1974-04

SMAL: A Symbolic Microcontroller Assembly Language

Kildall, Gary A.

Appendix 2

SMAL: A Symbolic Microcontroller Assembly Language

Gary A. Kildall
Computer Science Group
Naval Postgraduate School
Monterey, California
April, 1974

I. Introduction

A simple microcontroller has been designed by the Computer Science Group at the Naval Postgraduate School (Brubaker [1]) which can be used to replace many IC's in random logic designs. The microcontroller is intended to be the heart of a particular design, with additional random logic modules at the periphery, as required. The microcontroller performs only simple tests and operations, with no ALU or subroutine mechanisms (these mechanisms are added externally, if required).

Although the microcontroller is discussed in detail in Reference [1], it is briefly reviewed here for completeness. Basically, the microcontroller has 32 "input ports" and 32 "output ports," where each port is a single bit line to external modules, as shown in Figure 1.

An 8-bit data bus is also provided for passing information to external modules. An 8-bit register is also provided for controlling program flow externally. The use of these ports and registers are described in detail in Section V.

The microcontroller instructions are stored in Read-Only-Memory (ROM), where the ROM is divided into 256 byte "pages." The instruction set includes the following simple functions

(a) unconditional branch to a specific address in the range 0-32767

(b) branch on input port true (1) or false (0) to a specific page location (0-255)

(c) Strobe a specific output port and place data on the data bus.

The purpose here is to describe a simple assembly language for writing programs for this microcontroller. The language, called SMAL, is written in PL/M (Intel, [2]), and runs on the Intellec-8 or Intellec-80 developmental system (Intel, [3]).

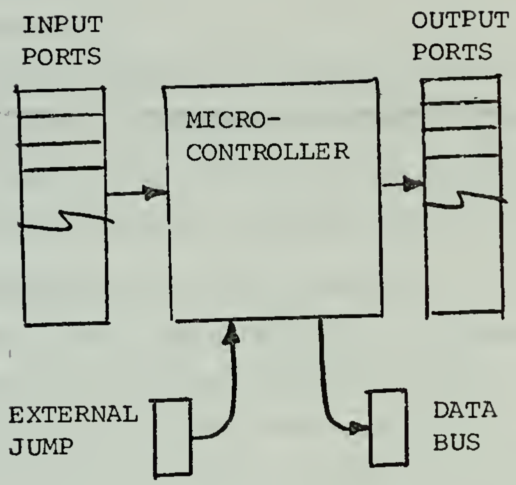

Figure 1. Microcontroller organization.

It is assumed that the reader has a basic familiarity with the micro-controller architecture throughout the discussion which follows. Further, it is assumed the reader is familiar with the Backus notation used to describe language syntax, although the examples used should suffice to present language forms.

II. The SMAL System

As mentioned previously, the SMAL assembler executes on an Intel developmental system. The machine code for SMAL is in the standard hexadecimal format (Intel, [4]), and is loaded with the standard Intel monitor. The SMAL assembler requires approximately 3K of program memory.

The assembler runs in three passes. The first pass performs the label resolution, while the remaining two passes generate Intel hexadecimal tapes for PROM or ROM programming. Two passes are required for tape generation since the microcontroller word size is 16-bits, thus requiring two 8-bit words in parallel for each memory location. The high order bytes are punched on pass-2 and the low order bytes are punched on pass-3.

III. Operating Procedures

After loading the SMAL assembler into the memory of the Intellec, the monitor command

```
G10,
```

is issued to transfer control to the first instruction of the assembler. The assembler responds with

```
#000
```

indicating that it is ready to accept the first SMAL statement, beginning at location 000_{16} in the microcontroller memory. As instructions are typed, the code address is incremented. At any given time, the value

```
#hhh
```

at the start of a line indicates the location where the next instruction is inserted.

Since the assembler requires two more passes on the source program, the user may wish to have the paper tape punch "on" so that subsequent passes can be read through the tape reader. In this case, the line numbers are also punched on the paper tape, but are ignored on subsequent passes, or if the tape is re-run after correction.

The end of the assembly is denoted by the symbols

$$

The assembler will immediately punch a leader of 40 "nulls" and begin the second pass. The source program is re-read, and the high order bytes are punched by the assembler. Each high-order hexadecimal record is preceded by the symbol 'H'.

At the end of pass-2, the assembler again punches a leader and then starts pass-3. Again, a hexadecimal tape is produced; this time the low order bytes are punched, preceded by the symbol 'L'. The assembler halts after pass-3.

The assembler prints a symbol table at the end of the first pass if the assembly is terminated with

$S

instead of '$$'.

