An interactive simulator generating system.

Ramon Tan
AN INTERACTIVE SIMULATOR GENERATING SYSTEM

by

Ramon Tan

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Computer Science

Lehigh University
1977
This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

April 18, 1977
(date)

Richard J. Uichelli
Professor in Charge

Arthur E. Pitcher
Chairman of Department
ACKNOWLEDGMENTS

I wish to thank the support provided by the Software Development Group of the Mathematics Department of Lehigh University. Sincere thanks are also due Messrs. Mueller and Johnson of the Colorado State University, the original authors of the GEN systems, who provided in source form the original versions that were basis for the interactive versions. Finally, the assistance, advice and encouragement of Mr. Richard J. Cichelli as thesis adviser is gratefully acknowledged. Any errors, inaccuracies or bugs remaining in the present work are the sole responsibility of this author.
# TABLE OF CONTENTS

Abstract ........................................................................................................ 1

CHAPTER I: Interactive simulation using the GEN systems

1.1 Modified ASM/GEN and SIM/GEN ...................................................... 2
1.2 Available commands from the generated simulator ......................... 6
1.3 Example using the modified GEN systems .................................... 7

CHAPTER II: The GEN systems

2.1 The generating process ................................................................. 15
2.2 The ASM/GEN generating process .............................................. 16
2.3 The SIM/GEN generating process ................................................ 22
2.4 The MEMORY module generator ................................................. 23
2.5 The DECODE module generator .................................................. 26
2.6 The XECUTE module generator ..................................................... 28
2.7 The IDL processor ........................................................................ 32

CHAPTER III: Using SIM/GEN: A tutorial

3.1 Introduction .................................................................................. 37
3.2 The Motorola 6800 microprocessor .............................................. 38
3.3 The Motorola 6800 instruction set ................................................ 40
3.4 MEMORY module for the M6800 ................................................. 44
3.5 DECODE module for the M6800 .................................................. 52
3.6 Class 2 and the XECUTE module ................................................. 55
3.7 The condition code subroutines ..................................................... 62
3.8 Two's complement arithmetic ....................................................... 66
3.9 XECUTE module for the M6800 .................................................... 70
APPENDIX A: ASM/GEN input for the M6800 microprocessor

APPENDIX B: DECODE input for the M6800 microprocessor

APPENDIX C: XECUTE input for the M6800 microprocessor (classes 1, 3, 4, 5, 6 & 7)

LIST OF ILLUSTRATIONS

ASM/GEN generator run for the M6800 ........................................... 8
Sample assembly using generated assembler .............................. 11
Sample simulation run using generated simulator ..................... 14
Class 2 input to the XECUTE module ........................................... 57

LIST OF FIGURES

Figure 1 (The Simulator generating process) ......................... 39
Figure 2 (Programming model of the M6800 microprocessing unit) 41
Figure 3 (The M6800 instruction set) ................................. 46
Figure 4 (Special Operations of the M6800) .............................. 50
Figure 5 (MEMORY module generation run) .............................. 53
Figure 6 (DECODE module generation run) .............................. 56
ABSTRACT

Automatic production of microprocessor assemblers and simulators has recently been realized by the ASM/GEN and SIM/GEN systems of Mueller and Johnson at the Colorado State University. A modified version of these two systems to include an interactive simulator generating capability is the subject of this paper.

Chapter I presents the rationale behind the modification of the original systems of Mueller and Johnson. The interactive and run-time debugging features available from a generated simulator of the modified versions is illustrated.

Chapter II is a detailed discussion of the generation techniques employed by these systems to produce the desired assembler or simulator. The high-level language (FORTRAN) realization of the microprocessor instruction emulation code is presented.

Chapter III is a tutorial on using the SIM/GEN system for prospective users wishing to generate a simulator. An actual and existing microprocessor, the Motorola 6800, is used as example and the necessary steps involved in the generation of a simulator for this microprocessor are covered. The CDC 6400 host machine is assumed for this actual case study.
1.1 Modified ASM/GEN and SIM/GEN

ASM/GEN and SIM/GEN are a software system comprised of a set of FORTRAN program writer modules designed to generate microcomputer assemblers and simulators. Briefly, ASM/GEN is a program that is capable of producing a complete assembler for any specified microcomputer instruction set. The resulting assembler uses 2-pass, absolute assembly and offers the usual macro facility, conditional assembly and a set of useful pseudo-directives. The counterpart system, SIM/GEN, produces a simulator for any specified microcomputer, excluding I/O interfaces and timing considerations. The generated simulator executes in batch mode, but is capable of providing runtime diagnostics in the form of a machine status dump. This capability relates to the *TRACE n pseudo-directive from the assembler generated by ASM/GEN. The *TRACE pseudo-directive is used during the assembly process to denote a call for a machine status dump at a certain point in a program. An instruction is said to be TRACEd if this pseudo-directive immediately precedes it. A machine status dump, depending on the value of n, is performed before the TRACEd instruction is executed. Thereafter, the status dump continues to be performed at the end of each instruction execution cycle, until a new TRACE level is encountered.

Specifically, a TRACE level number of n = 0 disables the TRACE option. This means that no further machine status dumps are to be performed. A level of n = 1 causes the contents of the program
counter and the instruction register to be displayed. When \( n = 2 \), the level 1 displays are performed, and in addition, all registers of the simulated machine are also displayed. When \( n = 3 \), a complete machine status dump is performed. This level of tracing displays entire sections of the memory configured, the address and data bus (a basic assumption of SIM/GEN for all data transfers), time elapsed and total instruction count, where the last 8 instructions were executed, as well as the displays at the lower level options.

It is our belief that while the above-mentioned diagnostic capability is no doubt informative and useful to the debugging of the simulated program, it suffers from certain inadequacies. First of all, the programmer must determine ahead of time where his TRACE points should be inserted. This implies that at assembly time, the programmer must decide on where he wishes to obtain machine status dumps. Secondly, because the simulator executes in a batch mode, the trace points are binding at simulation time. No capability for run time control exists on the part of the programmer. Finally, excessive printouts resulting from the machine status dumps will most likely occur. This is especially true of TRACEd loops that become indefinite, or for complete status dumps (trace level 3) in which the entire memory is displayed in addition to the microprocessor state variants. More so considering the fact that the machine status dump is performed after every instruction execution cycle. The need to facilitate increased programmer control over the program simulated and for better
run-time debugging tools has led to a modification of the ASM/GEN and SIM/GEN generating systems.

It was first decided that if a simulator was to be truly useful to the microprocessor development cycle, programmer interaction must exist during simulation time. Hence, SIM/GEN was modified to generate a simulator that could interact with the programmer at a terminal. Next, to allow for monitoring at run-time, a new level number was introduced to the *TRACE n pseudo-directive available from the generated assembler in ASM/GEN. A level number of 4 implies a "breakpoint" in the usual understanding of the term: when an instruction is encountered during execution that was trapped by a *TRACE 4 pseudo-operation, execution is suspended and programmer gains control. At this point, the programmer may input any one of the available commands in the simulator. The available commands allow for:

- displaying any section of memory;
- displaying any microprocessor architectural component, such as the program counter, the instruction register, the address or data bus, registers and so forth;
- changing the contents of any RAM memory location;
- changing the contents of any microprocessor architectural component;
- insert or remove breakpoints, or alter the level number of a breakpoint (note, that only a level number equal to 4 will cause suspension of program execution while all other level
numbers simply cause a machine status dump and proceed with execution);

- resume or halt execution, at the next logical location or at some other desired location;

The complete list of commands that may be used by the programmer using a simulator generated from SIM/GEN appears in the next section. In view of the ability to selectively dump sections of memory, the level 3 trace was reduced to a machine status dump which included all the displays mentioned earlier, except for the display of all of memory. Lastly, all machine status displays (levels 1 - 3) would be performed only at the encountered instruction, and not at every instruction thereafter.

The net result of a modified SIM/GEN is a simulator that offers the programmer a better debugging tool in the microprocessor software development process. While retaining the machine status displays previously available from the batch simulator, a simulator generated from the modified SIM/GEN will also allow for the interactive features described. Program monitoring and control have been achieved considerably.
1.2 Available commands from generated simulator

A separator is either a blank or a comma. Values are all assumed to be hexadecimal, while register numbers decimal.

<table>
<thead>
<tr>
<th>Command</th>
<th>Explanation/syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>display address and data bus;</td>
</tr>
<tr>
<td>PC</td>
<td>display program counter;</td>
</tr>
<tr>
<td>IR</td>
<td>display instruction register;</td>
</tr>
<tr>
<td>ST</td>
<td>display hardwired stack;</td>
</tr>
<tr>
<td>SRn</td>
<td>display special register n (no separator);</td>
</tr>
<tr>
<td>GRn</td>
<td>display general register n (no separator);</td>
</tr>
<tr>
<td>T</td>
<td>display total time elapsed, no. of instructions;</td>
</tr>
<tr>
<td>H</td>
<td>display available commands;</td>
</tr>
<tr>
<td>S</td>
<td>halt simulation run;</td>
</tr>
<tr>
<td>G</td>
<td>resume simulation run at current value of program counter;</td>
</tr>
</tbody>
</table>

/ n       - remove the nth breakpoint (separator required, default is n=current breakpoint encountered);
* addr n  - insert an nth level (n between 1 and 4, default=4) *TRACE at addr;
L         - list all breakpoints and their level numbers;
.         - display current location;
..        - display current location and next;
...       - display current and next 2 locations;
D addr1 addr2 - display memory locations addr1 to addr2, inclusive; addr2 is optional (separators required);
C arg hexval - change arg to the hexadecimal value hexval; arg may be any one of the ff:

AS = address bus,
DS = data bus,
PC = program counter,
SP = stack pointer,
ST = top of hardwired stack,
SRn = special register n,
GRn = general register n;

If not any of the above, arg is assumed to be a hexadecimal value denoting some memory location to be altered to contain hexval;

Sixteen (16) breakpoints is the maximum number allowed for by the generated simulator.
1.3 Example using the modified GEN systems

An assembler for the Motorola 6800 microprocessor may be generated using the set of inputs shown in Appendix A. The generation run listing is shown on the following pages. For the sake of clarity, the translation classes specified to ASM/GEN are based on the 7 different addressing modes of the Motorola 6800 microprocessor (with accumulator and implied addressing combined into class 1, and with immediate addressing broken down into 2 translation classes: class 4 for 1-byte operands, class 5 for 2-byte operands). In the last chapter, a tutorial is presented for generating the simulator. The intention of this section will be to show the reader what the generated simulator looks like.

The subroutine shown is assembled using the generated assembler and is taken from (3). It is entered with the Index Register, IX, containing the address of the most significant byte of the multiplicand. Register A contains the most significant byte of the multiplier and register B the least significant byte of the multiplier. The multiplicand and multiplier are treated as 16-bit unsigned numbers. A 16-bit product is generated in A and B. If the product is larger than 16 bits, only the least significant bits are retained.

Algorithm used is as follows:

Initial partial product (PP) = multiplier.

Repeat the following 16 times:

MULT10: Shift left, arithmetic, PP.
THE GENERATED ASSEMBLER HAS 7 CLASSES AND A 8 BIT WORD SIZE

**OBJECT WORD SPECIFICATIONS**

**TRANSLATION CLASS: 1**

- NO. OF FIELDS: 1
- NO. OF MEMORY WORDS: 1
- RIGHT-MOST BIT POSITION AND WIDTH OF EACH FIELD:
  1. (0, 8)

**TRANSLATION CLASS: 2**

- NO. OF FIELDS: 2
- NO. OF MEMORY WORDS: 2
- RIGHT-MOST BIT POSITION AND WIDTH OF EACH FIELD:
  1. (8, 8)
  2. (0, 8)

**TRANSLATION CLASS: 3**

- NO. OF FIELDS: 2
- NO. OF MEMORY WORDS: 2
- RIGHT-MOST BIT POSITION AND WIDTH OF EACH FIELD:
  1. (8, 8)
  2. (0, 8)

**TRANSLATION CLASS: 4**

- NO. OF FIELDS: 2
- NO. OF MEMORY WORDS: 2
- RIGHT-MOST BIT POSITION AND WIDTH OF EACH FIELD:
  1. (8, 8)
  2. (0, 8)

**TRANSLATION CLASS: 5**

- NO. OF FIELDS: 2
- NO. OF MEMORY WORDS: 3
- RIGHT-MOST BIT POSITION AND WIDTH OF EACH FIELD:
  1. (16, 8)
  2. (0, 16)

**TRANSLATION CLASS: 6**

- NO. OF FIELDS: 2
- NO. OF MEMORY WORDS: 2
- RIGHT-MOST BIT POSITION AND WIDTH OF EACH FIELD:
  1. (8, 8)
  2. (0, 8)

**TRANSLATION CLASS: 7**

- NO. OF FIELDS: 2
- NO. OF MEMORY WORDS: 3
- RIGHT-MOST BIT POSITION AND WIDTH OF EACH FIELD:
  1. (16, 8)
  2. (0, 16)

**GENERAL MNEMONICS**

- NONE DEFINED

**TRANSLATION CLASS 1 MNEMONICS**

<table>
<thead>
<tr>
<th>INX</th>
<th>8</th>
<th>ARRA</th>
<th>19</th>
<th>ASRA</th>
<th>47</th>
<th>ASRA</th>
<th>47</th>
</tr>
</thead>
<tbody>
<tr>
<td>RORA</td>
<td>46</td>
<td>CHA</td>
<td>11</td>
<td>SBA</td>
<td>10</td>
<td>TAP</td>
<td>6</td>
</tr>
<tr>
<td>TKS</td>
<td>15</td>
<td>NEGA</td>
<td>40</td>
<td>NEGA</td>
<td>50</td>
<td>CLRA</td>
<td>4F</td>
</tr>
</tbody>
</table>
MULT16 - MULTIPLY 16 BIT UNSIGNED NUMBERS.

ENTRY .. (ACCA) = M.S.B. OF MULTIPLIER,

(ACCB) = L.S.B. OF MULTIPLIER,

(IX) = ADDRESS OF M.S.B. OF MULTIPLICAND,

EXIT .. (ACCA) = M.S.B. OF 16-BIT PRODUCT,

(ACCB) = L.S.B. OF 16-BIT PRODUCT.

*CREF
*TITLE MULT16
*DS 128 :1ST MEMORY SEGMENT (RAM)
*BEGIN 80+16
*TRACE 4

MULTI
PSH B :KEEP ...

PSHA : MULTIPLIER

LDA AX 1 :KEEP ...

PSH A

PSHA :MULTIPICAND

LDA AX 0 :PUT COUNT

PSHA

TSX :STACK NOW READY

LDA AX 3 :ACCA = M.S.B. OF MULTIPLIER

MULTI8
ASLB :ARITH. LEFT SHIFT

ROL A : THE MULTIPLIER

ASL X 2 :ARITH. LEFT SHIFT MULTIPICAND,

ROL Y 1 : (M.S. BIT NOW IN -C-)

9CC MULT20-PC-2 :BRANCH IF CARRY CLEAR

ADD AX 4 :ADD MULTIPLIER TO

ADD AX 3 : PARTIAL PRODUCT

DECY 0 :DECREMENT COUNTER

9GT MULT16-PC-2 :RETURN FOR MORE

INS 31 :CLEAR

INS 1 :UP

INS 1 : THE STACK

RTS :RETURN
*END
Examine next bit of multiplicand (starting with most significant bit).
If 0, branch to MULT10.
Add multiplier to PP.
Branch to MULT10.

To insure that "entry" into the subroutine is proper, a *TRACE 4 directive is inserted at the first executable instruction at assembly time (in our case of the Motorola 6800, this is at location 128 decimal = 80 hexadecimal = first ROM word). The following conditions must be satisfied:

1. the stack pointer (SP) must point to a RAM location as the next available position on the stack;
2. the return address must be kept on the stack with the high order byte above the low order byte (a 16-bit address);
3. multiplier and multiplicand test values must be established using the change command;

The return address (arbitrarily we select the subroutine itself) is stored in locations 126 and 127 decimal (7E and 7F hexadecimal), so the stack pointer must point to 125 (7D hexadecimal). When execution reaches location 8E, we wish to view the status of the stack and the contents of the stack pointer (SP) and index register (SR2, special register 2 as declared to SIM/GEN). So a breakpoint is inserted. The last breakpoint will merely call for a level 2 machine status dump and resume the execution. This status dump will display all the special registers including the A and B accumulators, which were declared as special registers 0 and 1 respectively. Note that this subroutine uses
*TRACE 4 (DEBUG)
BREAKPOINT NO. 1 AT: 80
C SP 7D
SP
MEMORY BASED STACK POINTER= 7D
SR2
SREG 2= 0
C 1 11
D 0 1
LOCATION CONTENTS
  0  0
  1  11
C SR1 A3
SR0
SREG 0= 0
SR1
SREG 1= A3
* BE 4
BREAKPOINT INSERTED.
* 9E 2
BREAKPOINT INSERTED.

L
NO.  BREAKPOINT  *TRACE LEV.
1.  80      4
2.  8E      4
3.  9E      2

G
*TRACE 4 (DEBUG)
BREAKPOINT NO. 2 AT: 8E
SP
MEMORY BASED STACK POINTER= 7B
D 72 7F
LOCATION CONTENTS
  77  0
  78  0
  79 10
  7A  0
  7B 11
  7C  0
  7D A3
  7E  0
  7F  0
C 7F B0
D 7F
LOCATION CONTENTS
  7F  80
/
BREAKPOINT AT 8E REMOVED.
G
*TRACE 2 (PC+REGISTERS)

.PROGRAM COUNTER= 9E
LAST INSTRUCTION= F0
(SPECIAL REGISTERS)
SREG 0= A
SREG 1= D3
SREG 2= 79
SREG 3= 0
SREG 4= 0
SREG 5= 0
SREG 6= 1
SREG 7= 0
SREG 8= 0
RETURN FROM SUBROUTINE ♦
*TRACE 4 (DEBUG)
BREAKPOINT NO. 1 AT: 80
S
STOP
COMMAND= - 14 -
the memory-based stack as a work area (5 bytes following the return address).

2.1 The generating processes

This chapter is concerned with the generation process of ASM/GEN and SIM/GEN. It is best to begin the description of the generation process by directly quoting the original authors:

The actual program to be generated was written as a "skeleton routine", that is, the program was complete except for the constants defining the target machine's architectural dimensions and particulars that are "filled-in" from the user input. Within the skeleton are "markers" indicating that user input is required to complete a missing part at the marked point. The generating process then becomes a matter of first copying the skeleton to some disk file, and then having the generator transfer the skeleton to another file, card by card, filling in the missing parts with user-provided data at the appropriate places. The latter file would contain the complete module upon termination. Diagnostics are included to aid in correction of syntax errors, however, cases of misformatted values can result in a complete module with incorrect machine specifications.

The generating process therefore consists of a single, strictly sequential pass over the associated skeleton units, inserting the appropriate FORTRAN code to the resulting file between "markers". An essential characteristic of the generators is that it is always aware of the context in which the generation process is in. This is largely dependent on the ordering of the routines within the skeleton unit. In point of fact, the program structure of the GEN systems reflects this to a very high degree.

Another characteristic is the separation of the target machine specifics (termed by the original authors as the variants) from the machine-independent aspects of the generated program (invariants). The
variants are user input at generation time while the invariants reside on the skeleton units. Only those routines within the skeleton unit that are target-machine dependent in some aspect need be "marked", signifying to the generator that some insertion (or in some cases, a skip) into the current routine is required. It may come in the form of a COMMON statement, or a DATA statement (as is frequently the case), or several lines of FORTRAN statements (typically, assignment or computed go-to statements). A dollar sign '$' is used for a marker.

