there are plans for production and testing of the chip. The architecture for the chip has been verified by simulation with ISP.

ALU operations account for approximately 5 to 20 percent of the total time spent by the simulator. The VLSI chip could perform these operations 60 to 70 percent faster than the software ALU implementation. However, two quantities have not been taken into account in this measurement which may improve or detract from performance.

On the one hand, the VLSI chip performs alignment and field extraction of operands. In the software implementation, this function was performed by simulator code not included in the RTM. Since alignment and field extraction are performed very often, including this function in the hardware ALU will greatly improve total system performance.

But on the other hand, it takes additional time to load the system bus and feed the hardware ALU data and instructions necessary for the RTM operation. This includes handshaking or other methods for coordinating the data transfer. The transferring of data and instructions from the host to the ALU would detract from the performance improvement of a hardware ALU.

6.2 Future Work

A number of questions relating to a hardware implementation of the RTM ALU has not been touched on and thus require further investigation. First, the interface between the host and the ALU should be defined. The interface consists of both hardware and software. The simulator should be modified to present the ALU the opcode of the RTM instruction, the operands and the information necessary to align the operands (i.e., the bit offset of the operand within the host memory word and the bit count). The hardware interface consists of registers on the system bus (see 5-1). The ALU and host could signal each other READY or BUSY by using bits in these registers.

A second area for investigation is an evaluation of the potential performance improvement if a front-end barrel shifter and masker is added to do alignment and field extraction. This operation was the major bottleneck in both the AMD2900 and the VLSI implementation.

Appendix A. Alignment/Field Extraction of Operands Across Word Boundaries

ISPS allows the declaration of operands across word boundaries of the target machine. These operands can be anywhere within the host memory word or even across host memory word boundaries as shown by the example below.

```
MB[0:maxb]<0:7>                              1 Byte memory
MH[0:maxb]<0:15>{INCREMENT:2}  :«▪          t Half word memory
        MB[0:maxb]<0:7>,
MW[0:maxb]₉0:31>{INCREMENT:4}  :<▪          I Word memory
        MB[0:maxb]<0:7>ₜ
MDW[0:maxb]<0:63>{INCREMENT:8} :<▪          IDoubleword memory
        MB[0:raaxb]<0:7>,
```

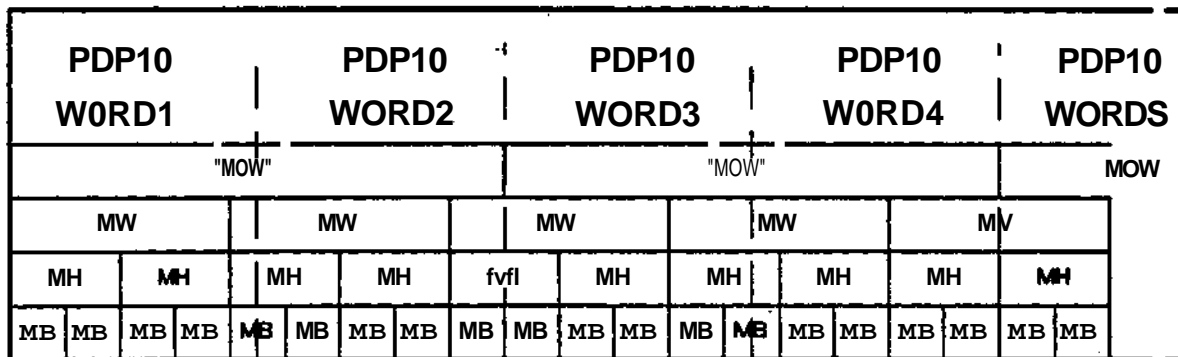| PDP10 W0RD1		PDP10 WORD2		PDP10 WORD3		PDP10 W0RD4		PDP10 WORDS
"MOW"				"MOW"				MOW
MW		MW		MW		MW		MW
MH	MH	MH	MH	fvfl	MH	MH	MH	MH	MH
MB	MB	MB	MB	MB	MB	MB	MB	MB	MB	MB	MB	MB	MB	MB	MB	MB	MB	MB	MB

Figu re 6-1: ISP of the IBM370 Virtual Memory Space and its Mapping onto
thePDPI0

