MOS/LSI microcomputer coding: It involves loaders, assemblers and even compilers. Use these and other tools to store algorithms in the system's memory.

Second of three articles

Engineers who incorporate MOS/LSI microcomputers in their designs face a critical need: conversion of system algorithms into instructions that can be loaded directly into the system's memory.

IC manufacturers are giving more and more attention to this phase of design, generally called coding, with improved tools and techniques to simplify the designer's task.

The basic tools available are these:

- Assemblers.
- Editors.
- Loaders.
- Compilers.
- Microprogramming.

Fig. 1 shows the primary function of the first four tools. In addition hardware or software simulators are available for program testing and error locating.

Assembly language: the most appropriate

An assembly language, the most common for microcomputer programming, has these features: symbolic operation codes; labels that refer to memory locations—instruction or data; and symbolic names for operands, such as registers, condition flip-flops and test conditions of conditional instructions (Fig. 2).

For example, in the Fairchild PPS-25 the instruction[1]

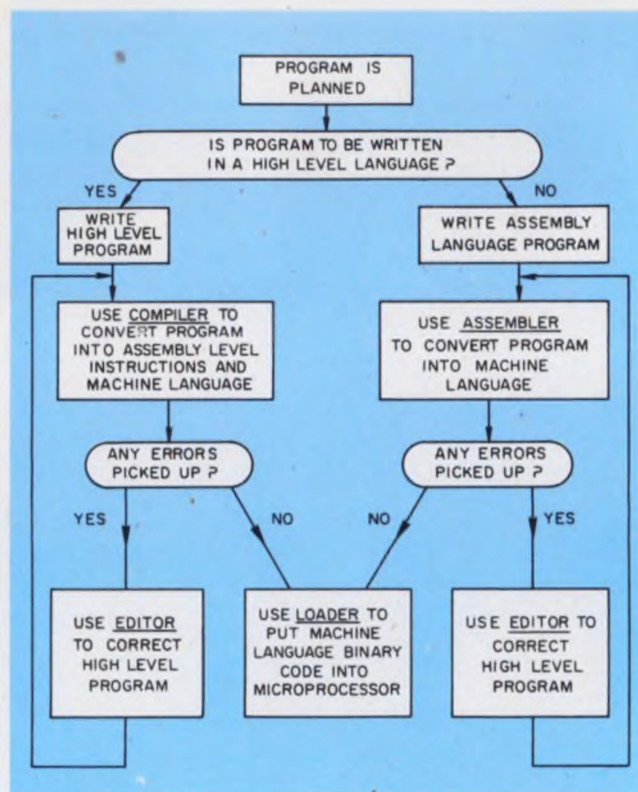
$$(R_{ij}) \leftarrow (A_j) + (R_{kj})$$

replaces the contents of register R_i with the sum of the contents of the accumulator and register R_k. However, only a designated field, j, in each register is involved in the addition. The Fairchild assembly-language equivalent reads

ADD Y, X, T.

Here Y represents the name of a destination register, X the name of a source register and T a previously selected code that represents the field over which addition is to take place. The

C. Dennis Weiss, Ph.D., Member of the Technical Staff, Bell Telephone Laboratories, Holmdel, N.J. 07733.



1. **Preparation of the binary code** to be placed in read-only memory can be simplified by use of a compiler or assembler and an editor and loader.

possible codes of T, with their meanings, include the following:

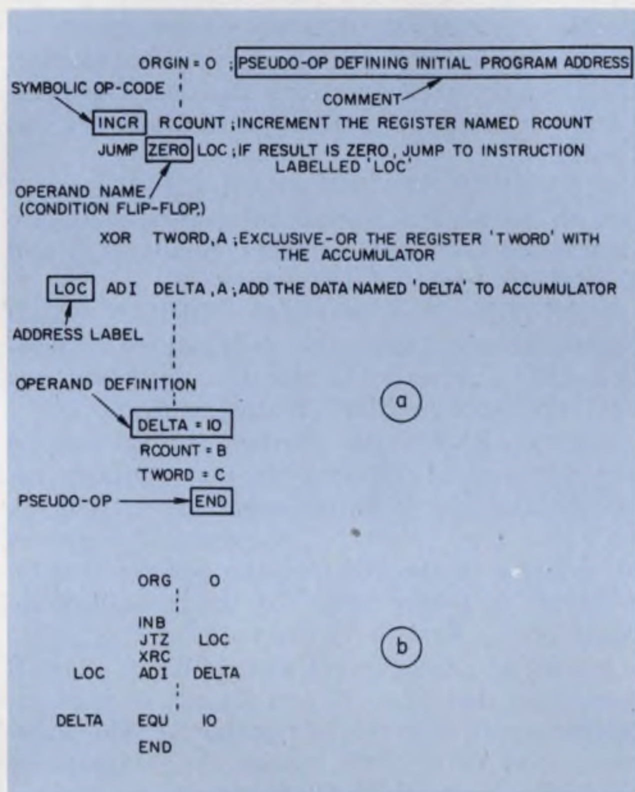
TOTAL:	Total field,
FRAC:	19 (left-most) digit fractional or mantissa field,
LSD:	Least significant digit,
PFIELD:	Digit selected by pointer register.