2.2 The ASM/GEN generating process

There are 2 skeleton units to this generator. The first is nearly a straightforward and completely pre-written skeleton, with only the last 2 that need "filling in". The first routine is a 1-parameter subroutine (OPCODE) which in the formal view represents the semantic routine invoked by the lexical analyzer when it is sensed that the current input symbol is that of an operation code mnemonic. To be more specific, the generated subroutine OPCODE actually takes the form of a computed go-to statement in FORTRAN, each branch of which is a CALL statement to the appropriate translation class processing subroutine. Recall that the user input to the ASM/GEN system also requires a grouping scheme such that in any group (called a translation class) all instructions shall have exactly identical instruction length, operation code length, total number of operand fields, as well as operand field length. That is, an instruction may be thought of simply as being made up of a string of bit-encoded fields and generally as
OPCODE OPR1 OPR2 ... OPRn

In every translation class, these fields would have identical bit-width for all instructions. Hence, all that needs to be known from the user are the OPCODE values and bit-widths for all fields. Returning to the 1-parameter subroutine OPCODE, we are now aware of its function: invoke the subroutine to process (i.e., extract operand fields and generate the bit string) an instruction, given the opcode value. What is actually passed is the position of the opcode mnemonic in the symbol table (which is, of course, built up during the generation). The computed go-to statement will actually be preceded by a statement that calculates the translation class number for the opcode in question, using the position parameter passed. There will be as many branches in the computed go-to as there are translation classes specified during generation. The subroutine for processing a particular translation class has yet to be generated of course, and this is where the other skeleton unit comes in.

The other routine remaining within the first skeleton unit requiring code insertion is the initializing routine which takes the form of a BLOCK DATA. Into this routine is inserted a series of DATA statements that initializes 2 tables: the first (collectively grouped under the COMMON /SYMTAB/ block) is the global symbol table to be used by the lexical analyzer for reserved symbol recognition and also for storing user-defined symbols during assembly; the second (grouped
under the COMMON /OPMAP/ block) is an opcode-to-translation-class-number mapping table for use by the subroutine OPCODE just mentioned.

The second skeleton unit constitutes the main body of a subroutine corresponding to a translation class processor (to reiterate, the process being extract operand fields and generate bit string). It is therefore "scanned over" as many times as there are specified translation classes. This is not to be confused with our earlier statement that the generating process is a single, strictly sequential pass over the skeleton unit. What must be observed is that if 7 translation classes were specified by the user, then seven translation class subroutines: CLAS1, CLAS2 ... CLAS7 must be generated. There is 1 parameter passed to each of these and is the opcode value corresponding to the opcode mnemonic, sensed by the lexical analyzer, which invoked the OPCODE subroutine (passing to it the opcode value's position in the /OPMAP/table), which in turn calculated the class number and invoked the corresponding CLAS subroutine.

The code inserted into this skeleton unit consists of first generating a subroutine header of the form SUBROUTINE CLASi, where i is the class number, followed by 3 DATA statements (at second insertion) specifying the instruction length, opcode and operand field bit-width and relative bit position. The remainder is completely identical in all translation classes except when the "field-swap" option is used. In this special instance, an extra 5 lines of code to perform the swap are included.
To summarize, the ASM/GEN generating process consists of the following steps:

(1) Read user inputs

(2) Transfer skeleton unit 1 until marker

(3) Do the following NCLAS times, where NCLAS is the number of translation classes that the user specified in his input:
   (3.1) Position to start of skeleton unit 2
   (3.2) Output a SUBROUTINE CLASi header, where i is the iteration count for this loop;
   (3.3) Transfer skeleton unit 2 until marker;
   (3.4) Output DATA statements to initialize the instruction length, opcode and operand field length and bit-width;
   (3.5) Output field-swap code, if user specified it for this class; else skip over it;
   (3.6) Transfer skeleton unit 2 until marker;

(4) Transfer skeleton unit 1 until marker;

(5) Output computed go-to statement with NCLAS branches, each branch being of the form:
   
i CALL CLASi (OPMAP (POS))
   
where i progresses from 1 to NCLAS;

(6) Output and END statement to complete the generation of the subroutine OPCODE;

(7) Transfer skeleton unit 1 until marker;

(8) Output COMMON /OPMAP/ statement to dimension this table,
based on the total number of opcodes; dimension will be 2
times opcode (1 for opcode value and 1 for class number
for this opcode value);

(9) Transfer skeleton unit 1 until marker;

(10) Output DATA statement to initialize the symbol tables
/OPMAP/ and /SYMTAB/;

(11) Output END statement to wrap up the BLOCK DATA;

In the outline above, "transfer" implies the line by line transfer
from the skeleton unit to the resulting file, while "output" implies a
formatted WRITE to the resulting file. The special case at (3.5) in-
volves some 5 lines of FORTRAN code to accommodate the field-swap
option for certain translation classes. The modified version selec-
tively generates these lines in the sense that they are "skipped over"
(skeleton unit is read, but not followed by a write to the resulting
file) if the option is not chosen. This results in a shorter sub-
routine for the translation class under consideration. A pictorial
view of the steps involved in the ASM/GEN generation is shown on the
following page.
skeleton unit 1

PROGRAM ASSEM

\$ 

SUBROUTINE OPCODE

\$ 

BLOCK DATA

\$ 

DATA

\$ 

resulting file

PROGRAM ASSEM

\$ SUBROUTINE CLAS1 (3.2)

\$ DATA bit-width

\$ (field-swap code) \$ (field-swap code)

\$ (generate bit-string code)

\$ 

RETURN

\$ END

\$ 

skeleton unit 2

\$ (FORTRAN declarations)

\$ 

SUBROUTINE OPCODE

\$ 

GO TO (1,2, ...)

\$ END

\$ 

BLOCK DATA

\$ COMMON /OPMAP/ ... (8)

\$ DATA OPMAP

\$ DATA SYMTAB

\$ END

- 21 -
2.3 The SIM/GEN generating process

The generating process for a complete simulator involves at least 3 different steps and is quite different from ASM/GEN. Following the top-down modular approach, SIM/GEN's generation process is again best described by the original author's statements:

The operation of all modern general-purpose digital computers is based on the repetitive sequence of FETCHing the next instruction, DECODEing it, and invoking the proper operations that EXECUTE the instruction. The initial goal of generating a simulator was divided into three subtasks: Those of generating a FETCH module, a DECODE module, and an EXECUTE module. The FETCH and DECODE are both well-defined and small enough to not have to be broken down further. EXECUTE, however, spans a wide variety of tasks and appeared to be far too extensive to be handled by a single unit.

The technique to treat that EXECUTE phase was to group the instruction set into classes in which all the instructions in a particular class matched identically in their component bit structures. That is, all instructions with an opcode of length 1, operand field one of length j, operand field two of length k, etc., and which all have m operand fields would be grouped in a single class. The DECODE routine could determine which class the instruction belonged in, branch to that module, and then decoding of the operand parts could be done at the start of the module without regard to which instruction in the module was being executed. This seemed to be an effective solution and required that the user define only one module at a time, totally independent of the remainder of the system.

In the current version of SIM/GEN, these 3 subtasks have been labelled the MEMORY, DECODE and EXECUTE modules, with MEMORY having the identical function of the FETCH module described above (from an earlier version). The basic philosophy behind the generating process
remains the same: transferring the skeleton unit to the resulting file, taking the appropriate action between "markers" in the form of code insertion & deletion (2.1).

In the remainder of this chapter, the term insertion will always be taken to mean the outputting of FORTRAN code to the resulting file. Insertion has already been used at the ASM/GEN generation and is always assumed to occur at the skeleton unit markers ("$" is used throughout). Deletion will be taken to mean the action of "skipping over" portions of the skeleton unit. That is, a read of the skeleton unit not followed by a write (transfer) to the resulting file. Quite often, the word 'skip' will also be used in this context. Deletion is used where the skeleton unit contains segments of code that are mutually exclusive, the segment to be chosen being dependent entirely on the user input specifications. As an example, consider the fact that one of the routines present in the skeleton unit of the MEMORY module manipulates the hardwired stack. If the simulated microprocessor does not have this option, there would be no need to include this routine in the resulting file. A skip over this segment of code is therefore necessary at the point where the stack utilities are generated.

2.4 The MEMORY module generator

Target machine dependent routines are generated by the MEMORY module generator. These routines involve memory references, data transfers to and from memory assuming a bus organization, memory addressing, stack manipulation and machine status displays. The
skeleton unit for this module generator consists of the following routines:

(1) **RDMEM** - function to read contents of memory at address bus to the data bus.

(2) **WRMEM** - subroutine to write contents of the data bus into memory specified by the address bus.

(3) **ROM** - function to check for memory type.

(4) **VIRMEM** - virtual address mapping function.

(5) **MEMDMP** - display memory contents subroutine.

(6) **PUSH** - push data onto stack routine.

(7) **PULL** - pull data off stack routine.

(8) **GETSTK** - return data off hardwired stack routine.

(9) **STATUS** - machine status display routine.

(10) **DGREG** - display general register routine.

(11) **DSREG** - display special register routine.

(12) **DSTK** - display hardwired stack routine.

(13) **CHANGE** - alter memory or hardware component routine.

(14) **IHEX, BINSER, XOR** - display utilities routines.

(15) **BLOCK DATA** - initializing routine.

Routines (1)-(5) deal with memory references and data transfers. The insertions performed on these routines are the memory dimensions, memory word size, memory segment type and memory segment boundaries. These come in the form of **COMMON** statements for the **MEMORY** array, which must be dimensioned to the total number of words that comprise the
simulated memory configuration, and DATA statements for the memory segment boundaries and memory type for each segment. Only FORTRAN declaratives are inserted into these 5 routines. The exception to this is the ROM function. If no Read Only Memory segments were declared, the entire body of the routine is skipped, and replaced by a single assignment statement: ROM = .FALSE.

Routines (6)-(8) are stack manipulation routines. A three-way decision is made by the generator at this point. If no stack facility is declared, these 3 routines are skipped. If a memory-based stack is declared, the first 2 are generated, but the third skipped, and if a hardwired stack is declared, all 3 routines are generated. Code insertion involves either a memory data transfer sequence, or a hard-wired stack data transfer. In the former case, the PUSH routine uses the bus structure to store data onto the stack (which is memory based) and so the following code is inserted:

```
data bus    = word to be pushed (parameter)
address bus = stack pointer
CALL WRMEM.
```

If a hardwired stack was declared, the code insertion first requires that a COMMON statement be generated to allocate a separate area of storage (named STACK) for the hardwired stack. It is naturally dimensioned to the stack depth which must be specified by the user. Then the simple assignment statement

```
STACK (stack pointer) = word to be pushed (parameter)
```

is inserted. In both cases, the decrementing of the stack pointer
following the actual push operation is part of the skeleton routine. In fact, that line of code immediately follows the marker where the PUSH code is inserted. A similar situation exists for the PULL routine.

Routines (9) to (13) involve combined insertions and deletions. Insertions are entirely the FORTRAN declaratives that dimension the register arrays, the memory array and several other counters (i.e., total number of special registers). The deletions are mainly concerned with the display function at the interactive level. For example, the body of the DSTK subroutine contains 2 segments of code: a first segment loops through the hardwired stack, converting each location to display format, and displays it. The other segment is merely a WRITE statement with a diagnostic message that no hardwired stack was declared for the simulated machine. Depending on the user declaration, the appropriate segment is skipped during the generation. A similar situation exists for the DGREG and DSREG routines. The 3 routines at (14) are print utilities, while the last (15), is used to initialize all the microprocessor's architectural properties in the form of counters, flags, and arrays. This completes our discussion of the MEMORY module generator.

2.5 The DECODE module generator

Generated by the DECODE module generator are:

(1) LOADRM - absolute loader to read load file created by the generated assembler and begin execution.
(2) FETCH, OPRDEC - instruction fetch routines.

(3) TRACE - trace manager that checks for occurrences of *TRACEd locations in the simulated program.

(4) GETKH, GETSYM, TYPSYM, EQL, GETHEX, NUMER, NUMERH, SORT, XCHANG - utilities required to interactively communicate with a user at a terminal. These routines were additions in the modified SIM/GEN and have to do with the command processing for a *TRACE 4 breakpoint.

(5) DECODE - routine to extract instruction opcode and branch to the proper execution class.

Code insertion occurs only at one point in the skeleton unit and is at the subroutine DECODE. This parallels the 2 routines in the first skeleton unit of the ASM/GEN system: OPCODE and BLOCK DATA. As a matter of fact, they are identical: the table generated by the DECODE module is also an opcode-to-execution class mapping function. This table is used to calculate the execution class (translation class equivalent of SIM/GEN) number after which the branch to the proper execution class is taken. Like subroutine OPCODE of the ASM/GEN skeleton unit, a computed go-to statement is generated, each branch of which is a call to the execution class processor (as against translation class processor in ASM/GEN). It is in the execution class processing subroutine where the simulation of the fetched instruction takes place. These subroutines are generated by the third and last module generator of the SIM/GEN system -- the XECUTE module.
Between the code insertion for the opcode table and the generated computed go-to, are also inserted several lines of FORTRAN code that first extract the opcode from the instruction register. Different lines of code will appear, depending on whether the simulated machine has fixed-size or variable-size operation codes. In any case the extract code is followed by a calculation of the execution class number (using a binary search routine BINSER, generated in the MEMORY module, on the inserted opcode table). This is then followed by the branch to the proper execution class.

2.6 The XECUTE module generator

In this last component of the SIM/GEN system are produced the emulation code for a certain execution class of instructions. As previously noted, an execution class to SIM/GEN is what a translation class is to ASM/GEN, and both names really mean the same thing. What is quite different to the generators is the processing that follows: ASM/GEN generates the necessary code to produce the bit-string for the assembled instruction, while SIM/GEN must generate the code to simulate the instruction. The latter task is certainly a more complicated one. An Instruction Definition Language (IDL) was designed to allow the user to describe a microprocessor instruction set to SIM/GEN. An IDL processor, which is part of the XECUTE module generator, translates the user's IDL statements into the equivalent FORTRAN statements. We quote from the SIM/GEN user's reference manual:
"IDL is an assembly-like language consisting of microlevel operators and architectural component operands which enable the user to emulate the functions of a microprocessor's instruction set. It can be viewed as a simulator microprogramming language in that it provides a convenient medium for specifying the micro-instructions whose results define the function of the machine instruction. Its operations represent those typically found in the functional unit(s) of microcomputer Arithmetic-Logic Units in addition to simple data transfers. The operands offered are typical storage elements, some with a dedicated duty (such as the Program Counter and Stack Pointer Register) and others more flexible and general-purpose such as General Registers, Address and Data Busses and Memory."

"The power of using IDL for instruction microprogramming lies in the high degree of similarity between its notations and those found in a representative vendor microprocessor description manual. It, therefore, allows the user to readily transfer the vendor's description of the instruction to SIM/GEN which greatly simplifies the process. IDL descriptions are totally sequential, with a conditional IF construct provided for conditional execution of blocks of IDL statements. Subroutines may be defined to eliminate the need for coding redundant functions common to groups of machine instructions (e.g., status bit settings on Arithmetic Logic instructions, address computations for external referencing, etc.)."

The skeleton unit to the XECUTE module generator parallels the second skeleton unit to ASM/GEN. A difference exists at the processing level of the generators: whereas ASM/GEN produces the translation class subroutines in a single run, the XECUTE module generator of SIM/GEN can only produce one execution class subroutine per run. So if n execution classes are involved, n runs of XECUTE are necessary to complete the generation. The XECUTE processing consists of the following steps:

(1) Output SUBROUTINE CLASi header, where i is the execution
class number in question.

(2) Transfer skeleton unit until marker.

(3) Insert user-input dependent declaratives (the only ones being general and special register count, which are part of the input to XECUTE).

(4) If this execution class has no operand fields, skip until marker is encountered. Otherwise, transfer skeleton unit until marker and generate the index calculation code.

(5) Generate a computed go-to statement with m branches, if there are m instructions in this class.

(6) Process instruction definitions.

(7) Process subroutine definitions.

The code that is either skipped over or transferred to the resulting file at step (4) is the operand extraction code. An assumption here is that the generated execution class subroutines are entered only with the opcode fetched into the instruction register. So operand fields must first be extracted. This is followed by the index calculation code for the instruction to be simulated. The branch to the simulation code for that instruction is then taken by way of the computed go-to statement using the earlier calculated index. When all instructions have been processed, user-defined subroutines are checked and processed similarly by the IDL processor. Each user-defined subroutine will result in exactly 1 FORTRAN subroutine. A diagram of the XECUTE module generation is shown below:

- 30 -
<table>
<thead>
<tr>
<th>XECUTE skeleton module</th>
<th>resulting file</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SUBROUTINE CLASi</td>
</tr>
<tr>
<td></td>
<td>(FORTRAN declarations)</td>
</tr>
<tr>
<td></td>
<td>(extract operand code)</td>
</tr>
<tr>
<td></td>
<td>(calculate instruction index code)</td>
</tr>
<tr>
<td></td>
<td>GO TO (2, 3, ... m) index</td>
</tr>
<tr>
<td></td>
<td>(instruction emulation code generated by the IDL processor)</td>
</tr>
<tr>
<td></td>
<td>END</td>
</tr>
<tr>
<td></td>
<td>(user-defined subroutines in FORTRAN equivalent)</td>
</tr>
</tbody>
</table>

(1) SUBROUTINE CLASi

(2) (FORTRAN declarations)

(3) (extract operand code)

(4) (calculate instruction index code)

(5) GO TO (2, 3, ... m) index

(6) (instruction emulation code generated by the IDL processor)

(7) (user-defined subroutines in FORTRAN equivalent)
2.7 The IDL processor

Central to the simulator generating process is the generation of FORTRAN statements that emulate a microprocessor instruction. Since user input consists of IDL statements, an IDL syntax recognizer and translator is required. This section will present the generation techniques involved in the IDL processing.

The first component to the IDL processor comes in the form of the scanner for IDL operands (note, at this point, that IDL operations are recognized separately and at another higher logical level), implemented as subroutine OPSCAN in the XECUTE generator program GENXEC. When called, OPSCAN will translate an input token, assumed to be an IDL operand, into its FORTRAN form. It is worth recalling that the card by card image processing assumption simplifies the task somewhat. The following table shows the generated outputs corresponding to the available IDL operands:

<table>
<thead>
<tr>
<th>IDL operand</th>
<th>generated output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABUS</td>
<td>ABUS (address bus)</td>
</tr>
<tr>
<td>DBUS</td>
<td>DBUS (data bus)</td>
</tr>
<tr>
<td>GREGi</td>
<td>GREG (i + 1) (general register i)</td>
</tr>
<tr>
<td>GREG.i</td>
<td>GREG (OPR(i+1)) (general register number at operand field i)</td>
</tr>
<tr>
<td>IMMi</td>
<td>OPR (i + 1) (immediate operand field i)</td>
</tr>
<tr>
<td>PC</td>
<td>PC (program counter)</td>
</tr>
<tr>
<td>STACKP</td>
<td>STACKP (stack pointer)</td>
</tr>
<tr>
<td>STACK</td>
<td>STACK (STACKP) (top of hardwired stack)</td>
</tr>
<tr>
<td>TEMPi</td>
<td>TEMP (i + 1) (temporary register i)</td>
</tr>
<tr>
<td>SREGi</td>
<td>SREG (i + 1) (special register i)</td>
</tr>
</tbody>
</table>

There is a current character pointer into the card image being processed, and is moved forward or backward depending on the processing
stage involved. The IDL \textbf{operator} recognizer is simply a checker for those keywords that indicate an IDL operation. Every IDL statement begins with an IDL operator, such as MOVE, ADD, IF, etc. (see page 109, manual, for a complete list).

Consider the simple IDL statement

\texttt{ADD SREG0 SREG0 DBUS.}

Assume now that the card image pointer has been left at the blank following 'ADD', so the operator has been sensed at some stage in the processing. The rest of the processing is as follows:

<table>
<thead>
<tr>
<th>card pointer</th>
<th>IDL processor action</th>
<th>generated output</th>
</tr>
</thead>
<tbody>
<tr>
<td>after 'ADD'</td>
<td>CALL OPSCAN</td>
<td>SREG(1)</td>
</tr>
<tr>
<td>after 1st 'SREGO'</td>
<td>emit '='</td>
<td>=</td>
</tr>
<tr>
<td>after 2nd 'SREGO'</td>
<td>emit '+'</td>
<td>SREG(1) + DBUS</td>
</tr>
</tbody>
</table>

The generated FORTRAN statement is thus

\texttt{SREG(1) = SREG(1) + DBUS.}

Constants are also processed by OPSCAN and converted to integer display format. The machine dependent functions like AND, OR, XOR, translate into statement function calls. In what follows, the FORTRAN implementations for each of the IDL operations are described.