As a simple introductory example, the following program checks input port 3 unit it changes to true. On a true input condition, the value on the data bus is changed from 0 to FF_{16}, and output port 5 is strobed. The program repeats this process after input port 4 to changes to false.

```
#000 /SIMPLE EXAMPLE OF A
#000 /MICROCONTROLLER PROGRAM
#000 CHANGE=3; REPEAT=4
#000 BUS=5
#000 /CHANGE REPEAT, AND BUS ARE SYNONYMS
#000 /FOR 3, 4, AND 5, RESPECTIVELY
#000
#000 START, BUS :=0 /SET BUS TO 0
#001        -CHANGE = :* /LOOP UNTIL PORT 3 IS TRUE
#002        BUS :=OFFH /SET BUS TO HEX FF
#003 LOOK, REPEAT =:START /LOOP WHEN 4 IS 0
#004 =: LOOK /OTHERWISE REPEAT THE PROGRAM
#005 $$
```

In general, the symbol '=' is used to assign assembly time values to a symbol, the character ',' is used as a label deliniter, the symbols '=:' and ':=' are used in conditional and unconditional branches, and in port assignments, while the minus symbol '-' denotes a false conditional test and the symbol '=::' denotes an external jump. Note that comments begin with a '/' and end at the next carriage return symbol (denoted here by '↵'). Multiple statements can be placed on a single line with the symbol ';' separating them. In all cases, the ';' is equivalent to a carriage-return. The exact language details are given in sections which follows.

IV. Error Messages

Errors in the assembly language source are flagged with the symbol '%' followed by a bell and a single error character. Note that although these characters are punched on the paper tape if the punch is on, they will be ignored by subsequent passes, or if the tape is completely re-run. The SMAL error characters are:

S - error in statement syntax

X - symbol table overflow*

Ø - error in operand

V - invalid port address

P - off page reference in conditional jump

D - definition error

E - superfluous characters at end of statement

  - undefined symbol (detected at end of assembly); the symbol is printed.

A simple sequential statement editor is built-in to the SMAL assembler to aid correction of paper tapes. This editor is described in detail in a later section..

*Each symbolic name requires n+3 symbol table locations, where n is the length of the name. The symbol table size is changed by altering the value of "symsize" in the SMAL assembler source program. This value is initially set at 200 bytes. There is no restriction on program length.

V. The SMAL Language

The basic tokens of SMAL are discussed first, followed by the syntax for the individual statements. In each case, the syntax is specified using BNF (Backus-Naur Form), the semantic actions are specified informally in English, and examples are given in each case.

A. The tokens of SMAL are similar to those of PL/M for

<identifier> and <number>

That is, an <identifier> is a sequence of up to 32 letters and digits, where the leading character is a letter. A <number> is an integer value in the range 0 to $2^{16}-1$, specified in one of the following bases:

base	base indicator	valid digits
binary	B	0,1
octal	Ø or Q	0,..,7
decimal	D or unspecified	0,...,9
hexadecimal	H	0,..,9,A,B,C,D,E,F

A <number> is a sequence of digits, followed by the base indicator. The leading digit must always be a decimal digit (0 will always suffice for hexadecimal numbers), and must be valid digits for the selected base.

examples

valid <identifier>s are

```
X        INPUT     BUS    REPEAT
X2       X2Y3      LONGSYMBOLNAME
```

invalid <identifier>s are

```
3X       (leading symbol not a letter)
X$Y      (contains a character which is not a letter or a digit)
REALLYLONGSTRINGOFSYMBOLSUSEDFORSYMBOLICNAME
         (symbolic name is too long)
```

valid <number>s are:

```
1    1D    123    80H    OF3H    25Q
11011B    3F5DH    772Ø    772Q    OFFH
```

invalid <number>s are

```
65539    (number exceeds 65535)
FFH      (hexadecimal number requires a leading decimal digit)
823Q     (invalid digits used in an octal number)
```

B. The syntax of a <program> in SMAL is now given.

syntax

```
1.1.  <program> ::= <statement set> <eof>
1.2.  <statement set> ::= <statement>|<statement set> <sep> <statement>
1.3.  <statement> ::= <label field> <basic statement> <comment>
1.4.  <eof> ::= $$|$S
1.5.  <sep> ::= ;|↵
```

semantics

A program is a sequence of <statement>s separated by carriage-returns or semicolons, where the last statement is followed by double dollar signs, or a dollar sign followed by an S. In the latter case, the value of each symbol used in the program is printed.

Although not reflected in the syntax, a control-I (denoted by ↑I) character can be used between the statement elements to "tab" to position across the line. The tab positions are defined as 1, 8, 15, ..., 7n+1... (i.e., every seven columns) across the teletype line. The use of tabs generally reduces the paper tape length since one character is used to represent several blanks.