In the Intel 8008, consider this conditional CALL instruction: PC↓S and (PC) ← 14 bit immediate field, if condition holds; otherwise do next instruction. PC refers to the program counter and S represents a last-in, first-out stack.

Such an instruction in the Intel assembly language is written

CTX PLACE.

X refers to C, Z, P or S, which mean, respective-



2. **Part of an assembly-language program** (a) illustrates the basic language features. The same program segment appears in the Intel 8008 assembly language (b).

A line of source tape

LD ACϕ ,@.+10

This says LOAD register ACϕ indirectly, through the address given by adding 10 (octal) to the current value of the program counter (denoted by·)

A line of the list tape

10 000 9109 LD ACϕ ,@.+10

10 = line number in assembly language program (on source tape)
000 = location of the instruction
9109 = hexadecimal representation of the 16-bit machine language word
LD ACϕ, @ .+10 = assembly language statement written by programmer on source tape

A line of the object tape

1001000100001001

This is a 16-bit machine language equivalent of the instruction above.

3. **The assembler,—a program,—converts a source tape** to a list tape and absolute object tape in this example from the National IMP-16.

ly, Carry = 1, Result Zero, Parity Even and Sign Bit 1. PLACE is the label associated with any other instruction in the sequence being assembled.

Hence the statement

CTP STEP1

causes the microcomputer to call STEP1 conditionally. The processor saves the program counter and replaces it by the address labeled STEP1, if the parity of the register last operated upon was even. Otherwise the instruction that follows would be executed.

The sequence

```
INB
CFP  STEP1
JMP  STEP2
```

increments register B, calls to STEP1 if the parity of register B is odd or performs an unconditional JUMP to STEP2 if the parity is even.

The assembler can read a source tape or file with statements written in the symbolic assembly language (Fig. 3). Also, the assembler can construct various tables from the source file and produce an output object tape, or file, with binary numbers for the microcomputer.

For example, in the Fairchild PPS-25,

ADD B,C, FRAC

appears in the object code as

000100101010.

From left to right, 000 is the operation code for ADD; 100 is the Fairchild code for the B register; 101 is the code for the C register, and 010 represents the mask-programmed code to select the left-most 19-digit field of a register.

```
         NUMBERS OCTAL
         ORIGIN 0
ENTRY 1  LOAD R1, MEM 1
         LOAD R2, MEM 2
* * *    'LOAD' IS UNDEFINED OP-CODE * * *
ENTRY 1  COMPARE R1, R2
* * *    DUPLICATE ADDRESS LABEL * * *
         JCOND PLACE
* * *    'PLACE' is UNDEFINED ADDRESS
         LABEL * * *
* * *    OPERAND MISSING * * *
         JUMP FINISH
         STORE R1, MEM; if R1 > R2, EX-
         CHANGE
* * *    'MEM' UNDEFINED * * *
         STORE R2, MEM 1
FINISH   HLT
* * *    'HLT' IS UNDEFINED OPERATION * * *
MEM 1    = 1732
MEM 2    = 1840
* * *    NUMBER IS INVALID OCTAL * * *
           END
```

```
         NUMBERS OCTAL
         ORIGIN 0
ENTRY 1  LOAD R1, MEM 1
         LOAD R2, MEM 2
ENTRY 2  COMPARE R1, R2
         JCOND GREATER, PLACE
         JUMP FINISH
PLACE    STORE R1, MEM 2; if R1 > R2, EX-
         CHANGE
         STORE R2, MEM 1
FINISH   HALT
MEM 1    = 1732
MEM 2    = 2040
         END
```

4. **An assembler provides error messages** that start with "***" in a program with errors (top). The corrected program appears at the bottom.

For the Intel 8008,

CTP STEP1

appears as the 3-byte instruction,

01111010
00110000
xx001110.

STEP1 is assumed to be an instruction stored at binary location 00111000110000. The last two bytes give, respectively, the low 8 and high 6 bits of the address. The bits marked x are "don't cares" for the 8008. The assembler could substitute any bit pattern, since the machine ignores these locations.