\textbf{CLEAR opr.}

This takes the obviously simple FORTRAN statement

\texttt{opr = 0.}

\texttt{COMONE opr1 opr2}

The resulting FORTRAN statement is

\texttt{opr1 = -opr2.}
COMTWO oprl opr2
   This has the FORTRAN statement
   oprl = -opr2 + 1.

CONCAT oprl opr2 (n) opr3
   This has the FORTRAN statement
   oprl = (opr2 * (2 ** n)) + opr3.

DECR opr
   This has the FORTRAN statement
   opr = opr - 1.

DISPLAY text
   This has the FORTRAN statement
   WRITE (6, m)
   m FORMAT (xxH text)

IF oprl relop opr2
   This has the FORTRAN statement
   BOOL = oprl .relop. opr2
   where .relop. is any one of: EQ, NE, GT, GE, LT, LE.
   The subsequent IDL statements processed are of the form
   IF (BOOL) FORTRAN equivalent of IDL statement until an 'ENDIF'
   statement is encountered.

INCR opr
   This has the FORTRAN statement
   opr = opr + 1

MOVE oprl opr2
   This has the FORTRAN statement
opr1 = opr2

PUSH opr

This has the FORTRAN statement

CALL PUSH(opr)

PULL opr

This has the FORTRAN statement

CALL PULL(opr)

Note that PUSH and PULL are subroutines generated by the MEMORY module generator.

READD

This has the FORTRAN statement

ITMP = RDMEM(X)

where X is some dummy argument (RDMEM is a statement function generated in MEMORY).

READI

This has the FORTRAN statements

ABUS = RDMEM(X)

ITMP = RDMEM(X)

SET opr

This has the FORTRAN statement

opr = 2 ** MEMSIZ - 1

Note that MEMSIZ is a COMMONed variable that is initialized in the BLOCK DATA generated by the MEMORY generator module.

SHLC opr1 opr2 n

This has the FORTRAN statement
\[ \text{oprl} = \text{MOD}(\text{opr2} \times (2^{**} n), 2^{**} \text{MEMSIZ}) + \frac{\text{opr2}}{2^{**}(\text{MEMSIZ} - n)} \]

\text{SHLL} \text{oprl} \text{ opr2} \text{ n}

This has the FORTRAN statement

\[ \text{oprl} = \text{MOD}(\text{opr2} \times (2^{**} n), 2^{**} \text{MEMSIZ}) \]

\text{SHRA} \text{ oprl} \text{ opr2} \text{ n}

This has the FORTRAN statement

\[ \text{oprl} = \frac{\text{opr2}}{2^{**} n} + \left(\frac{\text{opr2}}{2^{**(\text{MEMSIZ} - 1)}}\right) \times \left((2^{**} n) - 1\right) \times \left(2^{**}(\text{MEMSIZ} - n)\right) \]

\text{SHRL} \text{ oprl} \text{ opr2} \text{ n}

This has the FORTRAN statement

\[ \text{oprl} = \frac{\text{opr2}}{2^{**} n} \]

\text{WRITED}

This has the FORTRAN statement

\text{CALL WRMEM}

\text{WRITEI}

This has the FORTRAN statements

\text{IDUM} = \text{DBUS}
\text{ABUS} = \text{RDMEM}(\text{X})
\text{DBUS} = \text{IDUM}
\text{CALL WRMEM}

In the Shift instructions, \text{n} need not be a constant; it may be any valid IDL operand. The DUMP operator becomes a CALL STATUS (3) statement, while the HALT is a STOP statement.
3.1 **Introduction**

This chapter will acquaint the prospective user of the SIM/GEN system with the entire simulator generating process by way of an example. A minimum configuration for the Motorola 6800 microcomputer system is chosen for this purpose. Since SIM/GEN does not take into account the I/O interface of any microcomputer system, the minimum configuration mentioned will not include such components as the Peripheral Interface Adapter or the Asynchronous Communications Interface Adapter of the Motorola 6800 family. Furthermore, certain instructions of the "interrupt" type in the instruction set of this microcomputer system will be ignored. The Motorola 6800 system to be considered will therefore consist of 128 bytes of Read/Write memory (RAM), 1024 bytes of Read Only memory (ROM), and the microprocessing unit (MPU). Frequently, reference will be made to particular pages of the SIM/GEN user's manual, so the reader is urged to have this document on hand.

To obtain a complete simulator for any microprocessor under consideration, 3 distinct and separate components of the SIM/GEN system must have been executed to successful completion. These components, called **generator modules**, are the MEMORY, the DECODE and the XECUTE modules. Each is associated with a user specified set of inputs, and a pre-written set of routines called a **skeleton unit**. SIM/GEN requires that the MEMORY and DECODE modules be run successfully at least once, while the XECUTE module must run successfully
a certain number of times depending on the number of execution classes. The details of this will be taken up later. The overall view of SIM/GEN is illustrated in Figure 1.

3.2 The Motorola 6800 microprocessor

The Motorola 6800 microprocessor (hereafter abbreviated M6800) is an 8-bit machine with 2 general purpose, 8-bit accumulators labelled ACCA (the A accumulator) and ACCB (the B accumulator). These are used to hold operands and results from arithmetic-logic operations. There are also 3 special-purpose, 16-bit registers for use by the programmer: the Program Counter (PC) contains the address of the instruction currently being executed; the Index Register (IX) is used to store data or a 16-bit memory address for the indexed mode of addressing; the Stack Pointer (SP) points to a memory location that forms the "top" of a pushdown/pop-up store. In the case of the M6800, this is an area of memory set aside by the programmer for use as a stack. Normally, this must be a random access (Read/Write) type memory. Finally, an 8-bit Condition Code register is also available for the testing of conditions resulting from the last operation. Only the low 6 bits of this 8 bit register are used, whereas the high order two are always set to ones. The testable conditions are as follows:

Bit 0 (C) - the Carry bit from bit 7 of any applicable operand result; set or cleared depending on operation.
FIGURE 1.

Bit 1 (V) - the Overflow bit; set when operation resulted in two's complement overflow, cleared otherwise.

Bit 2 (Z) - the Zero bit; set when result was zero.

Bit 3 (N) - the Negative bit;

Bit 4 (I) - the Interrupt bit; for our purposes, not much will be said of this bit since this has to do with timing.

Bit 5 (H) - the Half-carry bit from bit 3;

The setting or clearing of these bits generally depend on the type of instruction executed. Most often, arithmetic-logic type instructions affect the H, N, Z, V, & C bits. The Half-carry bit, for instance, is affected only by 3 instructions: ADDA (add accumulator with memory), ABA (add accumulators), and ADC (add accumulator with carry). The branch instructions all leave these bits unaffected. The Motorola Programming Manual contains a complete table of Boolean formulas for calculating these bits based on the operand(s) and the result. We shall have occasion to use these in a latter part of this tutorial. The architectural properties mentioned so far are summarized on Figure 2.

3.3 The Motorola 6800 Instruction Set

There are a total of 72 different instructions for the M6800, with 7 addressing modes. Included are binary and decimal arithmetic, logical, shift, rotate, load, store, branch, interrupt (ignored as far as SIM/GEN here is concerned) and stack manipulation instructions. These 7 addressing modes are somewhat arbitrary, because certain
FIGURE 2. (Reprinted from M6800 Reference and Data Sheets)
instructions admit only certain addressing modes. The branch
instructions, for example, admit only the relative mode and no
other. The seven modes are as follows:

**Accumulator addressing:** In accumulator mode, either accumulator
A or B is implied in the operation. These are 1-byte instructions.

**Implied addressing:** These are 1-byte instructions in which the
operand(s) is(are) implied by opcode. These are actually similar to
the Accumulator mode, except that operands other than ACCA or ACCB
are implied. As an example, consider the PULA instruction (PULL data
into ACCA). This is a stack manipulation instruction that will add 1
to the contents of the Stack Pointer, and load the contents of the
memory location pointed to by the SP into accumulator A. Both ACCA
and the stack pointer are implied in the instruction.

**Immediate addressing:** In this mode, the operand is contained in
the second byte of the instruction, except for 3 instructions: LDS
(load Stack Pointer), LDX (load Index Register) and CPX (compare
Index Register). For these 3 instructions, the operand is contained
in the second and third bytes of the instruction. Hence, an instruc-
tion in the immediate mode of addressing may be 2 or 3 bytes in
length as noted.

**Direct addressing:** In direct addressing, the address of the
operand is contained in the second byte of the instruction. This
allows for direct addressing of the first 256 memory locations.
Accordingly, enhanced execution times are achieved by storing the
most frequently used data in these locations (zero through 255). These are 2-byte instructions.

**Extended addressing:** When the address of the operand is greater than 255, i.e., when it is desired to address memory locations that are not among the first 256 locations, extended addressing is used. Naturally enough, these are 3-byte instructions and the address of the operand is made up of the second and third bytes of the instruction: the second byte making up the high order 8 bits and the third byte making up the low order 8 bits of the resulting 16-bit address. This represents an absolute location in memory.

**Indexed addressing:** In indexed addressing, the address contained in the second byte of the instruction is added to the Index Register's lowest 8 bits. The carry is then added to the high-order 8 bits of the Index Register. The result is used to address memory. Note, that this is actually adding an offset that is at most 255 in magnitude. This is an unsigned value. The Index Register is not affected when this mode of addressing is used, since the effective address resulting from the addition of the 8-bit offset is held in some temporary address register. These are also 2-byte instructions.

**Relative addressing:** In relative addressing, the address contained in the second byte is treated as a signed, 7-bit value. This is added to the Program Counter's lowest 8 bits, plus two. The carry or borrow is then added to the high 8 bits. This allows for addressing data within a range of -125 to +129 bytes of the present instruction.
The only instructions that admit this mode (and only this) are the branch instructions. An obvious limitation exists: if one wishes to branch on certain testable conditions (of the condition code register), one cannot directly do so if the desired location is not within the range just described. These are also 2-byte instructions.

The M6800 has fixed-size opcodes of 8 bits/opcode. Taking into account all the valid addressing modes for every instruction, there are actually 197 instructions in the instruction set of the M6800. These are summarized in Figures 3.1 through 3.4. Figure 2 is a table of the symbols used to describe the instructions on Figures 3.1-3.4. Figure 4 explains some of the special instructions. Note the condition code settings on the last column of each instruction.

3.4 MEMORY module for the M6800

The user-required input to the MEMORY module is found on pages 16-23 of the SIM/GEN user manual. There are only 6 cards required from the user for this module, and in the case of the M6800 that we wish to generate a simulator for, our data cards will look like the following (each line represents 1 Hollerith card image):

0
9
8 (0,7F) (80,47F,R)
MEM
MOTOROLA 6800
1

Our first card tells SIM/GEN that the M6800 has no General Registers (see page 63). In SIM/GEN usage, a general register is one capable of being addressed explicitly in an instruction. The M6800
Symbols used in FIGURES 3.1 to 3.4

<table>
<thead>
<tr>
<th>LEGEND:</th>
<th>CONDITION CODE SYMBOLS:</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>Boolean Inclusive OR;</td>
</tr>
<tr>
<td>N</td>
<td>Boolean Exclusive OR;</td>
</tr>
<tr>
<td>Q</td>
<td>Complement of M;</td>
</tr>
<tr>
<td>+</td>
<td>Transfer into;</td>
</tr>
<tr>
<td>-</td>
<td>Bit = Zero;</td>
</tr>
<tr>
<td>*</td>
<td>Byte = Zero;</td>
</tr>
<tr>
<td>Mgp</td>
<td>Contents of memory location pointed to be Stack Pointer;</td>
</tr>
</tbody>
</table>

Note: Accumulator addressing mode instructions are included in the column for IMPLIED addressing.

FIGURE 2 (Reprinted from M6800 Reference & Data Sheets)
## ACCUMULATOR AND MEMORY INSTRUCTIONS

<table>
<thead>
<tr>
<th>OPERATIONS</th>
<th>MNEMONIC</th>
<th>ADDRESSING MODES</th>
<th>BOOLEAN/ARITHMETIC OPERATION</th>
<th>CORR. CODE REG.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>ADDA</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADDB</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Acctors</td>
<td>ABA</td>
<td>EXTRD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add with Carry</td>
<td>ADC</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADCB</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>ANDA</td>
<td>EXTRD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ANDB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit Test</td>
<td>BITA</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BITB</td>
<td>EXTRD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear</td>
<td>CLR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLRA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLRB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compare</td>
<td>CMPA</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CMPB</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compare Acctrs</td>
<td>CRA</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMA</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complement, Z's</td>
<td>NEG</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Nega)</td>
<td>NEGB</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decimal Adjust, A</td>
<td>DA</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DAA</td>
<td>EXTRD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decrement</td>
<td>DEC</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECA</td>
<td>EXTRD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exclusive OR</td>
<td>EOR</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EORB</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Increment</td>
<td>INC</td>
<td>EXTRD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Accntrs</td>
<td>LDAA</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDAB</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Or, Inclusive</td>
<td>DRA</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DRAA</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push Data</td>
<td>PSHAA</td>
<td>EXTRD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSHAB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pull</td>
<td>EXTRD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotate Left</td>
<td>ROL</td>
<td>EXTRD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotate Right</td>
<td>ROR</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RORB</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Left, Arithmetic</td>
<td>AS</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ASLA</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Right, Arithmetic</td>
<td>AS</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ASRA</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Right, Logic</td>
<td>LS</td>
<td>DIRECTORY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LSRA</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Score Acctrs</td>
<td>STA</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STAB</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtract</td>
<td>SUBA</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUBB</td>
<td>INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtract Acctrs</td>
<td>SRA</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtract with Carry</td>
<td>SBC</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transfer Acctrs</td>
<td>TAB</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test Zero or Minus</td>
<td>TST</td>
<td>DIRECT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### TABLE: ADDRESSING MODES

<table>
<thead>
<tr>
<th>IMMEDIATE</th>
<th>DIRECT</th>
<th>INDEX</th>
<th>EXTENDED</th>
<th>IMPLIED</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP ~</td>
<td>OP ~</td>
<td>OP ~</td>
<td>OP ~</td>
<td></td>
</tr>
</tbody>
</table>

### TABLE: BOOLEAN/ARITHMETIC OPERATION

<table>
<thead>
<tr>
<th>(All register labels refer to comments)</th>
<th>S</th>
<th>R</th>
<th>I</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>A</td>
<td>N</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

### TABLE: CORRELATION CODE REG.

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>I</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>A</td>
<td>N</td>
<td>B</td>
</tr>
</tbody>
</table>

---

**FIGURE 3.1** (Reprinted from M6800 Reference & Data Sheets)

- 46 -
### INDEX REGISTER AND STACK MANIPULATION INSTRUCTIONS

<table>
<thead>
<tr>
<th>POINT OPERATIONS</th>
<th>MNEMONIC</th>
<th>IMMED</th>
<th>DIRECT</th>
<th>INDEX</th>
<th>EXTEND</th>
<th>IMPLIED</th>
<th>BOOLEAN/ARITHMETIC OPERATION</th>
<th>COND. CODE REG.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare Index Reg</td>
<td>CPX</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>AC 6 2 BC 5 3</td>
<td></td>
</tr>
<tr>
<td>Decrement Index Reg</td>
<td>DEX</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>09 4' 1</td>
<td></td>
</tr>
<tr>
<td>Decrement Stack Pnt</td>
<td>DES</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>38 4 1</td>
<td></td>
</tr>
<tr>
<td>Increment Index Reg</td>
<td>INX</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>08 4 1</td>
<td></td>
</tr>
<tr>
<td>Increment Stack Pnt</td>
<td>INS</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>31 4 1</td>
<td></td>
</tr>
<tr>
<td>Load Index Reg</td>
<td>LDX</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>EE 6 2 FE 5 3</td>
<td></td>
</tr>
<tr>
<td>Load Stack Pnt</td>
<td>LDS</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>EF 7 2 FF 6 3</td>
<td></td>
</tr>
<tr>
<td>Store Index Reg</td>
<td>STX</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>OF 5 2 EF 7 2 FF 6 3</td>
<td></td>
</tr>
<tr>
<td>Store Stack Pnt</td>
<td>STS</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>9F 5 2 AF 7 2 BF 6 3</td>
<td></td>
</tr>
<tr>
<td>Index Reg → Stack Pnt</td>
<td>TXS</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>35 4 1</td>
<td></td>
</tr>
<tr>
<td>Stack Pnt → Index Reg</td>
<td>TSX</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>30 4 1</td>
<td></td>
</tr>
</tbody>
</table>

**Booleans/Arithmetic Operation**
- \( X_H = M, X_L = (M + 1) \)
- \( X - 1 \rightarrow X \)
- \( SP - 1 \rightarrow SP \)
- \( X + 1 \rightarrow X \)
- \( SP + 1 \rightarrow SP \)
- \( M \rightarrow X_H, (M + 1) \rightarrow X_L \)
- \( M \rightarrow SP_H, (M + 1) \rightarrow SP_L \)
- \( X_H \rightarrow M, X_L \rightarrow (M + 1) \)
- \( SP_H \rightarrow M, SP_L \rightarrow (M + 1) \)

**Condition Code Reg.**
- \( H \) 1 1 1 1 1
- \( I \) 1 1 1 1 1
- \( Z \) 1 1 1 1 1
- \( V \) 1 1 1 1 1
- \( C \) 1 1 1 1 1

---

**FIGURE 3.2** (Reprinted from M6800 Reference & Data Sheets)
### JUMP AND BRANCH INSTRUCTIONS

<table>
<thead>
<tr>
<th>OPERATIONS</th>
<th>MNEMONIC</th>
<th>RELATIVE</th>
<th>INDEX</th>
<th>EXTEND</th>
<th>IMPLIED</th>
<th>BRANCH TEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Always</td>
<td>BRA</td>
<td>20</td>
<td>4</td>
<td>2</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>Branch If Carry Clear</td>
<td>BCC</td>
<td>24</td>
<td>4</td>
<td>2</td>
<td></td>
<td>C = 0</td>
</tr>
<tr>
<td>Branch If Carry Set</td>
<td>BCS</td>
<td>25</td>
<td>4</td>
<td>2</td>
<td></td>
<td>C = 1</td>
</tr>
<tr>
<td>Branch If Zero</td>
<td>BEQ</td>
<td>27</td>
<td>4</td>
<td>2</td>
<td></td>
<td>Z = 1</td>
</tr>
<tr>
<td>Branch If ≥ Zero</td>
<td>BGE</td>
<td>2C</td>
<td>4</td>
<td>2</td>
<td></td>
<td>N ⊕ V = 0</td>
</tr>
<tr>
<td>Branch If &gt; Zero</td>
<td>BGT</td>
<td>2E</td>
<td>4</td>
<td>2</td>
<td></td>
<td>Z + (N ⊕ V1) = 0</td>
</tr>
<tr>
<td>Branch If Higher</td>
<td>BHI</td>
<td>22</td>
<td>4</td>
<td>2</td>
<td></td>
<td>C + Z = 0</td>
</tr>
<tr>
<td>Branch If ≤ Zero</td>
<td>BLE</td>
<td>2F</td>
<td>4</td>
<td>2</td>
<td></td>
<td>Z + (N ⊕ V) = 1</td>
</tr>
<tr>
<td>Branch If Lower Or Same</td>
<td>BLS</td>
<td>23</td>
<td>4</td>
<td>2</td>
<td></td>
<td>C + Z = 1</td>
</tr>
<tr>
<td>Branch If ≤ Zero</td>
<td>BLT</td>
<td>2D</td>
<td>4</td>
<td>2</td>
<td></td>
<td>N ⊕ V = 1</td>
</tr>
<tr>
<td>Branch If Minus</td>
<td>BMI</td>
<td>28</td>
<td>4</td>
<td>2</td>
<td></td>
<td>N = 1</td>
</tr>
<tr>
<td>Branch If Not Equal Zero</td>
<td>BNE</td>
<td>26</td>
<td>4</td>
<td>2</td>
<td></td>
<td>Z = 0</td>
</tr>
<tr>
<td>Branch If Overflow Clear</td>
<td>BVC</td>
<td>28</td>
<td>4</td>
<td>2</td>
<td></td>
<td>V = 0</td>
</tr>
<tr>
<td>Branch If Overflow Set</td>
<td>BVS</td>
<td>29</td>
<td>4</td>
<td>2</td>
<td></td>
<td>V = 1</td>
</tr>
<tr>
<td>Branch If Plus</td>
<td>BPL</td>
<td>2A</td>
<td>4</td>
<td>2</td>
<td></td>
<td>N = 0</td>
</tr>
<tr>
<td>Branch To Subroutine</td>
<td>BSR</td>
<td>8D</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>JMP</td>
<td>6E</td>
<td>4</td>
<td>2</td>
<td>7E</td>
<td>3</td>
</tr>
<tr>
<td>Jump To Subroutine</td>
<td>JSR</td>
<td>AD</td>
<td>8</td>
<td>2</td>
<td>8D</td>
<td>9</td>
</tr>
<tr>
<td>No Operation</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return From Interrupt</td>
<td>RTI</td>
<td>3B</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return From Subroutine</td>
<td>RTS</td>
<td>39</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Interrupt</td>
<td>SWI</td>
<td>3F</td>
<td>12</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait for Interrupt</td>
<td>WAI</td>
<td>3E</td>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### COND. CODE REG.