All statements are "free-field", and thus are not dependent upon particular columns of the teletype line. Note also that Rubout and Line-feed characters are always ignored on input. Thus, paper tapes can be prepared "off-line" for later assembly.

C. The syntax of the statement elements is given below.

syntax

```
2.1.  <label field> ::= <label element>|<label field> <label element>
2.2.  <label element> ::= <identifier>,|
                          <number>,|
                          <empty>
2.3.  <basic statement> ::= <value definition>|
                            <unconditional jump>|
                            <conditional jump>|
                            <output>|
                            <external jump>|
                            <empty>|<literal>
```

2.4. <comment> ::= /<comment string>|<empty>

2.5. <comment string> ::= {a sequence of arbitrary characters, not including ';' ',', '#', or '%'}

2.6. <empty> ::= {the null string of symbols}

semantics

The <label field> is a sequence of zero or more <label elements>, where each <label element> is an identifier or a number. The labels are separated from one another, and from the statement being labelled by the ',' symbol.

If the label is a <number>, then the origin of code generation is set to this value. If multiple <number>s are encountered in a <label field>, code generation begins at the rightmost such value. The value of a numeric label must be in the range 0 to 32767. Note also that the code origin may be set to an area where code was previously generated. In this case, additional output machine code records are produced for this area of memory.

If the label is an <identifier> then two cases must be considered. If the <identifier> has not previously occurred, then the <identifier> takes the value of the current code location (and is subsequently completely synomous with this value). If the <identifier> has occurred previously as a label, or as a defined identifier (see <value definition> below), then the <identifier> already has an associated value. This value is then used in the same manner as a <number> to re-originate code generation at a (possibly) different location.

examples

100H,	code generation begins at $100_{16}=256_{10}$
START,10H,	assuming the location counter is zero upon entry, and START has not previously occured, START takes the value 0, and code generation begins at $10_{16}=16_{10}$.

Again, there are no column dependencies in the <label field>. All labels are identified by the comma which follows. Further, note that the <label field> may be omitted altogether, in which case code generation continues at the next sequential location.

A <comment> can appear following the <basic statement>, and continues to the next semi-colon or carriage-return. All symbols in a <comment> are read but ignored by the assembler. Since a <basic statement> is optionally <empty>, it is possible to write a <comment> as the only entry in the <statement>.

D. The syntax of the <basic statement>s is now presented.

<u>syntax</u>

3.1. <value definition> ::= <identifier> = <right part>

3.2. <unconditional jump> ::= =: <right part>

3.3. <conditional jump> ::= <port reference> =: <right part>

3.4. <port reference> ::= <port value>|-<port value>

3.5. <output> ::= <port value> := <right part>

3.6. <external jump> ::= <external reference> =:: <right part>

3.7. <right part> ::= *|<number>|<identifier>

3.8. <port value> ::= <number>|<identifier>

3.9. <literal> ::= <number>|-<number>|<identifier>| -<identifier>

3.10. <external reference> ::= <number>| <identifier>

<u>semantics</u>

A <value definition> is used to associate a particular number with an <identifier> name. The <identifier> must not appear elsewhere on the left of a <value definition>, nor can it occur previous to this statement as a statement label. If these rules are observed then the <identifier> defined by the <value definition> can be used in place of the numeric result of the <right part>.

The <right part> can be one of three types. If it takes the form '*' then the numeric value of the <right part> is the current code location (after all elements of the <label field> are processed). If the <right part> is a <number>, then the value is simply the number itself, which must be in the range 0 to 32767. If the <right part> is an <identifier> then the value of the <right part> is the value of the <identifier>. That is, the <identifier> must appear elsewhere (before or after) as the left part of a <value definition>, or as a statement label. In this case, the value of the <identifier> is treated in exactly the same manner as a <number>.

examples

```
        X  = 55Q
        Y  = Z1   (Z1 defined elsewhere)
    GAMMA  = *
```

semantics

The <unconditional statement> causes microprocessor program control to transfer to the absolute memory location given by the <right part>. As above, the <right part> must evaluate to a numeric value in the range 0 to 32767.

examples

```
        =: 500H
        =: X    (X defined elsewhere)
        =: *    (infinite loop)
```

semantics

A <conditional jump> is used to conditionally alter program control to a location within the page containing the jump instruction. The value of <port reference> is either a <number> or an <identifier> which evaluates to a number through a <value definition> or labeled statement. The resulting <port value>, however, must always evaluate to a number p in the range 0 through 31. If the <port value> is preceded by a minus sign, then the jump takes place on a 0 value on input line p, otherwise the jump is taken on a 1 value at port p. Program control continues to the next memory location if the condition is not met.