The assembler—a program

The assembler is a program that must be run on some computer. One assembler program—from Intel—can be loaded into several pROM or ROM chips and executed by a microcomputer of the type for which it is assembling. These are called "hardware assemblers," because they run on the hardware itself.

A more common situation is one in which the assembler itself is written in Fortran. With minor modifications, the program can be run on any computer that compiles Fortran programs. Thus the designer prepares source programs, assembling them on some other computer, to obtain the object tape for the microcomputer. The Fortran-written assemblers are often made available to users through various national time-sharing, computer-service companies.

Assemblers contain pseudo-operations

Assemblers provide more sophisticated features. These are usually pseudo-operations, or assembler instructions, that do not assemble into microcomputer instructions directly but control the assembly of instructions that do. The more significant and common psuedo-ops are as follows:

- NUMBER SYSTEM (B,O,D). If B is written, all literals that appear in operand fields are interpreted as binary numbers. Similarly O and D establish octal and decimal modes.
- ORIGIN. The statement ORIGIN 256D causes the next instruction to be stored at location 256 (decimal). Consecutive locations are used until another ORIGIN statement appears.
- COMMENTS. It's common to intersperse English text in a source file that contains assembly language. With the selection of a symbol, such as "/" or ";" or ":", the assembler ignores all symbols to the right of the selected one on each line of source text. But the assembler reproduces the symbols in the final list file.
- EQUAL. A statement such as R1 = PLACE establishes that PLACE, and R1 can be used interchangeably as names of register R1. The statement DATA1 = 53D causes the contents of DATA1 to be taken as 53 (decimal).
- DATA GENERATING STATEMENT. A statement such as TABLE D 7, 53, 29 creates three data words stored in successive locations in memory. The first location is labeled TABLE.

Assemblers give error messages

The ability of assemblers to detect and point to a variety of errors in source statements is one of their most valuable features (Fig. 4). These errors are syntactic—they deal with misuse of the actual language. Assemblers normally cannot catch logic errors in the program, errors of intent or other subtle problems. A statement that contains an error is printed in the list file with a code letter—a flag—beside it. Or the entire error message may be printed.

Some common errors that can be detected include duplicate address label, undefined label and unrecognized instruction mnemonic (due perhaps to the misspelling of an operation code). Other

detectable errors include undefined operand field names, wrong number of operands and an invalid number in the number system chosen. In addition an assembler could be made to detect the error of an address referred to the same ROM page, as in a short JUMP when a long JUMP is required.

Not all errors of syntax are flagged in current microprocessor assemblers. For example, when the labeled address for a JUMP or CALL instruction is not the start of an executable instruction, the error is not generally detected.

A macro facility—a deluxe feature in assemblers—is very useful when similar sections of code are used repeatedly but variations preclude the use of conventional subroutine techniques. A macro consists of a sequence of code or a routine that is defined with such parameters as data values, addresses, labels or even instructions. An expansion of a macro involves a specific copy of this sequence in which all parameters have assigned values.

For the assembler to produce an expansion of the macro, only a single statement need be written—if you assume that the macro definition has already been given to the assembler. This statement appears at the location at which the expansion is to begin, and it contains a list of the values to be assigned. The assembler creates the complete expansion where requested.

Editors make changes

Editors are interactive systems that allow designers to prepare a program, or text, and to make changes with simple commands. Time-sharing services, which provide remote access to microcomputer assemblers, have such editor systems. Hence designers can prepare assembly-language programs and correct them. They can add documentation and store, combine and retrieve programs. And they can output programs onto paper tape and printers with relative ease.

Once a program has been written, assembler-flagged errors corrected and a binary object tape, or file, created, the program must be loaded into the memory of the microcomputer system.

Assembled programs can be loaded into mask or field-programmable ROMs. They can also be loaded into RAMs, in which case a small bootstrap loader is required. The latter may be a minimal program loaded into several ROMs or pROMs. This bootstrap program has just enough capability to read an object tape of a complete loader program, which is placed on a tape reader under microprocessor control. More often, the bootstrap loader contains the entire loader program, and all RAM space is available to load the application program.

Application programs can be conveniently tested in RAM before they are committed to ROMs or pROMs. However, if they are to be used in RAMs in the final system, a startup or restart procedure is needed. The procedure permits bootstrapping of the microcomputer into operation. A permanent loader is required in read-only memory.