<table>
<thead>
<tr>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>I</td>
<td>N</td>
<td>Z</td>
<td>V</td>
<td>C</td>
</tr>
</tbody>
</table>

- See Special Operations
- Advances Prog. Cntr. Only

---

**FIGURE 3.3** (Reprinted from Reference & Data Sheets)
### CONDITION CODE REGISTER MANIPULATION INSTRUCTIONS

<table>
<thead>
<tr>
<th>OPERATIONS</th>
<th>MNEMONIC</th>
<th>IMPLIED</th>
<th>OP</th>
<th>~</th>
<th>#</th>
<th>BOOLEAN OPERATION</th>
<th>COND. CODE REG.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Carry</td>
<td>CLC</td>
<td>0C</td>
<td>2</td>
<td>1</td>
<td></td>
<td>0 → C</td>
<td></td>
</tr>
<tr>
<td>Clear Interrupt Mask</td>
<td>CLI</td>
<td>0E</td>
<td>2</td>
<td>1</td>
<td></td>
<td>0 → I</td>
<td></td>
</tr>
<tr>
<td>Clear Overflow</td>
<td>CLV</td>
<td>0A</td>
<td>2</td>
<td>1</td>
<td></td>
<td>0 → V</td>
<td></td>
</tr>
<tr>
<td>Set Carry</td>
<td>SEC</td>
<td>00</td>
<td>2</td>
<td>1</td>
<td></td>
<td>1 → C</td>
<td></td>
</tr>
<tr>
<td>Set Interrupt Mask</td>
<td>SEI</td>
<td>0F</td>
<td>2</td>
<td>1</td>
<td></td>
<td>1 → I</td>
<td></td>
</tr>
<tr>
<td>Set Overflow</td>
<td>SEV</td>
<td>0B</td>
<td>2</td>
<td>1</td>
<td></td>
<td>1 → V</td>
<td></td>
</tr>
<tr>
<td>Acclmr A → CCR</td>
<td>TAP</td>
<td>06</td>
<td>2</td>
<td>1</td>
<td></td>
<td>A → CCR</td>
<td></td>
</tr>
<tr>
<td>CCR → Acclmr A</td>
<td>TPA</td>
<td>07</td>
<td>2</td>
<td>1</td>
<td></td>
<td>CCR → A</td>
<td></td>
</tr>
</tbody>
</table>

### CONDITION CODE REGISTER NOTES:

1. (Bit V) Test: Result = 10000000?
2. (Bit C) Test: Result = 00000000?
3. (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)
4. (Bit V) Test: Operand = 10000000 prior to execution?
5. (Bit V) Test: Operand = 01111111 prior to execution?
6. (Bit V) Test: Set equal to result of N©C after shift has occurred
7. (Bit N) Test: Sign bit of most significant (MS) byte = 1?
8. (Bit V) Test: 2’s complement overflow from subtraction of MS bytes?
9. (Bit N) Test: Result less than zero? (Bit 15 = 1)
10. (All) Load Condition Code Register from Stack. (See Special Operations)
11. (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.
12. (All) Set according to the contents of Accumulator A.

FIGURE 3.4 (Reprinted from M6800 Reference & Data Sheets)
### SPECIAL OPERATIONS

#### JSR, JUMP TO SUBROUTINE:

<table>
<thead>
<tr>
<th>PC Main Program</th>
<th>SP Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>n AD = JSR</td>
<td>SP-2</td>
</tr>
<tr>
<td>n+1 K = Offset*</td>
<td>SP-1</td>
</tr>
<tr>
<td>n+2 Next Main Instr.</td>
<td>SP [n+2] H</td>
</tr>
</tbody>
</table>

*K = 8-Bit Unsigned Value

<table>
<thead>
<tr>
<th>PC Main Program</th>
<th>SP Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>n BD = JSR</td>
<td>SP-2</td>
</tr>
<tr>
<td>n+1 SH = Subr. Addr.</td>
<td>SP [n+3] H</td>
</tr>
<tr>
<td>n+2 SL = Subr. Addr.</td>
<td>SP [n+3] L</td>
</tr>
</tbody>
</table>

= Stack Pointer After Execution.

#### BSR, BRANCH TO SUBROUTINE:

<table>
<thead>
<tr>
<th>PC Main Program</th>
<th>SP Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>n BD = BSR</td>
<td>SP-2</td>
</tr>
<tr>
<td>n+1 ± K = Offset*</td>
<td>SP-1</td>
</tr>
<tr>
<td>n+2 Next Main Instr.</td>
<td>SP [n+2] H</td>
</tr>
</tbody>
</table>

*K = 7-Bit Signed Value;

n+2 Formed From [n+2] H and [n+2] L

#### JMP, JUMP:

<table>
<thead>
<tr>
<th>PC Main Program</th>
<th>SP Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>n 8E = JMP</td>
<td>SP-2</td>
</tr>
<tr>
<td>n+1 K = Offset*</td>
<td>SP-1</td>
</tr>
<tr>
<td>n+2 Next Instruction</td>
<td>SP [n+2] H</td>
</tr>
</tbody>
</table>

#### RTS, RETURN FROM SUBROUTINE:

<table>
<thead>
<tr>
<th>PC Subroutine</th>
<th>SP Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>S 39 = RTS</td>
<td>SP+1</td>
</tr>
<tr>
<td></td>
<td>SP+2 N H</td>
</tr>
</tbody>
</table>

#### RTI, RETURN FROM INTERRUPT:

<table>
<thead>
<tr>
<th>PC Interrupt Program</th>
<th>SP Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>S 38 = RTI</td>
<td>SP+1</td>
</tr>
<tr>
<td></td>
<td>SP+2 Acmlt B</td>
</tr>
<tr>
<td></td>
<td>SP+3 Acmlt A</td>
</tr>
<tr>
<td></td>
<td>SP+4 Index Register (XH)</td>
</tr>
<tr>
<td></td>
<td>SP+5 Index Register (XL)</td>
</tr>
<tr>
<td></td>
<td>SP+6 N H</td>
</tr>
<tr>
<td></td>
<td>SP+7 N L</td>
</tr>
</tbody>
</table>

**FIGURE 4 (Reprinted from Reference and Data Sheets)**
has no occurrence of such a register, since the A and B accumulators are addressed solely on the basis of the opcode, and not on any extra operand field. In fact, in the accumulator mode of addressing, there are no operand fields: the one-byte opcode is sufficient. A similar case exists for those in the implied mode of addressing.

Our 9 Special Registers are assigned to SIM/GEN as follows:

SREG0 - accumulator A.
SREG1 - accumulator B.
SREG2 - Index Register.
SREG3 - Half-carry bit of the condition code register.
SREG4 - Interrupt bit of the condition code register.
SREG5 - Negative bit of the condition code register.
SREG6 - Zero bit of the condition code register.
SREG7 - Overflow bit of the condition code register.
SREG8 - Carry bit of the condition code register.

The Interrupt bit is accommodated for the sake of completeness, although we shall have no occasion to really use it for simulation. It is obvious that the condition code register was chosen not to be represented to SIM/GEN as a stand-alone register. This would considerably ease the instruction definitions for the XECUTE module when these bits of the condition code register would have to be set or cleared depending on the instruction. If maintained as a single Special Register to SIM/GEN, the user will face the extra burden of having to "shift-and-mask" each time a particular bit is being
considered.

The Program Counter and the Stack Pointer of the M6800 have not been declared as Special Registers since SIM/GEN has a set of operands which already include them. The names PC and STACKP have been given to registers with similar functions (see page 65).

The third data card indicates bit width of the M6800: 8 bits. The two memory segments so declared are considered the minimum for a M6800 configuration. We chose the first 128 memory locations to be a RAM type of memory, and the next 1024 to be ROM. Hence the first segment will have addresses in the range from 0 to 127 (decimal), while the next segment are in the range from 128 to 1151 (decimal). The equivalent form in hexadecimal SIM/GEN notation are shown on the third card.

The fourth data card simply indicates that our stack for the M6800 is memory-based. The fifth data card is our machine name. The sixth card indicates the number of memory words that must be fetched for every execution cycle to uniquely determine the instruction opcode. In the M6800, this is exactly 1 memory word. This completes the description of the required user input for the MEMORY module. An actual generation run for the above data set is shown on Figure 5.

3.5 DECODE module for the M6800

Pages 8-15 of the reference manual describe the user required input to the DECODE module. Our input to the DECODE module is shown
**SIM/GEN: VERSION 5.3**

**MEMORY MODULE**

**GENERATOR RUN**

---

**MICROPROCESSOR TITLE:** MOTOROLA 6800

**NO GENERAL REGISTERS DECLARED**

**SPECIAL REGISTER SPECIFICATIONS...NUMBER OF AVAILABLE REGISTERS:** 9

**MEMORY-BASED STACK FACILITY DECLARED**

**NUMBER OF MEMORY WORDS LOADED INTO THE INSTRUCTION REGISTER DURING EACH FETCH PHASE:** 1

**MEMORY BIT WIDTH:** 8

****MEMORY SEGMENT DECLARATIONS ARE GIVEN BELOW****

  0 TO 7F [READ-WRITE]

  80 TO 47F [READ-ONLY]

---

**FIGURE 5 (MEMORY generation run)**
on Appendix B. For uniformity, we have retained the same scheme used for specifying the translation classes in using ASM/GEN earlier in this paper. There are 7 execution classes:

Class 1: All the instructions in the accumulator and implied mode of addressing have been grouped under this class. These are simply the 1-byte instructions. There are 51 of them.

Class 2: All the instructions in the relative mode of addressing fall under this execution class. These are all the branch instructions in the instruction set (note: JSR, jump to subroutine, and JMP, unconditional jump, are not in relative mode. There is, however, nothing in SIM/GEN to prevent us from including, for instance, the JMP instruction in indexed mode in Class 2.). There are 16 such instructions.

Class 3: All instructions in the direct mode of addressing are grouped in this class. There are 27 instructions in this class.

Class 4: All 2-byte immediate mode instructions. There are 20 instructions in this execution class. Note, that the 3-byte immediate mode of addressing instructions cannot be members of this class.

Class 5: All 3-byte immediate mode instructions. There are only 3 instructions in this execution class: LDX (load IX with a 16-bit immediate operand field); CPX (compare IX with a 16-bit immediate operand field); and LDS (load Stack Pointer with a 16-bit immediate operand field).

Class 6: All indexed mode of addressing instructions. There are
40 instructions in this class.

Class 7: All extended mode of addressing instructions. There are also 40 instructions in this class and they parallel the Class 6 instructions. That is, every instruction that admits the indexed mode of addressing also admits the extended mode of addressing, and conversely.

An actual generation listing for the above classification, using the data set found in Appendix B, is shown in Figure 6 below.

3.6 Class 2 and the XECUTE module

The instructions of this execution class admit the relative mode of addressing. All leave the condition code register unaffected. Each is of the form

\textbf{IF} condition \textbf{THEN} branch to effective address,

and may be easily described to SIM/GEN using the IF statement of IDL. Each testable condition involves a check of one or more of the status bits in the condition code register. If more than 1 status bit is involved, a calculation must first be performed and the result of the calculation held in some temporary IDL operand TEMP\text{ip} (i from 1 to 8, inclusive). The check is then performed against the temporary operand holding the result of calculation. The complete list of the testable conditions for each instruction is found in Figure 3.3 under the column labelled "Branch Test". The complete set of inputs to SIM/GEN for this class is shown on the following pages.

Since any instruction requires the calculation of an effective
**SIMGENI VERSION 5.3**

**DECODE MODULE**

**GENERATOR RUN**

---

**GENERATION RUN SUMMARY**

---

There are 7 execution classes modules.

The opcode bit width is: 8

The legal opcodes are mapped to their execution class below.

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x01</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x02</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x03</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x04</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x05</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x06</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x07</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x08</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x09</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x0A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x0B</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x0C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x0D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x0E</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x0F</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x12</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x13</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x14</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x15</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x16</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x17</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x18</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x19</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x1A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x1B</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x1C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x1D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x1E</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x1F</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

*Figure 6 (DECODE generation run)*
CLS2 2
CLS2 3
CLS2 4
CLS2 5
CLS2 6
CLS2 7
CLS2 8
CLS2 9
CLS2 10
CLS2 11
CLS2 12
CLS2 13
CLS2 14
CLS2 15
CLS2 16
CLS2 17
CLS2 18
CLS2 19
CLS2 20
CLS2 21
CLS2 22
CLS2 23
CLS2 24
CLS2 25
CLS2 26
CLS2 27
CLS2 28
CLS2 29
CLS2 30
CLS2 31
CLS2 32
CLS2 33
CLS2 34
CLS2 35
CLS2 36
CLS2 37
CLS2 38
CLS2 39
CLS2 40
CLS2 41
CLS2 42
CLS2 43
CLS2 44
CLS2 45
CLS2 46
CLS2 47
CLS2 48
CLS2 49
CLS2 50
CLS2 51
CLS2 52
CLS2 53
CLS2 54
CLS2 55
CLS2 56
CLS2 57
CLS2 58

2 16 1
8 0 9

BRA - BRANCH ALWAYS
20 4.0
CALL BRNHTO IMM1 PC.
ENDINSTR

BCC - BRANCH IF CARRY CLEAR
24 4.0
IF SREG6 EQ 0
CALL BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BCS - BRANCH IF CARRY SET
25 4.0
IF SREG6 EQ 1
CALL BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BEQ - BRANCH IF = 0
27 4.0
IF SREG6 EQ 1
CALL BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BGE - BRANCH IF >= 0
2C 4.0
XOR TEMP2 SREG5 SREG7
IF TEMP2 EQ 0
CALL BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BGT - BRANCH IF > 0
2E 4.0
XOR TEMP2 SREG5 SREG7
OR TEMP2 TEMP2 SREG6
IF TEMP2 EQ 0
CALL BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BMI - BRANCH IF HIGHER
22 4.0
OR TEMP2 SREG6 SREG6
IF TEMP2 EQ 0
CALL BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BLE - BRANCH IF < 0
2F 4.0
XOR TEMP2 SREG5 SREG7
OR TEMP2 TEMP2 SREG6
IF TEMP2 EQ 1
CALL BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BLS - BRANCH IF LOWER OR SAME
```assembly
23  4,0  OR     TEMP2 SREG8 SREG6
    IF     TEMP2 EQ 1
    CALL  BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BLT - BRANCH IF < 0
2D  4,0  XOR    TEMP2 SREG5 SREG7
    IF     TEMP2 EQ 1
    CALL  BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BMI - BRANCH IF MINUS
28  4,0  IF     SREG5 EQ 0
    CALL  BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BNE - BRANCH IF NOT = 0
26  4,0  IF     SREG6 EQ 1
    CALL  BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BVC - BRANCH IF OVERFLOW CLEAR
2B  4,0  IF     SREG7 EQ 0
    CALL  BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BVS - BRANCH IF OVERFLOW SET
29  4,0  IF     SREG7 EQ 1
    CALL  BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BPL - BRANCH IF PLUS
24  4,0  IF     SREG5 EQ 1
    CALL  BRNHTO IMM1 PC.
ENDIF
ENDINSTR

BSR - BRANCH TO SUBROUTINE
3D  8,0  ADD    TEMP2 PC 2
    ADD    TEMP3 TEMP2 FF16
    ADD    TEMP4 TEMP2 FF0016
    SHR   TEMP4 TEMP4 8
    PUSH  TEMP3
    PUSH  TEMP4
    CALL  BRNHTO IMM1 PC.
ENDINSTR

DEFINE BRNHTO TEMP1 TEMP2.
MOVE  TEMP3 TEMP1
AND   TEMP4 TEMP3 6016
IF    TEMP4 NE 0
```
CONCAT TEMP3 FF+16 (8) TEMP3
ENDIF
ADD TEMP2 TEMP2 TEMP3
AND TEMP2 TEMP2 FFFF+16
RETURN
ENDINSTR
ENDCLASS
address if its testable condition is true, this common task may be factored out using the subroutine definition capability of IDL. Two operands are necessary to this task of calculating the effective address: the program counter (PC), and the first operand field of the instruction (known to SIM/GEN as IMM1). These are passed as actual parameters to the subroutine BRNHTO (CLS2.113-123), since the only operands allowed in the body of a user-defined subroutine in IDL are the temporary IDL operands TEMPl, TEMP2 ... TEMP8 (page 53). The CALL statement in IDL is then used to invoke the subroutine, with the actual parameters replacing the formal parameters in the usual understanding of a FORTRAN CALL (by reference).

The BRNHTO subroutine simulates the calculation of the effective address for the relative mode of addressing of the M6800 microprocessor: the second byte of the instruction treated as a 7-bit signed value (TEMPl) is added to the 16-bit program counter (TEMP2), plus 2. Upon entry, a check is first made to find out if the 7-bit signed offset is negative. If so, the equivalent negative number in 16-bit format is generated using the CONCAT operation (page 52) of IDL. The result is held in TEMP3 and then added to the contents of the program counter (CLS2.119). The AND operation before the RETURN ensures that only the low 16 bits are kept on the host word. If this is not done, an incorrect PC value is very likely to result. Consider the case when PC = 0080 and IMM1 = FB, both expressed in hexadecimal. Since IMM1 is a 7-bit signed value (-5 in 2's complement), it becomes FFFB,
which is the 16-bit signed value equivalent. When the addition is carried out on the host word (of 60-bits in our case), the sum is 1007B and is not correct! Only by masking out the low 16 bits is the correct answer of 7B obtained.

A second point to be made here concerns the question: why is 2 not added to the PC? This has to do with the simulator that SIM/GEN produces. It is natural to assume that during program execution (not to be confused with program simulation) the instruction currently executed has its address in the PC. In the case of this 2-byte relative mode of addressing instruction of the M6800, it is quite correct to make the analogous assumption that the program counter contains the address of the opcode (the first byte of the 2-byte instruction). This is not the case with the simulator produced by SIM/GEN. Upon entry to the code proper to simulate an instruction, the value of the PC is already 1 more than the address of the last memory word comprising the simulated instruction. That is, PC has been advanced to point to the next instruction. Adding 2 to the PC in this instance would result in an incorrect simulation for this class of instructions.

To summarize, we note the following:

(1) CALLing a subroutine DEFINed in IDL assumes that binding with the actual parameters from the calling statement is ordered as in FORTRAN (call by reference). Care must be taken, therefore, not to involve the parameters in destructive-type IDL operations (hence the MOVE statement at CLS2.114).
The bit-width of the host-word is particularly important in arithmetic operations. The simulated operands have bit-widths of their own and the two must not be confused. Throughout, in this tutorial, our host machine has a 60-bit word.