Appendix B. Example of the Effects of Programming Style on the Percentage of RTM ALU Operation Generated

```
AM2910:=
  begin

! ISPS description of AMD AM2910 microprogram sequencer.

! The AM2910 is a 12 bit microprogram address controller.
! The controller is designed to be used with the AM2901 or the AM2903
! microprocessor slice and external memory.

! Simulation of the AM2910 alone is possible, but emulation of any
! computer systems requires that this description be joined with
! the AM2901 or AM2903 description.
        .
        .
        .


•Address.SourcaSe*ection-{us}

  Y()K11:0> :■
    begin
    IFieqr2 => enable = map.;
    IFieql"6 => enable « vect;
    IFCineq^N2)andrineq^N6) => enable * pL;

!•■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

    fail = (not CCEN) and CC next     ! EXPLICITLY STATED BEFORE
    pass = not fail next          I BEFORE THE DECODE

!•■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

    DECODE I «>
      begin
      ^W0:=J2  :« (Y= SP»O;FULL « 1),
      "1 := CJS :» (IFfail =>(Y « uPC);
            IFpass_5=>(Y_sD;push.O)),
      ^H2:s JMAP:= (Y = D),
      ^H3:= CJP :* OFfaH =>(Y « uPQ;
            IFpass=>{YsrD))_f
          .
          .
          .
```

```
! J. Duane Northcutt  8-5-81
! AM2910 Micro-sequencer

AM2910(OE.not.K>,a.iO,RLD.not.iO¡CCEN.not.iO,CC.not.K>¡Inst.K3^)>,D. K11:0)K15:0> :=
BEGIN

! This is an ISPS description of AMD AM2910 microprogram sequencer.

••Macro.Definitions**

  MACRO Enable: = |0|,           ! Enable Active Low Outputs
  MACRO HiZ  : = TFFFFI,         ' High impedence constant

|••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

  MACRO Pass : = |CCEN.noti OR NOT(CC.noti)),   ! MACRO DERNITION TO
  MACRO Fail: = (NOT(CCEN.noti) AND CC.not.t|,  ! BE INCLUDED INSIDE
                        I THE INTERPRETATION
                        ! OF EACH INSTRUCTION

r••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
            .
            .
            .
            .

•• Address.Source.Setectton* *{us}
  Next.Addr()K11:0> :=
    BEGIN
    DECODE (Insti)=>
      BEGIN
      "O\JZ:= BEGIN

!•.........*........•••••••••••••......••••••••••«•».,•••••••••••••••••••••

          IF (Pass OR Fail) =>    ! INCLUSION OF THE MACRO

!....*....•••••••••••....•.....•.......••....•....••••••••••••••••••••••

            BEGIN
            NextAddr = a,
            SP = 0;
            Full.noto s 1
            END;
          PI. no to s Enable
          END,
            .
            .
            .
```

References

[AMD 81]   AMD.
                   AMD Bipolar Microprocessor Logic and Interface Data Book.

[Atlas 82]   Atlas, Benjamin.
                   Dynamic Instruction Frequency Count.
                   Technical Report, Carnegie Mellon University, Department of Electrical
                        Engineering, 1982.

[Barbacci 80a]   Barbacci, Mario R., Andrew W. Nagle and Dwayne J. Northcutt.
                   An ISPS Simulator.
                   Technical Report, Carnegie Mellon University, Department of Computer Science,
                        1980.

[Barbacci 80b]   Barbacci, Mario R., et al.
                   The ISPS Computer Description Language.
                   Technical Report, Carnegie Mellon University, Department of Computer Science,
                        1980.

[Barbacci 81]   Barbacci, Mario R.
                   The Register Transfer Machine.
                   Technical Report, Carnegie Mellon University, Department of Computer Science,
                        1981.

[Barbacci 82]   Barbacci, Mario.
                   Personal Communications.

[Bell and Newell 71]
                   Bell, Gordon, and Newell, Allen.
                   Computer Structures, Principles and Examples.
                   McGraw Hill, 1971.

[Birmingham 82]   Birmingham, William and James Saulnier.
                   P.mbs Final Report for VLSI.

[Flores 63]   Flores, Ivan.
                   The Logic of Computer Arithmetic.
                   Prentice Hall, Inc., 1963.

[Halstead 77]   Halstead, Maurice H.
                   Elements of Software Science.
                   El Sevier North Holland, Inc., 1977.

[Meyers 80]   Meyers, Glenford J.
                   Digital System Design with LSI Bit Slice Logic.
                   John Wiley and Sons, 1980.

[Mick and Brick 80]
        Mick, John and Jim Brick.
        Bit Slice Microprocessor Design.
        McGraw Hill, 1980.

[Northcutt 82]   Northcutt, Duane.
        Personal Communications.

[Siegel 82]   Siegel, Zary.
        Personal Communications.

[Siewiorek 82]   Siewiorek, Daniel P.
        Personal Communications.