A PL/M statement

```
DECLARE (X,Y,Z) BYTE; IF X > Y THEN Z =
   X - Y + 2; ELSE Z = Y - X + 2
```

An equivalent set of assembly language statements for the Intel 8008

```
         ORG 4000
BEGIN    LLI LOW X
         LHI HIGH X
         LAM;          accumulator contains X
         LLI LOW Y
         LHI HIGH Y
         LBM;          B-register contains Y
         SUB;          Subtract B-register from
                       accumulator
         JTS LOC2;     if result negative, jump
                       to LOC2.
LOC 1    ADI 2;        add 2 to accumulator
         LLI LOW Z
         LHI HIGH Z
         LMA;          store answer in the loca-
                       tion for Z
         JMP FINISH
LOC 2    LCI 377
         XRC;          accumulator bits
         ADI 1:        complemented
                       2's complement of X-Y in
                       accumulator
         JMP LOC1
FINISH   HLT
LOW X    EQU 70;       word address of X
HIGH X   EQU 10;       page address of X
LOW Y    EQU 71
HIGH X   EQU 10
LOW Z    EQU 72
HIGH Z   EQU 10
         ORG 4070
LOC X    DEF 0;        X = 0 initially. Value as-
                            signed elsewhere
LOC Y    DEF 0;        Y = 0 initially
LOC Z    DEF 0;        Z = 0 initially
```

5. **A short, readable compiler statement** corresponds to many assembly-language statements.

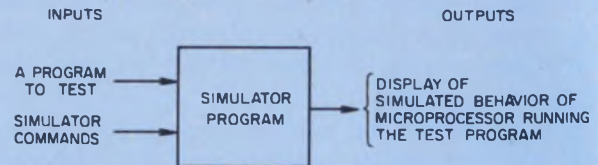
Advanced loader features

The most elementary binary loader simply reads successive words on the object tape and writes them into successive locations of RAM memory. The loader generally starts at a fixed origin. A relocating loader is more complex and not generally available. The reloading loader uses a special object tape and the desired origin data to automatically adjust the program addresses and load the resulting binary instructions.

With a basic binary loader, the same flexibility can be achieved by reassembly of the original source tape or file, but with a change of the ori-

- Errors in basic system design
  - difference between intended or desired operation and that achieved
- Errors in basic algorithms
  - incorrect algorithm
  - wrong strategy
  - algorithm takes too long to execute
  - arithmetic accuracy or precision unsatisfactory
- Errors in implementation
  - logic error
    - off by one count
    - conditions reversed
    - data stored in wrong order
    - microcomputer hangs up in a loop
    - data destroyed by overstore
    - wrong register used
  - coding errors
  - wrong instruction
- Errors in hardware
  - marginal operation
    - races
    - propagation delays too great
  - wiring error
  - interface signals incorrect
  - peripheral device operated improperly

6. **Many potential sources of error** exist in a microcomputer design.



EXAMPLE

INPUT TO SIMULATOR
- The program itself (machine language instructions produced by an assembler)
- A list of simulator commands
  - SET PC = 0, REGISTERS R1 = 1, R2 = 1, R3 = 1
  - SET MEMORY LOCATIONS MEM (100) TO MEM (150) = 0
  - STOP SIMULATION AFTER 20 INSTRUCTIONS
  - DISPLAY REGISTERS R1, R2, R3
  - DISPLAY MEM (100) TO M (103)
  - DISPLAY OCTAL
  - START

OUTPUT OF SIMULATOR
- R1 = 5 R2 = 10 R3 = 73
- MEM (100) = 0
- MEM (101) = 377
- MEM (102) = 264
- MEM (103) = 113

7. **Commands to a simulator** allow designers to verify that a program is correct.

gin using a suitable ORIGIN pseudo-operation.

Another feature of more advanced loaders is linking capability. Here program segments or routines with undefined labels or names can be loaded. The loader supplies missing cross references between the separate routines. Again, this feature can be achieved by reassembly of the entire collection of programs.

Compilers translate languages

A compiler is a program that accepts as input data another program, written in a so-called source language. The compiler then outputs another program, written in what is called the target language. The latter can be either the assembly language or a machine language.

The source language is usually a high-level language, in which the instructions or commands are much more powerful than those of the target language. Examples of source languages are FORTRAN, COBOL, APL, ALGOL or PL/1.

Compilers make the programmer's job easier because they provide a language that requires fewer statements for an algorithm. Compilers eliminate the need to write detailed codes to control loops, to access complex data structures, or to program formulas and functions.