At the instruction emulation level, the program counter already contains the address of the next instruction in sequence. This is characteristic of the generated simulator.

The M6800 microprocessor, our target machine in this tutorial, uses 2's complement arithmetic. But our host machine, the CDC-6400, is a 1's complement machine. The details of this will be taken up in a later section.

3.7 The condition code subroutines

This section will deal with the condition code settings that are affected by certain instructions as noted in Figures 3.1-3.4. The mode of addressing has nothing to do with the setting or clearing of a particular status bit of the condition code register. So these routines will be needed according to instruction, not addressing mode. Our classification scheme is by addressing mode and tends to obscure this commonality somewhat. So this section will present the subroutines that are CALLed from more than 1 execution class. We start with the following table, condensed from Figures 3.1, 3.2, 3.4:
<table>
<thead>
<tr>
<th>SREGn</th>
<th>Addition</th>
<th>Subtraction</th>
<th>Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half-Carry (3)</td>
<td>$P_3Q_3 + P_3R_3 + R_3Q_3$</td>
<td>unaffected</td>
<td>unaffected</td>
</tr>
<tr>
<td>Negative (5)</td>
<td>$R_7$</td>
<td>$R_7$</td>
<td>$R_7$</td>
</tr>
<tr>
<td>Zero (6)</td>
<td>all bits = 0</td>
<td>all bits = 0</td>
<td>all bits = 0</td>
</tr>
<tr>
<td>Overflow (7)</td>
<td>$P_7Q_7R_7 + P_7Q_7R_7$</td>
<td>$P_7Q_7R_7 + P_7Q_7R_7$</td>
<td>N 0 C</td>
</tr>
<tr>
<td>Carry (8)</td>
<td>$P_7Q_7 + P_7R_7 + R_7Q_7$</td>
<td>$P_7Q_7 + Q_7R_7 + R_7F_7$</td>
<td>$P_0 / P_7$</td>
</tr>
</tbody>
</table>

For the arithmetic-type operations, it is assumed that $R = P + Q$ (addition), or $R = P - Q$ (subtraction). The subscripts denote the bit positions of these 8-bit operands. In the usual understanding, $\overline{P_7}$ implies the complement operation on bit 7 (using a right-to-left numbering with 0 as the rightmost) of the operand $P$; $PQ$ implies the logical and operation of $P$ and $Q$, $P + Q$ the logical or operation, and $P \oplus Q$ the exclusive or operation. The Carry bit setting for the Shift (all Rotate, Shift arithmetic or logical instructions) depends on the initial operand's left-most or right-most bit, the former if a left shift operation is involved, the latter is a right shift.

The following table contains the subroutine names defined to implement each of the necessary status bit settings:

<table>
<thead>
<tr>
<th>Addition</th>
<th>Subtraction</th>
<th>Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARYHA</td>
<td>SIGN</td>
<td>SIGN</td>
</tr>
<tr>
<td>SIGN</td>
<td>ZERO</td>
<td>ZERO</td>
</tr>
<tr>
<td>ZERO</td>
<td>OVERFA</td>
<td>OVERFS</td>
</tr>
<tr>
<td>OVERFA</td>
<td>CARYSA</td>
<td>NZVCB</td>
</tr>
<tr>
<td>CARYSA</td>
<td>NZVCB</td>
<td></td>
</tr>
</tbody>
</table>

The IDL statements for these routines are contained between CLS1.318-423. In the last column, the SIGN and ZERO routines are
actually embedded in the NZVB subroutine (CLS1.318-328).

SIGN and ZERO (CLS1.343-356) are both self-explanatory. Each is defined with 2 parameters, the first parameter being the operand whose sign bit is to be checked for a '1' (for SIGN), or, for which a zero check on all bits is wanted (for ZERO). The second parameter will always be SREG5 for SIGN and SREG6 for ZERO for all CALL's.

The CARYHA, OVERFA, OVERFS, CARYSA and CARYSS subroutines are implementations of the Boolean formulas corresponding to the first table. As is evident from the table, 3 operands are necessary to establish the correct setting for the status bits involved. The first 3 parameters of these 5 subroutines serve this purpose and will always correspond respectively to R, P, and Q in the table. Recall that we are considering $R = P + Q$ (addition) and $R = P - Q$ (subtraction). The fourth parameter will always be SREG3 for CARYHA, SREG7 for OVERFA and OVERFS, and SREG8 for CARYSA and CARYSS. These assumptions are necessary to build up other subroutines that combine the ones already defined. HNZVCA (CLS1.387-394) and NZVCS (CLS1.465-471) are examples of this form of build-up: this way, the condition code setting for a large number of instructions may be invoked by a single CALL statement at the instruction definition level without digressing from the IDL statement or statements for the instruction itself.

A difficulty involved in reading the subroutine definitions in IDL arises from the restriction that the only operands (apart from...
constants) appearing in subroutine definitions are the IDL temporary registers TEMP1 ... TEMP8. But even at the instruction definition level, where more helpful mnemonics have been provided for such operands as PC, STACKP, IMMi, there is still a considerable degree of difficulty in remembering, say, what SREGO stands for. For purposes of the subroutines involved in this section, we urge the substitution of TEMP1, TEMP2 and TEMP3 everywhere for R, P, and Q respectively and referring back to the first table presented. TEMP4 will depend on the particular routine as earlier noted. Thus:

\[
\begin{array}{c}
\text{CARYHA} & R & P & Q & H \\
\text{CARYSA} & R & P & Q & C \\
\text{CARYSS} & R & P & Q & C \\
\text{OVERFA} & R & P & Q & V \\
\text{OVERFS} & R & P & Q & V \\
\end{array}
\]

where \( R = P + Q \) or \( R = P - Q \) (\( R \) = Result, \( P \) = first operand, \( Q \) = second operand). At the next "higher" level, we would have:

\[
\begin{array}{c}
\text{HNZVCA} & R & P & Q & H & N & Z & V & C \\
\text{NZVCS} & R & P & Q & N & Z & V & C \\
\text{NZVCB} & R & P & N & Z & V & C & \text{mask}.
\end{array}
\]

In the above, HNZVCA will handle the \( R = P + Q \) (addition) instructions. NZVCS will handle the \( R = P - Q \) (subtract) instructions. NZVCB involves only \( R \) and \( P \) since this is the group of "shift" (CLS1.120-187) operations and so has a single operand. The last parameter, denoted by "mask", is either a mask for bit 7 (if shift
direction was left-ward) or for bit 0 (if shift direction was right-
ward). The mask is applied on P, the operand.

A last subroutine is the NZR subroutine defined at CLS3.195-200.
It is called at a large number of instruction definitions and involves
checking the Negative and Zero status bits, and resetting the Overflow
bit to 0. It is invoked in some cases where the Negative and Zero
status bits are to be checked as usual, but where the Overflow bit is
set or cleared in a different manner. In this latter case, the third
parameter is a dummy TEMP7. The reader will observe numerous occur-
rences of

\begin{verbatim}
CALL NZR result SREG5  SREG6  SREG7, or
CALL NZR result SREG5  SREG6  TEMP7,
\end{verbatim}

usually with load/store accumulators instructions, accumulator and
memory word operation instructions, test or transfer accumulator/
memory instructions (see Figure 3.1 right-hand column).

This completes the condition code setting subroutines.

3.8 Two's complement arithmetic

In this section, the subroutine to perform 2's complement arith-
metic on the 1's complement host machine will be defined. Three
parameters, corresponding to the calculated difference, the minuend
and the subtrahend are necessary for this purpose. The IDL subroutine
will have the form

\begin{verbatim}
DEFINE SUBTR TEMP1  TEMP2  TEMP3.
\end{verbatim}

where it is assumed that the operation
\[ \text{TEMP1} = \text{TEMP2} - \text{TEMP3} \]
is to be carried out. In the ensuing discussion, the operands are assumed to be 8 bit operands for the purpose of simulating the M6800. But the fact that they will be simulated on the 60-bit host word must not be overlooked.

Consider first the following IDL statements to carry out the desired subtraction:

\[
\begin{align*}
\text{CLEAR} & \quad \text{TEMP1} & \quad \text{EQ} & \quad \text{TEMP3} & \quad \text{RETURN} \\
\text{IF} & \quad \text{TEMP2} & \quad \text{ENDIF} \\
\text{COMTWO} & \quad \text{TEMP1} & \quad \text{TEMP3} & \quad \text{ADD} & \quad \text{TEMP1} & \quad \text{TEMP2} & \quad \text{TEMP1} & \quad \text{AND} & \quad \text{TEMP1} & \quad \text{TEMP1} & \quad 255 & \quad \text{RETURN} \\
\end{align*}
\]

- zero difference
- if minuend = subtrahend
- complement subtrahend
- get difference
- done

If the minuend is equal to the subtrahend, the answer is obvious. Otherwise, in the usual understanding of machine subtraction, we must add to the minuend the negative representation of the subtrahend. This is achieved by the COMTWO operation in IDL. To insure that we are working only with the low 8 bits of these 60-bit operands, a mask (= FF hexadecimal) is performed on the result to clear any garbage beyond the 8th bit. As earlier noted in section 3.6, incorrect operands may subsequently appear if this "safeguard" is not observed. The ADD operation generates the desired difference and is followed by the same safeguard. The importance of this safeguard cannot be overemphasized: consider the simple case when SUBTR is called with the following actual parameters
Upon entry, the higher-ordered bits beyond the 8th bit in these 60-bit host words are clear. The COMTWO operation, which translates into the FORTRAN statement

\[
\text{TEMP1} = -\text{TEMP3} + 1
\]

will execute as follows

\[
\begin{align*}
(\text{TEMP3}) &= 0 \ldots 0 \ 0000 \ 0011 & \text{. initially} \\
&= 1 \ldots 1 \ 1111 \ 1100 & \text{. after } -\text{TEMP3} \\
(\text{TEMP1}) &= 1 \ldots 1 \ 1111 \ 1101 & \text{. after } -\text{TEMP3} + 1
\end{align*}
\]

Without the mask of the low 8 bits, the situation just before the ADD looks like

\[
\begin{align*}
(\text{TEMP2}) &= 0 \ldots 0 \ 1111 \ 1111 \\
(\text{TEMP1}) &= 1 \ldots 1 \ 1111 \ 1101.
\end{align*}
\]

When the ADD is executed, a carry is propagated into and out of the highest ordered bit (60th bit), causing an "end around" carry that is automatically done by the host machine, which results in:

\[
(\text{TEMP1}) = 0 \ldots 0 \ 1111 \ 1101. (-3)
\]

This is an incorrect result, if it is to be interpreted as an 8-bit signed value for the M6800 machine, which in 2's complement representation is -3 (decimal). The point here is clear: at any stage during the simulation of an n-bit operand, the higher ordered (60-n) bits of the host word must always be kept clear (i.e., zeroes).

Two other checks have to be done separately. These stem from the fact that in the 1's complement machine, the number 0 has 2 different representations; all ones or zeroes. An extra effort must be
made to get around this. When, for instance, SUBTR is invoked with 
(TEMP3) = 0 (all zeroes), the COMTWO operation executed in the con-
text of the host machine generates a positive 1, which then gets 
added to the minuend. Subtracting zero will increment the minuend. 
If now (TEMP3) = 1, COMTWO will result in a string of ones which is 
zero to the machine. In this case, subtracting 1 leaves the minuend 
unaffected. As it turns out, the subsequent mask should correct the 
situation, but the compiler involved had "avoid negative zero" code 
in the first place, so that at the IDL level of simulation, we must 
treat this as a special case.

The complete text for SUBTR is shown on lines CLS1.431-449. An 
equivalent for 16 bit operands which will be necessary for some M6800 
instructions, is shown on lines CLS3.232-250.

3.9 EXECUTE module for the M6800

Referring to Figures 3.1 and 3.2, we observe that many instruc-
tions can be emulated by a single IDL operation. For those with 
more than 1 mode of addressing, only the operand fetch step is dif-
ferent. The rest is completely identical. This property is most 
evidently brought out through the LDAA instruction, (load accumula-
tor A). The 4 sets of IDL statements for this instruction in the 
direct, immediate, indexed, and extended mode of addressing are as 
follows:
The immediate mode of addressing does not require a memory reference for the operand so the MOVE statement is not preceded by any CALL. GETOPN, GETXOP and GETEOP correspond to the direct, indexed and extended addressing mode routines to fetch the operand using the bus-organized assumption in SIM/GEN. The body of the subroutines should be self-explanatory: upon return the desired operand is in the data bus.

Every instruction with an immediate mode of addressing will also have the direct, indexed and extended modes of addressing. The exceptions to these are the store operations, STAA, STAB, STX, and STS. The indexed and extended modes (CLS6 and CLS7) are completely
identical following the CALL GETXOP or GETEOP, except for the JMP and JSR instructions (CLS6.305-320, CLS7.301-314). Note, that there are several instructions that deal with 16-bit operands, as in LDX, LDS, STX, STS (load, store/index register or stack pointer). The NZVLO, STOLOP and GETLOP (CLS3.201-231) are concerned with the status bit settings, storing & fetching of 16-bit operands.

The routines earlier defined concerning the status bits may now be CALLed where appropriate. In the special cases, the setting is localized, as for instance, in the CPX instruction (CLS5.6-17). The majority of the instructions CALL the ones already defined (NZR, HNZVCA, NZVCS, NZVCB). The details of the emulation code in IDL for all the instructions should be evident from the statements themselves. The remainder of this section is devoted to presenting the less obvious ones.

JSR - Jump to subroutine admits the indexed (CLS6.310-320) and extended (CLS7.305-314) modes of addressing. The operation is graphically illustrated on Figure 4. Note that in the indexed mode, the return address is 2 bytes away from the present instruction. In the extended, it is 3 bytes. Although this is to be noted, we already have the return address in the PC (program counter) upon entry to the emulation code for these instructions, because as already pointed out in section 3.6, the SIM/GEN simulator does it in the operand extract process. Adding 2 or 3 to PC would result in an incorrectly simulated return address. The return address is 16 bits, and must
be kept on the next 2 bytes of the stack. This is accomplished by the 2 AND operations and the SHRL (to right justify the high order 8 bits) operation, followed by the PUSH. Note that the low byte must be PUSHed first.

The only other instruction that requires a detailed explanation is the DAA instruction (Decimal Adjust Accumulator A). In the 8-4-2-1 BCD (Binary Coded Decimal) representation, each decimal digit is represented by its equivalent 4-bit code in binary. That is:

- $0 = 0000$
- $1 = 0001$
- $2 = 0010$
- $3 = 0011$
- $4 = 0100$
- $5 = 0101$
- $6 = 0110$
- $7 = 0111$
- $8 = 1000$
- $9 = 1001$

The above convention is so widely used, people hardly qualify it with the 8-4-2-1 when they say BCD. There are actually 2 more types of BCD, called the 2-4-2-1 and the excess-3 BCD codes, but we shall not be concerned with these.

In the binary addition of decimally interpreted operands, (that is, operands of the form $0 = 0_1 \ 0_2$, where $0$ is 8 bits, $0_1$ is the high-order 4 bits and $0_2$ low order 4 bits, with $0_1$ and $0_2$ having only values from 0 to 9) the following 3 situations may arise (consider 4 bit-operands for the moment):

(1) The sum $S$, of the 2 digits is such that $S$ is in the range $(0000, 1001)$ inclusive, so the result is correct;
(2) S is in the range (1010, 1111), in which case the result is a nonvalid BCD combination, so a correction is required;

(3) S is in the range (10000, 11001) in which case a carry is generated and the result - as read in the assumed BCD code - is incorrect.

A correction is required in some cases and none in others. The solution is to add 6 decimal to the result when a correction effort is required. Consider:

(1) \[
\begin{array}{c}
5_{10} = 0101 \\
3_{10} = 0011 \\
\text{sum} = 1000 = 8_{10} \quad \text{(correct result)}
\end{array}
\]

(2) \[
\begin{array}{c}
6_{10} = 0110 \\
8_{10} = 1000 \\
\text{(sum=)} \ 1110 \quad \text{(nonvalid BCD number)} \\
\ 0110 \quad \text{(add } 6_{10}) \\
\text{sum= 1 0100 = 14}_{10} \quad \text{(correct BCD result)}
\end{array}
\]

(3) \[
\begin{array}{c}
9_{10} = 1001 \\
8_{10} = 1000 \\
\text{(sum=)} \ 10001 \quad \text{(a carry is generated and the result is incorrect)} \\
\ 0110 \quad \text{(add } 6_{10}) \\
\text{sum= 1 0111 = 17}_{10} \quad \text{(correct BCD result)}
\end{array}
\]

The DECIML (CLS1.450-463) is defined with 3 parameters that have the following meanings:

TEMP1 = a 4 bit operand that is either the high 4 bits or low 4 bits of an 8-bit M6800 byte.

TEMP2 = either SREG3 or SREG8, depending on whether the low (SREG3) or high (SREG8) 4 bits are examined;

TEMP3 = returned as a flag to indicate that a carry was generated...
in the adjustment phase (i.e., in the addition of $6_{10}$);

To decimally adjust ACCA, (CLS1.63-75) we mask out the low and the high 4 bits into TEMP1 and TEMP2 respectively, and invoke DECIML to do the necessary correction. In adjusting the low 4 bits, DECIML is called with a second parameter of SREG3 - the Half-Carry flag of the condition code register. This is indeed important to the DECIML routine as there would be no way of finding out whether the previous add resulting in the current contents of ACCA resulted in a carry out of bit 3 (the 4th bit). The second call uses SREG8 - the Carry flag bit - for the same reason. Upon return from the first call, the result flag TEMP3, must be added to the high order 4 bits (TEMP2) first before further adjustment is to be performed.

Returning to the DECIML subroutine, we begin by clearing the return flag parameter (TEMP3) and check to see if any adjustment is required. That is, we ask if operand is greater than 9 or was a carry generated? If not, simply return. If either condition in question is satisfied, the flag is set and $6_{10}$ is added to TEMP1.

When both the high and low order 4 bits of ACCA have been adjusted, the CONCAT operation is used to establish the decimally adjusted ACCA. The Carry bit, if previously set, cannot be cleared by the DAA instruction. If previously cleared, it will depend on the adjustment of the high order 4.

In closing, observe the following:

1) The "safeguard" operation (section 3.6) of masking out the low 8 or 16 bits of any arithmetic operand result is followed
throughout all execution classes.

(2) The DECR operation of IDL is avoided for the same reasons as the extra check needed in SUBTR to treat a subtrahend of 1 separately (see CLS1.76-89). Instead, an add with a negative 1 in 2's complement form is performed.

(3) The status bit setting routines are effectively grouped for sets of instructions that set/clear these bits under identical conditions. Examples are NZR, NZVCS, HNZVCA, NZVCB.

(4) The program counter will have as its contents the location of the next available instruction upon entry to the emulation code for an instruction. Instructions affecting the PC (Class 2, JSR) should take this into account.

(5) The bus organized assumption is observed at all phases of simulation. This includes the instruction fetch and operand fetch routines generated at the DECODE module generator. This also explains the state of the PC as noted in (4).
REFERENCES


(3) Designing With Microprocessors, IEEE Compcon Fall 76, Catalog no. 76CH1178-3C.