The jump location which is used when the condition is met comes from the value of the <right part>. As above, the <right part> must evaluate to a number k in the range $0 \leq k \leq 32767$. Note, however, that if the value of the code location counter is c after processing all statement labels, then it must be the case that

$$\left\lfloor \frac{c}{256} \right\rfloor = \left\lfloor \frac{k}{256} \right\rfloor$$

(where $\lfloor n \rfloor$ denotes the "integer part" of n). That is, the destination of the conditional jump <u>must be</u> to a program location on the same <u>page</u> as the conditional jump instruction.

examples

```
      5 =: 100H        (jump to 256 if input port 5 is 1)
      X =: 50          (jump to location 50 if the port given by X's
                        value is 1)
    -31 =: *           (jump to this instruction while port 31 is 0)
 -GAMMA =: DELTA       (jump to the location given by DELTA's value if
                        the input port given by GAMMA's value is 0).
```

semantics

The <output> statement probes an output line and loads data on the 8-bit data bus. In this case, the <port value> is similar to the description above (i.e., it must evaluate to a number in the range 0 through 31), but instead designates a particular output line to be strobed. The <right part> must evaluate to an operand that can be placed on the data bus, and thus is restricted to the range 0 through 255.

examples

15 := 5	(place a 5 on the data bus and strobe output line 15)
X := OFFH	(place FF_{16} on the data bus and strobe the output line given by X's value)
XYZ := VAL	(place VAL's value on the data bus, and strobe the line given by XYZ's value).

semantics

In the <external jump> command, which takes the form X =:: Y, an unconditional jump to location Y occurs with the exception that the low order bits of Y are replaced by bits from external source X. From 0 to 8 bits of Y may be replaced. The number and source of the external bits is a function of address multiplexing circuitry added to the micro-controller for particular applications.

In general, the jump external command can be used as an externally selected "case statement." For example, the jump external operation can be used to rapidly interpret an encoded command (e.g. an op code) received from external hardware. Y specifies the base address of a table and the external bits specify the entry into the table. Each entry into the table normally contains an unconditional jump to a routine to handle the particular command represented. Note that the placement of such a jump table is critical. For example, if 4 bits are being replaced, the table

must be located on a word address that is a multiple of 16.

examples

```
        X =:: Y         (rightmost two bits to be obtained from external
     4,Y, =:  Y1         source X)
          =:  Y2
          =:  Y3
          =:  Y4
```

semantics

The <literal> statement allows the programmer to place literal constants into the program storage area. The form <number> evaluates to a constant in the range 0 to 65535. If the <literal> is an <identifier> then the identifier name must appear elsewhere in a <value definition> or as a statement label. In this case, the literal becomes the value associated with the <identifier>. If -<number> or -<identifier> is used, then the value v resulting from the <number> or <identifier> is "inverted." That is, the value which is taken is

65535 - V

Note also that the microcontroller inverts the rightmost five bits of a memory word when it is fetched from memory (Brubaker [1]), and thus the rightmost five bits of the literal are always stored in inverted form in the ROM so that they will come from memory in "true" form when they are eventually fetched.

examples

```
        5
        X
        -OFE32H
        -XYZ
```

Editing Commands

In order to simplify the process of correcting source tapes, a tape editor is included in the SMAL system. All editing commands are entered in the "blind" mode which prevents them from appearing on the output tape. The available commands are:

(ctl)L	-- generates a tape leader of 40 nulls
(ctl)A	-- assemble the remainder of the program
(ctl)A#hhh	-- assemble down to line #hhh
(ctl)Annn	-- assemble nnn lines of source code
(ctl)S#hhh	-- skip down to line #hhh in the source tape
(ctl)Snnn	-- skip nnn lines of code on the source tape
(ctl)P	-- print toggle, turns the printing of the program on and off during assembly

NOTE: (ctl) represents the control key and must be typed at the same time as the first symbol in the command.

For example, if a source tape with an error in line #5E is to be corrected, the following commands would be applicable:

```
(ctl)L↵
(ctl)A#5E↵
(ctl)S1↵
<type correct statement>
(ctl)A↵
```

VI. References

1. Brubaker, R. H. A general purpose microcontroller, Computer Science Group, Internal Document (see enclosure) Naval Postgraduate School, Monterey, California, March, 1974.

2. A Guide to PL/M Programming, Intel Corporation, 3065 Bowers Ave., Santa Clara, California, September, 1973.

3. 8008 8 Bit Parallel Central Processor Unit, Users Manual, Intel Corporation, November, 1973.

4. Intellec 8 Microcomputer System Operator's Manual, Intel Corporation, November, 1973.