For example, a compiler from Intel has a subset of PL/1 instructions as its source language. The subset language is called PL/M.[2] An example from PL/M illustrates the powerful nature of the source-language instructions:

```
DECLARE (X,Y,Z) BYTE;
IF X > Y THEN Z = X – Y + 2;
ELSE Z = Y – X + 2.
```

The PL/M statements are converted by the compiler into a sequence of assembly-language instructions. The instructions compute Z after they test to see if $X > Y$. If X is bigger, then $Z = X - Y + 2$ is computed. If $X \leq Y$, then $Z = Y - X + 2$ is computed. X, Y and Z refer to the contents of three, single-byte locations established by the DECLARE statement.

Fig. 5 shows an equivalent sequence of instructions written directly in the assembly language of the Intel 8008. Notice how much more difficult the instructions are to understand, despite the comments. And notice the increased amount of writing required, even without comments.

The use of higher-level languages has its limitations. Although errors may be reduced because of the lessened detail, new problems can be caused by failure to understand all the conventions built into the compiler. There is also invariably some loss in efficiency in compiler-generated code.

If you rely too heavily on a compiler, your mode of thinking may be too far removed from the actual microcomputer capabilities. While

programs are compact, easy to read and much easier to write, the net result may be excessive storage space and slower execution.

One solution is to write routines that are typical for an application in both the compiler's source and assembly languages. The comparison helps to determine any loss of efficiency and how significant the loss may be.

A compiler that produces assembly-language code—and not simply machine-language words—permits the use of an assembly listing for tests and verification. Also, such a compiler lets the designer eliminate redundant data movement.

Microprogramming tailors designs

Some microcomputers—the National GPC/P,[3] for example—can be tailored to design requirements through use of a mask-programmed control ROM. In effect, the designer can choose, within limits, the basic machine-language instruction set if he writes the microprogram.

This flexibility simplifies use of a microcomputer as an emulator of another computer. The instruction set of the other computer is microprogrammed into the microcomputer control ROM. Execution of a program instruction corresponds to selection of the equivalent microroutine.

Microprogramming can also be used for critical, short routines in applications where speed is of the essence. The routines can be executed faster when written in the basic control language of the microcomputer. A single machine-language instruction triggers the routine.

The microprogram instructions are more elemental than the usual machine-language instructions. Each instruction controls limited, simple operations in the microcomputer. A sequence of instructions is required for most machine-language instructions. Hence many instructions are required for an entire computational routine.

Simulator tests programs

Many potential sources of error exist in a microcomputer program of even modest complexity (Fig. 6). A software simulator provides one of the most useful tools for testing programs.

Input data to the simulator consist of an assembled program, or object file, written for the microcomputer. In addition various commands are available to control the simulated execution of the program (Fig. 7).

The simulator output contains representations of the contents of various registers, flags and memory locations. These are shown as they would appear inside the microcomputer. The simulator commands allow designers to obtain selected outputs at simulated instants. A listing of simulator commands similar to those for the Intel 4004 and 8008[4,5] appears in Fig. 8.

- Start simulation.
- Stop simulation after a given number of cycles of simulated instructions.
- Stop simulation when the processor reaches a specified instruction or memory location.
- Stop simulation when the contents of a specified memory location are altered.
- Display any registers, flags, program counter, stack contents, I/O ports, or memory locations specified in a command and range-list.
- Trace the simulated microprocessor by displaying elements such as registers whenever an an instruction is fetched from the memory region specified in a range-list.
- Display the number of instruction states used by the microprocessor since the last simulator initialization.
- Set specified memory locations, registers and I/O ports to specific values to initialize a run.
- Interrupt the simulated microprocessor and force a CALL instruction.

8. **A variety of simulator commands** is available to test microcomputer programs.

As with all computer systems, microcomputer program libraries are beginning to form, with contributions from vendors and users. A brief listing of frequently used programs appears in Fig. 9. ■■

- Hardware exercisers
- Test programs for RAMs
- Logic subroutines for microcomputers which do not have basic logic type instructions
- Decimal arithmetic routines
- Transcendental function routines
- Data format conversion routines
- Teletype or tape drive interface programs

9. **Program libraries** contain frequently used programs.

The first article in the series appeared in the April 1 issue, and dealt with microprocessor instruction sets. The concluding article will discuss an application example.

References:

1. "PPS-25, Programmed Processor System Preliminary Users Manual," October 25, 1972, Fairchild Semiconductor, Mountain View, Calif. 94040.
2. "A Guide to PL/M Programming," July, 1973, Intel Corp., Santa Clara, Calif. 95051.
3. "General Purpose Controller/Processor (GPC/P)," Publication No. 4200005A, National Semiconductor, Santa Clara, Calif. 95051.
4. "MCS-4 Microcomputer Set, Users Manual," Revision 4, February, 1973, Intel Corp.
5. "MCS-8, 8008 Simulator Software Package," November, 1972, Intel Corp.