APPENDIX A: ASM/GEN input for the M6800 microprocessor
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>8</th>
<th>8</th>
<th></th>
<th></th>
<th>ANDBI C4</th>
<th></th>
<th>CPXX AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 8</td>
<td>1 1</td>
<td>1 1</td>
<td>20</td>
<td>BIA</td>
<td>65</td>
<td>BIA</td>
<td>AE</td>
<td></td>
</tr>
<tr>
<td>8 8</td>
<td>5F</td>
<td>0 0</td>
<td>25</td>
<td>BCS</td>
<td>1</td>
<td>BCS</td>
<td>AF</td>
<td></td>
</tr>
<tr>
<td>A8</td>
<td>18</td>
<td>22</td>
<td>BHI</td>
<td>68</td>
<td>BHI</td>
<td>8E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CGA</td>
<td>11</td>
<td>26</td>
<td>BLE</td>
<td>8A</td>
<td>BLE</td>
<td>0E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMA</td>
<td>43</td>
<td>29</td>
<td>5</td>
<td>END</td>
<td>7</td>
<td>END</td>
<td>9E</td>
<td></td>
</tr>
<tr>
<td>COMB</td>
<td>53</td>
<td>2A</td>
<td>16</td>
<td>END</td>
<td>7</td>
<td>END</td>
<td>BB</td>
<td></td>
</tr>
<tr>
<td>DECA</td>
<td>44</td>
<td>2D</td>
<td>5C</td>
<td>END</td>
<td>8</td>
<td>END</td>
<td>F3</td>
<td></td>
</tr>
<tr>
<td>DECB</td>
<td>5A</td>
<td>2E</td>
<td>5D</td>
<td>END</td>
<td>9</td>
<td>END</td>
<td>F4</td>
<td></td>
</tr>
<tr>
<td>INCA</td>
<td>4C</td>
<td>3</td>
<td>2</td>
<td>CPX</td>
<td>8C</td>
<td>CPLX</td>
<td>7F</td>
<td></td>
</tr>
<tr>
<td>INCB</td>
<td>5C</td>
<td>4</td>
<td>6</td>
<td>CMPA</td>
<td>8E</td>
<td>CMPA</td>
<td>9E</td>
<td></td>
</tr>
<tr>
<td>PSHA</td>
<td>36</td>
<td>5</td>
<td>2</td>
<td>CMPB</td>
<td>9E</td>
<td>CMPB</td>
<td>F1</td>
<td></td>
</tr>
<tr>
<td>PSLA</td>
<td>37</td>
<td>6</td>
<td>0</td>
<td>CMPC</td>
<td>9F</td>
<td>CMPC</td>
<td>F5</td>
<td></td>
</tr>
<tr>
<td>PULB</td>
<td>33</td>
<td>7</td>
<td>2</td>
<td>CMPD</td>
<td>9A</td>
<td>CMPD</td>
<td>9F</td>
<td></td>
</tr>
<tr>
<td>ROLB</td>
<td>69</td>
<td>8</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROLB</td>
<td>69</td>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RORA</td>
<td>46</td>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RORB</td>
<td>56</td>
<td>A8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASLA</td>
<td>4B</td>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASLB</td>
<td>58</td>
<td>A0</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASRB</td>
<td>5A</td>
<td>A1</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASRB</td>
<td>5A</td>
<td>A2</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSA</td>
<td>10</td>
<td>B3</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TA</td>
<td>17</td>
<td>B4</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B5</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B6</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSTA</td>
<td>40</td>
<td>B8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSTA</td>
<td>40</td>
<td>B9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSB</td>
<td>50</td>
<td>BA</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSB</td>
<td>50</td>
<td>BB</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OEX</td>
<td>09</td>
<td>BC</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DESC</td>
<td>34</td>
<td>BD</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSB</td>
<td>35</td>
<td>BE</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSS</td>
<td>30</td>
<td>BF</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOP</td>
<td>02</td>
<td>C0</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTS</td>
<td>39</td>
<td>C1</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWI</td>
<td>3F</td>
<td>C2</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WAI</td>
<td>3E</td>
<td>C3</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLC</td>
<td>0C</td>
<td>C4</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLI</td>
<td>0E</td>
<td>C5</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLV</td>
<td>9A</td>
<td>C6</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEC</td>
<td>00</td>
<td>C7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEI</td>
<td>0F</td>
<td>C8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEV</td>
<td>03</td>
<td>C9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAP</td>
<td>06</td>
<td>CA</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPA</td>
<td>07</td>
<td>CB</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>23</td>
<td>CC</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>23</td>
<td>CD</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>23</td>
<td>CE</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX B: DEOCDE input for the M6800 microprocessor
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>86.0F</td>
<td>1</td>
</tr>
<tr>
<td>1A.11</td>
<td>1</td>
</tr>
<tr>
<td>16.17</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>1B</td>
<td>1</td>
</tr>
<tr>
<td>3A.3F</td>
<td>1</td>
</tr>
<tr>
<td>39</td>
<td>1</td>
</tr>
<tr>
<td>3B</td>
<td>1</td>
</tr>
<tr>
<td>3E.3F</td>
<td>1</td>
</tr>
<tr>
<td>48</td>
<td>1</td>
</tr>
<tr>
<td>43.44</td>
<td>1</td>
</tr>
<tr>
<td>46.4A</td>
<td>1</td>
</tr>
<tr>
<td>4C.40</td>
<td>1</td>
</tr>
<tr>
<td>4F</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>53.54</td>
<td>1</td>
</tr>
<tr>
<td>56.5A</td>
<td>1</td>
</tr>
<tr>
<td>5C.5D</td>
<td>1</td>
</tr>
<tr>
<td>5F</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>22.2F</td>
<td>2</td>
</tr>
<tr>
<td>3D</td>
<td>2</td>
</tr>
<tr>
<td>90.92</td>
<td>3</td>
</tr>
<tr>
<td>94.9C</td>
<td>3</td>
</tr>
<tr>
<td>9E.9F</td>
<td>3</td>
</tr>
<tr>
<td>D8.02</td>
<td>3</td>
</tr>
<tr>
<td>84.8B</td>
<td>3</td>
</tr>
<tr>
<td>DC.0F</td>
<td>3</td>
</tr>
<tr>
<td>81.82</td>
<td>4</td>
</tr>
<tr>
<td>84.86</td>
<td>4</td>
</tr>
<tr>
<td>88.88</td>
<td>4</td>
</tr>
<tr>
<td>C9.C2</td>
<td>4</td>
</tr>
<tr>
<td>C4.C6</td>
<td>4</td>
</tr>
<tr>
<td>C8.CB</td>
<td>4</td>
</tr>
<tr>
<td>8C</td>
<td>5</td>
</tr>
<tr>
<td>8E</td>
<td>5</td>
</tr>
<tr>
<td>CE</td>
<td>5</td>
</tr>
<tr>
<td>60</td>
<td>6</td>
</tr>
<tr>
<td>63.64</td>
<td>6</td>
</tr>
<tr>
<td>65.6A</td>
<td>6</td>
</tr>
<tr>
<td>6C.6F</td>
<td>6</td>
</tr>
<tr>
<td>A0.A2</td>
<td>6</td>
</tr>
<tr>
<td>44.AF</td>
<td>6</td>
</tr>
<tr>
<td>E8.E2</td>
<td>6</td>
</tr>
<tr>
<td>E4.E3</td>
<td>6</td>
</tr>
<tr>
<td>EE.EF</td>
<td>6</td>
</tr>
<tr>
<td>79</td>
<td>7</td>
</tr>
<tr>
<td>73.74</td>
<td>7</td>
</tr>
<tr>
<td>76.74</td>
<td>7</td>
</tr>
<tr>
<td>7C.7F</td>
<td>7</td>
</tr>
<tr>
<td>88.82</td>
<td>7</td>
</tr>
<tr>
<td>94.BF</td>
<td>7</td>
</tr>
<tr>
<td>F0.F2</td>
<td>7</td>
</tr>
<tr>
<td>F4.F8</td>
<td>7</td>
</tr>
<tr>
<td>FE.FF</td>
<td>7</td>
</tr>
</tbody>
</table>
APPENDIX C: XECUTE input for the M6800 microprocessor (classes 1, 3-7)
ABA - ADD ACCUMULATORS (A = A+B)
18 2,0
ADD TEMP1 SREG0 SREG1
AND TEMP1 TEMP1 FF*16
CALL HNZVCA TEMP1 SREG0 SREG1 SREG3 SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMP1
ENDINSTR

CLRA - CLEAR ACCUMULATOR A
4F 2,0
CLEAR SREG0
CLEAR SREG5
MOVE SREG6 1
CLEAR SREG7
CLEAR SREG8
ENDINSTR

CLRB - CLEAR ACCUMULATOR B
5F 2,0
CLEAR SREG1
CLEAR SREG5
MOVE SREG6 1
CLEAR SREG7
CLEAR SREG8
ENDINSTR

CBA - COMPARE ACCUMULATORS A, B.
11 2,0
CALL SUBTR TEMP1 SREG0 SREG1.
CALL NZVCS TEMP1 SREG0 SREG1 SREG5 SREG6 SREG7 SREG8.
ENDINSTR

COMA - 1’S COMPLEMENT, ACCE
43 2,0
COMONE SREG0 SREG0
AND SREG0 SREG0 FF*16
CALL NZR SREG0 SREG5 SREG6 SREG7.
MOVE SREG8 1
ENDINSTR

COMB - 1’S COMPLEMENT, ACCB
53 2,0
COMONE SREG1 SREG1
AND SREG1 SREG1 FF*16
CALL NZR SREG1 SREG5 SREG6 SREG7.
MOVE SREG8 1
ENDINSTR

NEGA - 2’S COMPLEMENT, ACCA
4B 2,0
COMTHW SREG0 SREG0
AND SREG0 SREG0 FF*16
CALL NZR SREG0 SREG5 SREG6 TEMP7.
CALL SETV SREG0 SREG7 80*16.
CALL SETV SREG6 SREG8 8.
ENDINSTR

NEGB - 2’S COMPLEMENT, ACCB
5B 2,0
COMTHW SREG1 SREG1
AND SREG1 SREG1 FF*16
CALL NZR SREG1 SREG5 SREG6 TEMP7.
CALL SETV SREG1 SREG7 80*16.
CALL SETV SREG6 SREG8 0.
ENDINSTR

0A - DECIMAL ADJUST ACCA

2.0
AND TEMP1 SREG0 F*16
SHRL TEMP2 SREG0 4
CALL DECIML TEMP1 SREG3 TEMP3.
ADD TEMP2 TEMP2 TEMP3
CALL DECIML TEMP2 SREG8 TEMP4.
IF TEMP4 EN 1
MOVE SREG0 1
ENDIF
CONCAT SREG0 TEMP2 (4) TEMP1
CALL NZR SREG0 SREG5 SREG6 TEMPT.
ENDINSTR

DECA - DECREMENT ACCA

4A 2.0
CALL SETV SREG0 SREG7 80*16.
ADD SREG0 SREG0 FF*16
AND SREG0 SREG0 FF*16
CALL NZR SREG0 SREG5 SREG6 TEMPT.
ENDINSTR

DEGB - DECREMENT ACCB

5A 2.0
CALL SETV SREG1 SREG7 80*16.
ADD SREG1 SREG1 FF*16
AND SREG1 SREG1 FF*16
CALL NZR SREG1 SREG5 SREG6 TEMPT.
ENDINSTR

INCA - INCREMENT ACCA

4C 2.0
CALL SETV SREG0 SREG7 7F*16.
ADD SREG0 SREG0 FF*16
AND SREG0 SREG0 FF*16
CALL NZR SREG0 SREG5 SREG6 TEMPT.
ENDINSTR

INCB - INCREMENT ACCB

5C 2.0
CALL SETV SREG1 SREG7 7F*16.
INCR SREG1 SREG1
AND SREG1 SREG1 FF*16
CALL NZR SREG1 SREG5 SREG6 TEMPT.
ENDINSTR

PSHA - PUSH ACCA ONTO STACK

36 4.0
PUSH SREG0
ENDINSTR

PSHB - PUSH ACCB ONTO STACK

37 4.0
PUSH SREG1
ENDINSTR

PULA - PULL DATA ONTO ACCA FROM STACK

32 4.0
PULL SREG0
ENDINSTR

PULB - PULL DATA ONTO ACCB FROM STACK

33 4.0
PULL SREG1
ENDINSTR

ROLA - ROTATE LEFT, ACCA
49 2.0
SHLL TEMPI SREG0 1
OR TEMPI SREG1 SREG8
CALL NZVCB TEMPI SREG0 SREG5 SREG6 SREG7 SREG8 80*16.
MOVE SREG0 TEMPI
ENDINSTR

ROLB - ROTATE LEFT, ACCB
59 2.0
SHLL TEMPI SREG1 1
OR TEMPI SREG1 SREG8
CALL NZVCB TEMPI SREG1 SREG6 SREG7 SREG8 80*16.
MOVE SREG1 TEMPI
ENDINSTR

RORA - ROTATE RIGHT, ACCA
46 2.0
SHRL TEMPI SREG0 1
IF SREG0 NE 0
OR TEMPI TEMPI 80*16
ENDIF
CALL NZVCB TEMPI SREG0 SREG5 SREG6 SREG7 SREG8 1.
MOVE SREG0 TEMPI
ENDINSTR

RORB - ROTATE RIGHT, ACCB
56 2.0
SHRL TEMPI SREG1 1
IF SREG8 NE 0
OR TEMPI TEMPI 80*16
ENDIF
CALL NZVCB TEMPI SREG1 SREG6 SREG7 SREG8 1.
MOVE SREG1 TEMPI
ENDINSTR

ASLA - SHIFT LEFT, ARITHMETIC, ACCA
48 2.0
SHLL TEMPI SREG0 1
CALL NZVCB TEMPI SREG0 SREG5 SREG6 SREG7 SREG8 80*16.
MOVE SREG0 TEMPI
ENDINSTR

ASLB - SHIFT LEFT, ARITHMETIC, ACCB
58 2.0
SHLL TEMPI SREG1 1
CALL NZVCB TEMPI SREG1 SREG5 SREG6 SREG7 SREG8 80*16.
MOVE SREG1 TEMPI
ENDINSTR

ASRA - SHIFT RIGHT, ARITHMETIC, ACCA
47 2.0
SHRA TEMPI SREG0 1
CALL NZVCB TEMPI SREG0 SREG5 SREG6 SREG7 SREG8 1.
MOVE SREG0 TEMPI
ENDINSTR

ASRB - SHIFT RIGHT, ARITHMETIC, ACCB
57 2.0
SHRA TEMPI SREG1 1
CALL NZVCB TEMPI SREG1 SREG5 SREG6 SREG7 SREG8 1.
MOVE SREG1 TEMPI
ENDINSTR
LSRA - LOGICAL SHIFT RIGHT, ACCA

44  2.0
SHRL TEMP1 SREG0 1
CALL NZVCB TEMP1 SREG5 SREG6 SREG7 SREG8 1.
MOVE SREG0 TEMP1
ENDINSTR

LSRB - LOGICAL SHIFT RIGHT, ACCB

54  2.0
SHRL TEMP1 SREG1 1
CALL NZVCB TEMP1 SREG5 SREG6 SREG7 SREG8 1.
MOVE SREG1 TEMP1
ENDINSTR

S3A - SUBTRACT ACCB FROM ACCA

10  2.0
CALL SUBTR TEMP1 SREG0 SREG1.
CALL NZVCS TEMP1 SREG0 SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMP1
ENDINSTR

T3A - TRANSFER ACC3 TO ACCA

17  2.0
MOVE SREG0 SREG1
CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR

T3B - TEST ACCB

5D  2.0
CALL NZR SREG5 SREG6 SREG7.
CLEAR SREG8
ENDINSTR

DSE - DECREMENT STACK POINTER

34  4.0
ADD STACKP STACKP FFFF16
AND STACKP STACKP FFFF16
ENDINSTR

INX - INCREMENT INDEX REGISTER

04  4.0
INCR SREG2 SREG2
AND SREG2 SREG2 FFFF16
CALL ZERO SREG2 SREG6.
ENDINSTR

INS - INCREMENT STACK POINTER

31  4.0
INCR STACKP STACKP
AND   STACKP  STACKP  FFFF*16
ENDINSTR
TXS - TRANSFER (SP = IX-1)
35  4.0
ADD   STACKP  SREG2  FFFF*16
AND   STACKP  STACKP  FFFF*16
ENDINSTR
TSX - TRANSFER (IX = SP+1)
30  4.0
INCR  SREG2  STACKP
AND   SREG2  SREG2  FFFF*16
ENDINSTR
NOP - NO OPERATION
02  2.0
EXIT
ENDINSTR
RTI - RETURN FROM INTERRUPT
33  10.0
DISPLAY  # RETURN FROM INTERRUPT #
ENDINSTR
RTS - RETURN FROM SUBROUTINE
39  5.0
DISPLAY  # RETURN FROM SUBROUTINE #
PULL   TEMP1
PULL   TEMP2
CONCAT  PC   TEMP1 (B)   TEMP2
ENDINSTR
SWI - SOFTWARE INTERRUPT
3F  12.0
DISPLAY  # SOFTWARE INTERRUPT #
ENDINSTR
WAI - WAIT FOR INTERRUPT
3E  9.0
DISPLAY  # WAIT FOR INTERRUPT #
ENDINSTR
CLC - CLEAR CARRY
0C  2.0
CLEAR  SREG8
ENDINSTR
CLI - CLEAR Interrupt MASK
0E  2.0
CLEAR  SREG4
ENDINSTR
CLV - CLEAR OVERFLOW
0A  2.0
CLEAR  SREG7
ENDINSTR
SEC - SET CARRY
0D  2.0
MOVE  SREG8  1
ENDINSTR
SEI - SET INTERRUPT
0F  2.0
MOVE  SREG4  1
ENDINSTR
SEV - SET OVERFLOW
0B  2.0
MOVE  SREG7  1
ENDINSTR

TAP - TRANSFER ACCA TO CCR

06

AND SREG4 SREGO 1
AND SREG7 SREGO 2
AND SREG6 SREGO 4
AND SREG5 SREGO 8
AND SREG4 SREGO 10+16
AND SREG3 SREGO 20+16

ENDINSTR

TPA - TRANSFER CCR'S TO ACCA

07

MOVE SREG0 C0+16
SHLC TEMPI SREG3 5
OR SREG0 TEMPI SREG0
SHLC TEMPI SREG4 4
OR SREG0 TEMPI SREG0
SHLC TEMPI SREG5 3
OR SREG0 TEMPI SREG0
SHLC TEMPI SREG6 2
OR SREG0 TEMPI SREG0
SHLC TEMPI SREG7 1
OR SREG0 TEMPI SREG0
OR SREG0 SREG0 SREG0

ENDINSTR

ENDINDEF

DEFINE 2.0
CALL SIGN TEMPI TEMPI.
CLEAR TEMPI
AND TEMPS TEMPS TEMPS
IF TEMPS NE 0
MOVE TEMPS 1
ENDiF
XOR TEMPS TEMPS TEMPS
RETURN

ENDINSTR

DEFINE 2.0
CALL CARRY TEMPI TEMPI.
AND TEMPS TEMPS TEMPS
AND TEMPS TEMPS TEMPS
AND TEMPS TEMPS TEMPS
OR TEMPS TEMPS TEMPS
OR TEMPS TEMPS TEMPS
AND TEMPS TEMPS TEMPS
CLEAR TEMPS
IF TEMPS NE 0
MOVE TEMPS 1
ENDiF
RETURN

ENDINSTR

DEFINE 2.0
CLEAR TEMPS
AND TEMPS TEMPS TEMPS
IF TEMPS NE 0
MOVE TEMPS 1
ENDiF
RETURN
DEFINE ZERO TEMPI TEMPS.
CLEAR TEMPS.
IF TEMPI EO 0 MOVE TEMPS 1 ENDF.
ENDINSTR
DEFINE OVERFA TEMPI TEMPS TEMPS TEMPS.
COMONE TEMPS TEMPS TEMPS.
AND TEMPS TEMPS TEMPS.
COMONE TEMPS TEMPS.
AND TEMPS TEMPS TEMPS.
OR TEMPS TEMPS TEMPS.
AND TEMPS TEMPS TEMPS.
CLEAR TEMPS.
IF TEMPS NE 0 MOVE TEMPS 1 ENDF.
RETURN
ENDINSTR
DEFINE CARYSA TEMPI TEMPS TEMPS TEMPS.
AND TEMPS TEMPS TEMPS.
COMONE TEMPS TEMPS.
AND TEMPS TEMPS TEMPS.
OR TEMPS TEMPS TEMPS.
AND TEMPS TEMPS TEMPS.
CLEAR TEMPS.
IF TEMPS NE 0 MOVE TEMPS 1 ENDF.
RETURN
ENDINSTR
DEFINE HNZvCA TEMPI TEMPS TEMPS TEMPS TEMPS TEMPS TEMPS.
CALL CARYHA TEMPS TEMPS TEMPS.
CALL SIGN TEMPS TEMPS.
CALL ZERO TEMPS TEMPS.
CALL OVERFA TEMPS TEMPS TEMPS.
CALL CARYSA TEMPS TEMPS TEMPS.
RETURN
ENDINSTR
DEFINE OVERFS TEMPI TEMPS TEMPS TEMPS.
COMONE TEMPS TEMPS.
AND TEMPS TEMPS TEMPS.
COMONE TEMPS TEMPS.
AND TEMPS TEMPS TEMPS.
OR TEMPS TEMPS TEMPS.
AND TEMPS TEMPS TEMPS.
CLEAR TEMPS.
IF TEMPS NE 0 MOVE TEMPS 1
CALL SIGN TEMP1 TEMP4.
CALL ZERO TEMP1 TEMP5.
CALL OVERFS TEMP1 TEMP2 TEMP3 TEMP6.
CALL CARYSS TEMP1 TEMP2 TEMP3 TEMP7.
RETURN
ENDINSTR
ENDCLASS
ADDAD - (ACCA := ACCA + M)

98  3.0
    CALL GETOPN IMM1 ABUS.
    ADD TEMPI SREG0 DBUS
    AND TEMPI TEMPI FF+16
    CALL HNZVCA TEMPI SREG0 DBUS SREG3 SREG5 SREG6 SREG7 SREG8.
    MOVE SREG0 TEMPI
ENDINSTR

ADDAD - (ACCA := ACCA + M)

09  3.0
    CALL GETOPN IMM1 ABUS.
    ADD TEMPI SREG1 DBUS
    AND TEMPI TEMPI FF+16
    CALL HNZVCA TEMPI SREG1 DBUS SREG3 SREG5 SREG6 SREG7 SREG8.
    MOVE SREG1 TEMPI
ENDINSTR

ADCAO - (ACCA := ACCA + M + CARRY)

99  3.0
    CALL GETOPN IMM1 ABUS.
    ADD TEMPI SREG0 DBUS
    ADD TEMPI TEMPI SREG8
    AND TEMPI TEMPI FF+16
    CALL HNZVCA TEMPI SREG0 DBUS SREG3 SREG5 SREG6 SREG7 SREG8.
    MOVE SREG0 TEMPI
ENDINSTR

ADCOO - (ACCB := ACCA + M + CARRY)

09  3.0
    CALL GETOPN IMM1 ABUS.
    ADD TEMPI SREG1 DBUS
    ADD TEMPI TEMPI SREG8
    AND TEMPI TEMPI FF+16
    CALL HNZVCA TEMPI SREG1 DBUS SREG3 SREG5 SREG6 SREG7 SREG8.
    MOVE SREG1 TEMPI
ENDINSTR

ANDAD - (ACCA := ACCA .AND. M)

94  3.0
    CALL GETOPN IMM1 ABUS.
    AND SREG0 SREG0 DBUS
    CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR

ANDAD - (ACCA := ACCA .AND. M)

94  3.0
    CALL GETOPN IMM1 ABUS.
    AND SREG1 SREG1 DBUS
    CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR

BITAO - (ACCA .AND. MEMORY)

95  3.0
    CALL GETOPN IMM1 ABUS.
    AND TEMPI SREG0 DBUS
    CALL NZR TEMPI SREG5 SREG6 SREG7.
ENDINSTR

BITAD - (ACC3 .AND. MEMORY)
```
D5 3.0
CALL GETOPN IMM1 ABUS.
AND TEMP1 SREG1 DBUS.
CALL NZR TEMP1 SREG5 SREG6 SREG7.
ENDINSTR

CMPAD - (COMPARE ACCA WITH MEMORY)
91 3.0
CALL GETOPN IMM1 ABUS.
CALL SUBTR TEMP1 SREG0 DBUS.
CALL NZVCS TEMP1 SREG0 DBUS SREG5 SREG6 SREG7 SREG8.
ENDINSTR

CMPBD - (COMPARE ACCB WITH MEMORY)
D1 3.0
CALL GETOPN IMM1 ABUS.
CALL SUBTR TEMP1 SREG1 DBUS.
CALL NZVCS TEMP1 SREG0 DBUS SREG5 SREG6 SREG7 SREG8.
ENDINSTR

EORAD - EXCLUSIVE OR ACCA W/MEMORY
9B 3.0
CALL GETOPN IMM1 ABUS.
XOR SREG0 SREG0 DBUS.
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR

EORBD - EXCLUSIVE OR ACCB W/MEMORY
DB 3.0
CALL GETOPN IMM1 ABUS.
XOR SREG1 SREG1 DBUS.
CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR

LDAAO - LOAD ACCA FROM MEMORY
95 3.0
CALL GETOPN IMM1 ABUS.
MOVE SREG0 DBUS.
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR

LDABD - LOAD ACCB FROM MEMORY
D6 3.0
CALL GETOPN IMM1 ABUS.
MOVE SREG1 DBUS.
CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR

ORAAO - ACCA := ACCA .OR. #
9A 3.0
CALL GETOPN IMM1 ABUS.
OR SREG0 SREG0 DBUS.
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR

ORABD - ACCB := ACCB .OR. #
DA 3.0
CALL GETOPN IMM1 ABUS.
OR SREG1 SREG1 DBUS.
CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR

STAAO - STORE ACCA INTO MEMORY
97 4.0
MOVE ABUS IMM1
MOVE DBUS SPEGO
WRITEM
```

CALL NZR D9US SREG5 SREG6 SREG7.
ENDINSTR

STA8D - STORE ACCP INTO MEMORY

D7 4.0
MOVE ABUS IMM1
MOVE D9US SREG1

W8TED
CALL NZR D9US SREG5 SREG6 SREG7.

ENDINSTR

SU3AD - (ACCA = ACCA - M)

90 3.0
CALL GETOPN IMM1 ABUS.
CALL SU3TR TEMPI SREG0 DBUS.
CALL NZVCS TEMP1 SREG0 DBUS SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMPI

ENDINSTR

SU3AD - (ACCA = ACCA - M)

08 3.0
CALL GETOPN IMM1 ABUS.
CALL SU3TR TEMPI SREG0 DBUS.
CALL NZVCS TEMPI SREG0 DBUS SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMPI

ENDINSTR

S36AD - (ACCA=ACCA - M - CARRY)

92 3.0
CALL GETOPN IMM1 ABUS.
CALL SU3TR TEMPI SREG0 DBUS.
CALL NZVCS TEMPI SREG0 DBUS SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMPI

ENDINSTR

S36AD - (ACCA=ACCA - M - CARRY)

D2 3.0
CALL GETOPN IMM1 ABUS.
CALL SU3TR TEMPI SREG0 DBUS.
CALL NZVCS TEMPI SREG0 DBUS SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMPI

ENDINSTR

CPXO - COMPARE IX W/ MEMORY

9C 4.0
CALL GETLOP TEMPI IMM1 ABUS DBUS.
CALL SU3TR TEMPI SREG2 TEMP3.
AND TEMPI TEMP4 0000+16
CLEAR SREG6
CLEAR SREG7
IF TEMPI NE 0
MOVE SREG5 1
MOVE SREG7 1
ENDIF
CALL ZERO TEMPI SREG6.

ENDINSTR

LDXO - LOAD IX FROM MEMORY

DE 4.0
CALL GETLOP SREG2 IMM1 ABUS DBUS.
CALL NZVLO SREG2 SREG5 SREG6 SREG7.

ENDINSTR

LO50 - LOAD STACK POINTER FROM MEMORY
CALL GETLOP STACKP IMM1 ABUS DBUS.
CALL NZVLO STACKP SREG5 SREG6 SREG7.
END INSTR

STXD - STORE IX INTO MEMORY

CALL STOLOP SREG2 IMM1 ABUS DBUS.
CALL NZVLO SREG2 SREG6 SREG7.
END INSTR

END INSDEF
DEFINE GETOPN TEMPI TEMP2.
MOVE TEMP2 TEMPI
READ
RETURN
END INSTR

DEFINE NZR TEMPI TEMP2 TEMP3 TEMPI*.
CALL SIGN TEMPI TEMP2.
CALL ZERC TEMPI TEMP3.
MOVE TEMPI* 0
RETURN
END INSTR

DEFINE GETLOP TEMPI TEMP2 TEMP3 TEMP4.
CALL GETOPN TEMP2 TEMP3.
MOVE TEMP7 TEMP4.
INCR TEMP5 TEMP3.
CALL GETOPN TEMP5 TEMP3.
MOVE TEMP8 TEMP4.
CONCAT TEMP1 TEMP7 (8) TEMP8
RETURN
END INSTR

DEFINE STOLOP TEMPI TEMP2 TEMP3 TEMP4.
AND TEMP6 TEMPI FF*16
AND TEMP5 TEMPI FF00*16
SHRL TEMP5 TEMP5 8
MOVE TEMP4 TEMP5
MOVE TEMP3 TEMP2
WRITED
MOVE TEMP4 TEMP6
INCR TEMP3 TEMP3
WRITED
RETURN
END INSTR

DEFINE NZVLO TEMPI TEMP2 TEMP3 TEMP4.
AND TEMP5 TEMPI FF00*16
CLEAR TEMP2
IF TEMP5 NE 0
MOVE TEMP2 1
ENDIF
CALL ZERO TEMPI TEMP3.
MOVE TEMP4 0
RETURN
END INSTR

DEFINE SUBLOP TEMPI TEMP2 TEMP3.

- 96 -
CLEAR TEMP1
IF TEMP2 ED TEMP3
RETURN
ENDIF
IF TEMP3 ED 0
MOVE TEMP1 TEMP2
RETURN
ENDIF
IF TEMP3 ED 1
ADD TEMP1 TEMP2 FFFF+16
RETURN
ENDIF
COMTHO TEMP1 TEMP3
AND TEMP1 TEMP1 FFFF+16
ADD TEMP1 TEMP2 TEMP1
AND TEMP1 TEMP1 FFFF+16
RETURN
ENDINSTR
ENDCLASS
ADDAI - (ACCA := ACCA + IMMED.)
88 2.0
ADD TEMP1 SREG0 IMM1
AND TEMP1 TEMP1 FF+16
CALL HNZPCA TEMP1 SREG0 IMM1 SREG3 SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMP1
ENDINSTR

ADDBI - (ACCB := ACCB + IMMED.)
C9 2.0
ADD TEMP1 SREG1 IMM1
AND TEMP1 TEMP1 FF+16
CALL HNZVCA TEMP1 SREG1 IMM1 SREG3 SREG5 SREG6 SREG7 SREG8.
MOVE SREG1 TEMP1
ENDINSTR

ADCAI - (ACCA := ACCA + IMMED.)
A0 2.0
ADD TEMP1 SREG0 IMM1
ADD TEMP1 TEMP1 SREG3
AND TEMP1 FF+16
CALL HNZPCA TEMP1 SREG0 IMM1 SREG3 SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMP1
ENDINSTR

ADCBI - (ACCB := ACCB + IMMED.)
C9 2.0
ADD TEMP1 SREG1 IMM1
ADD TEMP1 TEMP1 SREG8
AND TEMP1 FF+16
CALL HNZVCA TEMP1 SREG1 IMM1 SREG3 SREG5 SREG6 SREG7 SREG8.
MOVE SREG1 TEMP1
ENDINSTR

ANDAI - (ACCA := ACCA .AND. IMMED.)
84 2.0
AND SREG0 SREG0 IMM1.
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR

ANDBI - (ACCB := ACCB .AND. IMMED.)
C4 2.0
AND SREG1 SREG1 IMM1
CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR

BITAI - (ACCA := ACCA + IMMEO.)
85 2.0
AND TEMP1 SREG0 IMM1
CALL NZR TEMP1 SREG5 SREG6 SREG7.
ENDINSTR

BITBI - (ACCB := ACCB + IMMEO.)
C5 2.0
AND TEMP1 SREG1 IMM1
CALL NZR TEMP1 SREG5 SREG6 SREG7.
ENDINSTR

CMPAI - COMPARE ACCA W/ IMMEO.
81 2.1
CALL SUBTR TEMP1 SREG0 IMM1.
CALL NZVCS TEMP1 SREG0 IMM1 SREG5 SREG6 SREG7 SREG8.
ENDINSTR

CMF3I - COMPARE ACC3 W/ IMMED.
C1 2,1
CALL SUBTR TEMP1 SREG1 IMM1.
CALL NZVCS TEMP1 SREG0 IMM1 SREG5 SREG6 SREG7 SREG8.
ENDINSTR

ENDINSTR

EORAI - EXCLUSIVE OR ACCA W/ IMMED.
88 2,1
XOR SREG0 SREG0 IMM1
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR

ENDINSTR

EORBI - EXCLUSIVE OR ACCB W/ IMMED.
C8 2,6
XOR SREG1 SREG1 IMM1
CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR

ENDINSTR

LOAAI - LOAD ACCA FROM IMMED.
86 2,0
MOVE SREG0 IMM1
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR

ENDINSTR

LOABI - LOAD ACCB FROM IMMED.
C6 2,3
MOVE SREG1 IMM1
CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR

ENDINSTR

ORAII - (ACCA = ACCA .OR. IMMED.)
8A 2,6
OR SREG0 SREG0 IMM1
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR

ENDINSTR

ORABI - (ACCB = ACCB .OR. IMMED.)
CA 2,0
OR SREG1 SREG1 IMM1
CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR

ENDINSTR

SUBAI - (ACCA = ACCA - IMMED.)
8G 2,0
CALL SUBTR TEMP1 SREG0 IMM1.
CALL NZVCS TEMP1 SREG0 IMM1 SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMP1
ENDINSTR

ENDINSTR

SUBBI - (ACCB = ACCB - IMMED.)
C0 2,0
CALL SUBTR TEMP1 SREG1 IMM1.
CALL NZVCS TEMP1 SREG1 IMM1 SREG5 SREG6 SREG7 SREG8.
MOVE SREG1 TEMP1
ENDINSTR

ENDINSTR

S3CAI - (ACCAI=ACCA - IMMED. - CAPRY)
82 2,6
CALL SUBTR TEMP1 SREG0 IMM1.
CALL SUBTR TEMP2 TEMP1 SREG8.
CALL NZVCS TEMP2 SREG0 IMM1 SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMP2
ENDINSTR

ENDINSTR

SECBI - (ACCB = ACCB - IMMED. - CARRY)
C2 2,0
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR

ENDINSTR

- 99 -
CALL SUBTR TEMP1 SREG1 IMM1.
CALL SUBTR TEMP2 TEMP1 SREGA.
CALL NZVCS TEMP2 SREG1 IMM1 SREG5 SREG6 SREG7 SREG8.
MOVE SREG1 TEMP2
ENDINSTR
ENDINSDEF
ENDCLASS
CPX| - COMPAIR IX W/ IMEED.

CALL SUBL0 TEMP4 SREG2 IMM1.
AND TEMP1 TEMP4 8000*16
CLEAR SREG5
CLEAR SREG7
IF TEMP1 NE 0
    MOVE SREG5 1
    MOVE SREG7 1
ENDIF
CALL ZERO TEMP«
ENOINSTR

LDXI - LOAD IX FROM IMEED.
MOVE SREG2 IMM1.
CALL NZVLO SREG2 SREG5 SREG6 SREG7.
ENOINSTR

LOSI - LOAD STACK POINTER FROM IMEED.
MOVE STACKP IMM1
CALL NZVLO STACKP SREG5 SREG6 SREG7.
ENOINSTR

ENDINS0EF
ENDCLASS
ADDAX - (ACCA := ACCA + M)
A9 5.0
CALL GETXOP IMM1 SREG2 ABUS.
ADD TEMP1 SREG0 OBUS
AND TEMP1 TEMP1 FF*16
CALL HNZVCA TEMP1 SREG0 OBUS SREG3 SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMP1
ENDINSTR
ADDAX - (ACCA := ACCA + M)
EB 5.0
CALL GETXOP IMM1 SREG2 ABUS.
ADD TEMP1 SREG1 OBUS
AND TEMP1 TEMP1 FF*16
CALL HNZVCA TEMP1 SREG1 OBUS SREG3 SREG5 SREG6 SREG7 SREG8.
MOVE SREG1 TEMP1
ENDINSTR
ADDAX - (ACCA := ACCA + M + CARRY)
A9 5.0
CALL GETXOP IMM1 SREG2 ABUS.
ADD TEMP1 SREG0 OBUS
ADD TEMP1 TEMP1 SREG8
AND TEMP1 TEMP1 FF*16
CALL HNZVCA TEMP1 SREG0 OBUS SREG3 SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMP1
ENDINSTR
ADCBX - (ACCA := ACCA + M + CARRY)
EB 5.0
CALL GETXOP IMM1 SREG2 ABUS.
ADD TEMP1 SREG1 OBUS
ADD TEMP1 TEMP1 SREG8
AND TEMP1 TEMP1 FF*16
CALL HNZVCA TEMP1 SREG1 OBUS SREG3 SREG5 SREG6 SREG7 SREG8.
MOVE SREG1 TEMP1
ENDINSTR
ANDAX - (ACCA := ACCA .AND. M)
A4 5.0
CALL GETXOP IMM1 SREG2 ABUS.
AND SREG0 SREG0 OBUS.
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR
ANDBX - (ACCA := ACCA .AND. M)
E4 5.0
CALL GETXOP IMM1 SREG2 ABUS.
AND SREG1 SREG1 OBUS.
CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR
BITAX - (ACCA .AND. MEMORY)
A5 5.0
CALL GETXOP IMM1 SREG2 ABUS.
AND TEMP1 SREG0 OBUS
CALL NZR TEMP1 SREG5 SREG6 SREG7.
ENDINSTR
BITBX - (ACCA .AND. MEMORY)
E5 5.3
CALL GETXOP IMM1 SREG2 ABUS.
AND TEMP1 SREG1 DBUS
CALL NZR TEMP1 SREG5 SREG6 SREG7.
ENDINSTR

CLRX - CLEAR MEMORY
6F 7.0
CALL GETXOP IMM1 SREG2 ABUS.
CLEAR DBUS
WRITE
CLEAR SREG5
MOVE SREG6 1
CLEAR SREG7
CLEAR SREG8
ENDINSTR

CMPAX - (COMPARE ACCA WITH MEMORY)
A1 5.0
CALL GETXOP IMM1 SREG2 ABUS.
CALL SUBTR TEMP1 SREG6 DBUS.
CALL NZVC TEMP1 SREG0 DBUS SREG5 SREG6 SREG7 SREG8.
ENDINSTR

CMPPBX - (COMPARE ACCB WITH MEMORY)
E1 5.0
CALL GETXOP IMM1 SREG2 ABUS.
CALL SUBTR TEMP1 SREG1 DBUS.
CALL NZVC TEMP1 SREG0 DBUS SREG5 SREG6 SREG7 SREG8.
ENDINSTR

COMX - 1'S COMPLEMENT MEMORY
63 7.0
CALL GETXOP IMM1 SREG2 ABUS.
COMONE DBUS DBUS
AND DBUS DBUS FF*16
WRITE
CALL NZR DBUS SREG5 SREG6 SREG7.
MOVE SREG8 1
ENDINSTR

NEGX - 2'S COMPLEMENT MEMORY
60 7.0
CALL GETXOP IMM1 SREG2 ABUS.
COMTHO DBUS DBUS
AND DBUS DBUS FF*16
WRITE
CALL NZR DBUS SREG5 SREG6 TEMP7.
CALL SETV DBUS SREG7 80*15.
CALL SETV SREG6 SREG8 0.
ENDINSTR

DEDX - DECREMENT MEMORY
6A 7.0
CALL GETXOP IMM1 SREG2 ABUS.
CALL SETV DBUS SREG7 80*15.
ADD DBUS DBUS FF*16
AND DBUS DBUS FF*16
WRITE
CALL NZR DBUS SREG5 SREG6 TEMP7.
ENDINSTR

EORAX - EXCLUSIVE OR ACCA W/MEMORY
A8 5.0
CALL GETXOP IMM1 SREG2 ABUS.
XOR SREG0 SREGD DBUS
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR

E8 5.0
CALL GETXOP IMM1 SREG2 ABUS.
XOR SREG1 SREGD DBUS
CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR

INCX - INCREMENT MEMORY
6C 7.0
CALL GETXOP IMM1 SREG2 ABUS.
CALL SETV DBUS SREG7 FF+16.
INCR DBUS DBUS
AND DBUS DBUS FF16
WRITED
CALL NZR DBUS SREG5 SREG6 TEMP7.
ENDINSTR

LOAX - LOAD ACCA FROM MEMORY
A6 5.0
CALL GETXOP IMM1 SREG2 ABUS.
MOVE SREG0 DBUS
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR

LOABX - LOAD ACCB FROM MEMORY
E6 5.0
CALL GETXOP IMM1 SREG2 ABUS.
MOVE SREG1 DBUS
CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR

ORAAX - (ACCA := ACCA .OR. M)
AA 5.0
CALL GETXOP IMM1 SREG2 ABUS.
OR SREG0 SREG0 DBUS
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR

ORABX - (ACCB := ACCB .OR. M)
EA 5.0
CALL GETXOP IMM1 SREG2 ABUS.
OR SREG1 SREG1 DBUS
CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR

ROLX - ROTATE LEFT, MEMORY
69 7.0
CALL GETXOP IMM1 SREG2 ABUS.
SHLL TEMP1 DBUS 1
OR TEMP1 TEMP1 SREG6
CALL NZVCB TEMP1 DBUS SREG5 SREG6 SREG7 SREG8 FF+16.
MOVE DBUS TEMP1
WRITED
ENDINSTR

RORX - ROTATE RIGHT, MEMORY
66 7.0
CALL GETXOP IMM1 SREG2 ABUS.
SHRL TEMP1 DBUS 1
IF SREG8 NE 0
OR TEMP1 TEMP1 FF+16
ENDIF
CALL NZVCB TEMP1 DBUS SREG5 SREG6 SREG7 SREG8 1.
MOVE DBUS TEMP1
WRITED ENDINSTR

ASLX - SHIFT LEFT, ARITHMETIC, MEMORY
68 7.0
CALL GETXOP IMM1 SREG2 ABUS.
SHLL TEMP1 DBUS 1
CALL NZVCB TEMP1 DBUS SREG5 SREG6 SREG7 SREG8 80*16.
MOVE DBUS TEMP1
WRITED ENDINSTR

ASRX - SHIFT RIGHT, ARITHMETIC, MEMORY
67 7.0
CALL GETXOP IMM1 SREG2 ABUS.
SHRA TEMP1 DBUS 1
CALL NZVCB TEMP1 DBUS SREG5 SREG6 SREG7 SREG8 1.
MOVE DBUS TEMP1
WRITED ENDINSTR

LSRX - LOGICAL SHIFT RIGHT, MEMORY
64 7.0
CALL GETXOP IMM1 SREG2 ABUS.
SHRL TEMP1 DBUS 1
CALL NZVCB TEMP1 DBUS SREG5 SREG6 SREG7 SREG8 1.
MOVE DBUS TEMP1
WRITED ENDINSTR

STIX - TEST MEMORY
6D 7.0
CALL GETXOP IMM1 SREG2 ABUS.
NZR DBUS SREG5 SREG6 SREG7.
CLEAR SREG8 ENDINSTR

STAAX - STORE ACCA INTO MEMORY
A7 6.0
CALL GETXOP IMM1 SREG2 ABUS.
MOVE DBUS SREG0
WRITED
CALL NZR DBUS SREG5 SREG6 SREG7.
ENDINSTR

STABX - STORE ACCB INTO MEMORY
E7 6.0
CALL GETXOP IMM1 SREG2 ABUS.
MOVE DBUS SREG1
WRITED
CALL NZR DBUS SREG5 SREG6 SREG7.
ENDINSTR

SUBAX - (ACCA = ACCA - M)
A0 5.0
CALL GETXOP IMM1 SREG2 ABUS.
CALL SUBAX TEMP1 SREG0 DBUS.
CALL NZVCB TEMP1 SREG0 DBUS SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMP1
ENDINSTR

SUBBX - (ACCB = ACCB - M)
E0 5.0
CALL GETXOP IMM1 SREG2 ABUS.
ADDAB - (ACCA = ACCA + M)
B9 4.0
CALL GETEOP IMM1 ABUS.
ADD TEMP1 SREG0 DBUS
AND TEMP1 TEMP1 FF+16
CALL HNZVCA TEMP1 SREG0 DBUS SREG5 SREG6 SREG7 SREG8.
MOVE SREGO TEMP1
ENDINSTR
ADDAB - (ACCA = ACCA + M)
F8 4.0
CALL GETEOP IMM1 ABUS.
ADD TEMP1 SREG1 DBUS
ADD TEMP1 TEMP1 SREGA
AND TEMP1 TEMP1 FF+16
CALL HNZVCA TEMP1 SREG1 DBUS SREG3 SREG5 SREG6 SREG7 SREG8.
MOVE SREG1 TEMP1
ENDINSTR
ADDAE - (ACCA = ACCA + M + CARRY)
B9 4.0
CALL GETEOP IMM1 ABUS.
ADD TEMP1 SREG0 DBUS
ADD TEMP1 TEMP1 SREGA
AND TEMP1 TEMP1 FF+16
CALL HNZVCA TEMP1 SREG0 DBUS SREG3 SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMP1
ENDINSTR
ADDAE - (ACCA = ACCA + M + CARRY)
F9 4.0
CALL GETEOP IMM1 ABUS.
ADD TEMP1 SREG1 DBUS
ADD TEMP1 TEMP1 SREGA
AND TEMP1 TEMP1 FF+16
CALL HNZVCA TEMP1 SREG1 DBUS SREG3 SREG5 SREG6 SREG7 SREG8.
MOVE SREG1 TEMP1
ENDINSTR
ANDAB - (ACCA = ACCA .AND. M)
B4 4.0
CALL GETEOP IMM1 ABUS.
AND SREG0 SREG0 DBUS.
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR
ANDAB - (ACCA .AND. M)
F4 4.0
CALL GETEOP IMM1 ABUS.
AND SREG1 SREG1 DBUS
CALL NZR SREG1 SREG5 SREG6 SREG7.
ENDINSTR
BITAE - (ACCA .AND. MEMORY)
B5 4.0
CALL GETEOP IMM1 ABUS.
AND TEMP1 SREG0 DBUS
CALL NZR TEMP1 SREG5 SREG6 SREG7.
ENDINSTR
BITAE - (ACCA .AND. MEMORY)
B5 4.0
CALL GETEOP IMM1 ABUS.
AND TEMP1 SREGO DBUS
CALL NZR TEMP1 SREG5 SREG6 SREG7.
ENDINSTR
<table>
<thead>
<tr>
<th>CLS7</th>
<th>59</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL GEOP IMM1 ABUS.</td>
<td>CLS7  60</td>
</tr>
<tr>
<td>AND TEMPI SREG1 DBUS.</td>
<td>CLS7  61</td>
</tr>
<tr>
<td>CALL NZR TEMPI SREG5 SREG6 SREG7.</td>
<td>CLS7  62</td>
</tr>
<tr>
<td>ENINSTR</td>
<td>CLS7  63</td>
</tr>
<tr>
<td>CLRE - CLEAR MEMORY</td>
<td>CLS7  64</td>
</tr>
<tr>
<td>7F  6.0</td>
<td>CLS7  65</td>
</tr>
<tr>
<td>CALL GEOP IMM1 ABUS.</td>
<td>CLS7  66</td>
</tr>
<tr>
<td>CLEAR DBUS</td>
<td>CLS7  67</td>
</tr>
<tr>
<td>WRITED</td>
<td>CLS7  68</td>
</tr>
<tr>
<td>CLEAR SREG6</td>
<td>CLS7  69</td>
</tr>
<tr>
<td>MOVE SREG6 1</td>
<td>CLS7  70</td>
</tr>
<tr>
<td>CLEAR SREG7</td>
<td>CLS7  71</td>
</tr>
<tr>
<td>CLEAR SREG8</td>
<td>CLS7  72</td>
</tr>
<tr>
<td>ENINSTR</td>
<td>CLS7  73</td>
</tr>
<tr>
<td>CMPE - COMPLEMENT MEMORY</td>
<td>CLS7  74</td>
</tr>
<tr>
<td>81  4.0</td>
<td>CLS7  75</td>
</tr>
<tr>
<td>CALL GEOP IMM1 ABUS.</td>
<td>CLS7  76</td>
</tr>
<tr>
<td>CALL SUB TR TEMPI SREG0 DBUS.</td>
<td>CLS7  77</td>
</tr>
<tr>
<td>CALL NZVCS TEMPI SREG0 DBUS SREG5 SREG6 SREG7 SREG8.</td>
<td>CLS7  78</td>
</tr>
<tr>
<td>ENINSTR</td>
<td>CLS7  79</td>
</tr>
<tr>
<td>CMPE - COMPLEMENT MEMORY</td>
<td>CLS7  80</td>
</tr>
<tr>
<td>8F  4.0</td>
<td>CLS7  81</td>
</tr>
<tr>
<td>CALL GEOP IMM1 ABUS.</td>
<td>CLS7  82</td>
</tr>
<tr>
<td>CALL SUB TR TEMPI SREG0 DBUS.</td>
<td>CLS7  83</td>
</tr>
<tr>
<td>CALL NZVCS TEMPI SREG0 DBUS SREG5 SREG6 SREG7 SREG8.</td>
<td>CLS7  84</td>
</tr>
<tr>
<td>ENINSTR</td>
<td>CLS7  85</td>
</tr>
<tr>
<td>COME - 1'S COMPLEMENT MEMORY</td>
<td>CLS7  86</td>
</tr>
<tr>
<td>73  6.0</td>
<td>CLS7  87</td>
</tr>
<tr>
<td>CALL GEOP IMM1 ABUS.</td>
<td>CLS7  88</td>
</tr>
<tr>
<td>COMWO DBUS DBUS</td>
<td>CLS7  89</td>
</tr>
<tr>
<td>AND DBUS DBUS FF+16</td>
<td>CLS7  90</td>
</tr>
<tr>
<td>WRITED</td>
<td>CLS7  91</td>
</tr>
<tr>
<td>CALL NZR DBUS SREG5 SREG6 SREG7.</td>
<td>CLS7  92</td>
</tr>
<tr>
<td>MOVE SREG8 1</td>
<td>CLS7  93</td>
</tr>
<tr>
<td>ENINSTR</td>
<td>CLS7  94</td>
</tr>
<tr>
<td>NEGE - 2'S COMPLEMENT MEMORY</td>
<td>CLS7  95</td>
</tr>
<tr>
<td>70  6.0</td>
<td>CLS7  96</td>
</tr>
<tr>
<td>CALL GEOP IMM1 ABUS.</td>
<td>CLS7  97</td>
</tr>
<tr>
<td>COMWO DBUS DBUS</td>
<td>CLS7  98</td>
</tr>
<tr>
<td>AND DBUS DBUS FF+16</td>
<td>CLS7  99</td>
</tr>
<tr>
<td>WRITED</td>
<td>CLS7 100</td>
</tr>
<tr>
<td>CALL NZR DBUS SREG5 SREG6 SREG7.</td>
<td>CLS7 101</td>
</tr>
<tr>
<td>CALL SETV DBUS SREG7 80+16.</td>
<td>CLS7 102</td>
</tr>
<tr>
<td>CALL SETV SREG6 SREG8 8.</td>
<td>CLS7 103</td>
</tr>
<tr>
<td>ENINSTR</td>
<td>CLS7 104</td>
</tr>
<tr>
<td>DECE - DECREMENT MEMORY</td>
<td>CLS7 105</td>
</tr>
<tr>
<td>7A  6.0</td>
<td>CLS7 106</td>
</tr>
<tr>
<td>CALL GEOP IMM1 ABUS.</td>
<td>CLS7 107</td>
</tr>
<tr>
<td>CALL SETV DBUS SREG7 80+16.</td>
<td>CLS7 108</td>
</tr>
<tr>
<td>ADD DBUS DBUS FF+16</td>
<td>CLS7 109</td>
</tr>
<tr>
<td>AND DBUS DBUS FF+16</td>
<td>CLS7 110</td>
</tr>
<tr>
<td>WRITED</td>
<td>CLS7 111</td>
</tr>
<tr>
<td>CALL NZR DBUS SREG5 SREG6 SREG7.</td>
<td>CLS7 112</td>
</tr>
<tr>
<td>ENINSTR</td>
<td>CLS7 113</td>
</tr>
<tr>
<td>EDRAE - EXCLUSIVE OR ACCA W/ MEMORY</td>
<td>CLS7 114</td>
</tr>
<tr>
<td>B8  4.0</td>
<td>CLS7 115</td>
</tr>
<tr>
<td>CALL GEOP IMM1 ABUS.</td>
<td>CLS7 116</td>
</tr>
</tbody>
</table>
XOR SREG0 SREG0 DBUS
CALL NZR SREG0 SREG5 SREG6 SREG7.
ENDINSTR

EXOR - EXCLUSIVE OR

INCE - INCREMENT MEMORY

LOADA - LOAD ACCA FROM MEMORY

ORAA - (ACCA := ACCA .OR. M)

ORAB - (ACCB := ACCB .OR. M)

ROLE - ROTATE LEFT, MEMORY

MORE - ROTATE RIGHT, MEMORY
CALL NZVCB TEMPI DBUS SREG5 SREG6 SREG7 SREG8 1.
MOVE DBUS TEMPI
WRIED ENDINSTR

ASLE - SHIFT LEFT, ARITHMETIC, MEMORY
78 6.0
CALL GETEOP IMM1 ABUS.
SMALL TEMPI DBUS 1
CALL NZVCB TEMPI DBUS SREG5 SREG6 SREG7 SREG8 80+16.
MOVE DBUS TEMPI
WRIED ENDINSTR

ASRE - SHIFT RIGHT, ARITHMETIC, MEMORY
77 6.0
CALL GETEOP IMM1 ABUS.
SRAR TEMPI DBUS 1
CALL NZVCB TEMPI DBUS SREG5 SREG6 SREG7 SREG8 1.
MOVE DBUS TEMPI
WRIED ENDINSTR

LSRE - LOGICAL SHIFT RIGHT, MEMORY
74 6.0
CALL GETEOP IMM1 ABUS.
SRRL TEMPI DBUS 1
CALL NZVCB TEMPI DBUS SREG5 SREG6 SREG7 SREG8 1.
MOVE DBUS TEMPI
WRIED ENDINSTR

TSTE - TEST MEMORY
7D 6.0
CALL GETEOP IMM1 ABUS.
CALL NZIR DBUS SREG5 SREG6 SREG7.
CLEAR SREG8
ENDINSTR

STAAE - STORE ACCA INTO MEMORY
87 6.0
CALL GETEOP IMM1 ABUS.
MOVE DBUS SREG0
WRIED
CALL NZIR DBUS SREG5 SREG6 SREG7.
ENDINSTR

STABE - STORE ACC9 INTO MEMORY
F7 6.0
CALL GETEOP IMM1 ABUS.
MOVE DBUS SREG1
WRIED
CALL NZIR DBUS SREG5 SREG6 SREG7.
ENDINSTR

SBAE - (ACCA := ACCA - M)
80 4.0
CALL GETEOP IMM1 ABUS.
CALL SUBTR TEMPI SREG0 DBUS.
CALL NZVCB TEMPI SREG0 DBUS SREG5 SREG6 SREG7 SREG8.
MOVE SREG0 TEMPI
ENDINSTR

SBBE - (ACCB := ACCB - M)
F0 4.0
CALL GETEOP IMM1 ABUS.
CALL SU3TR TEMP1 SREG1 DBUS.
CALL NZVCS TEMP1 SREG1 DBUS SREG5 SREG6 SREG7 SREG8.
MOVE SREG1 TEMP1.
ENDINSTR
S3CAE = (ACC1=ACCA - M = CARRY)
R2 4.5
CALL GETEOP IMM1 ABUS.
CALL SU3TR TEMP1 SREG3 DBUS.
CALL NZVCS TEMP2 TEMP1 SREG6.
CALL NZVCS TEMP1 SREG5 DBUS SREG6 SREG7 SREG8.
MOVE SREG6 TEMP2.
ENDINSTR
S3CBE = (ACC1=ACCA - M = CARRY)
F2 4.5
CALL GETEOP IMM1 ABUS.
CALL SU3TR TEMP2 SREG1 DBUS.
CALL SU3TR TEMP2 TEMP1 SREG6.
CALL NZVCS TEMP1 SREG6 DBUS SREG5 SREG6 SREG7 SREG8.
MOVE SREG1 TEMP2.
ENDINSTR
C3XE = COMPAR IX WITH MEMORY
BC 6.0
CALL GETEOP IMM1 ABUS.
MOVE TEMP1 ABUS.
INCB ABUS ABUS.
READO.
MOVE TEMP2 ABUS.
CONCAT TEMP3 TEMP1 (8) TEMP2.
CALL SU3TR TEMP4 SREG2 TEMP3.
AND TEMP1 TEMP4 M30+1E.
CLEAR SREG5.
CLEAR SREG7.
IF TEMP1 NE 0.
MOVE SREG5 1.
MOVE SREG7 1.
ENDIF.
CALL ZERO TEMP4 SREG6.
ENDINSTR
L0XE = LOAD IX FROM MEMORY
BE 6.0
CALL GETEOP IMM1 ABUS.
MOVE TEMP1 DBUS.
INCB ABUS ABUS.
READO.
MOVE TEMP2 DBUS.
CONCAT SREG2 TEMP1 (8) TEMP2.
CALL NZVLO SREG2 SREG6 SREG7 SREG8.
ENDINSTR
L3XE = LOAD STACK POINTER FROM MEMORY
BE 6.0
CALL GETEOP IMM1 ABUS.
MOVE TEMP1 DBUS.
INCB ABUS ABUS.
READO.
MOVE TEMP2 DBUS.
CONCAT STACK TEMP1 (8) TEMP2.
CALL NZVLO STACK SREG5 SREG6 SREG7.
ENDINSTR
STXE - STORE IX INTO MEMORY
FF 6.0
CALL STOLOP SREG2 IMM1 ABUS OBUS.
CALL NZVLO SREG2 SREG6 SREG7.
ENDINSTR
SIZE - STORE STACK POINTER INTO MEMORY
3F 6.0
CALL STOLOP STACKP IMM1 ABUS OBUS.
CALL NZVLO STACKP SREG6 SREG7.
ENDINSTR
JMPE - JUMP, EXTENDED
7E 4.0
MOVE PC IMM1
ENDINSTR
JSRE - JUMP TO SUBROUTINE
9D 9.0
MOVE TEMP1 PC
AND TEMP1 TEMP1 FF16
AND TEMP1 TEMP1 FF0016
SHRL TEMP1 TEMP1
PUSH TEMP1
PUSH TEMP1
MOVE PC IMM1
ENDINSTR
ENDINSTEDEF
DEFINE GETEOP TEMP1 TEMP2.
MOVE TEMP2 TEMP1
READ
RETURN
ENDINSTR
ENDCLASS
BIOGRAPHY

Ramon Tan was born on January 25, 1953 to TAN Hing Liong and SIAO Muy Tie in Cebu City, Cebu, Philippines. In 1968 he entered the University of the Philippines at Diliman, Quezon City, Philippines, where he majored in mathematics. He was graduated from the same university in April, 1972 with cum laude honors in the mathematics class. Subsequently, he joined the faculty of mathematics of the De La Salle College in Manila, Philippines, where he taught until May, 1974. From July 1974 to May 1976 he was a graduate student of computer science in the Mathematics Department of Lehigh University, and also Graduate Assistant at the Lehigh University Computing Center. Since June, 1976 he has been programmer/analyst for the Allentown and Sacred Heart Computer Center in Allentown, Pennsylvania.