BCA / MLDR

E. McCreight
draft of August 30, 1978 11:52 AM
filed under [Ivy]<McCreight>bca.press

BCA (Basic Cross-Assembler) is a general-purpose microcomputer assembler. It is written in Bcpl and runs on the Alto. It reads text files produced by an editor (such as Bravo) and produces listing files suitable for printing (via Empress) and relocatable binary output files. To date, it has been used to assemble code for the following microcomputers: Intel 4004, Intel 4040, Intel 8080, Intel 8748, Intel 8086, Motorola 6800, MOS Technology 6502, Zilog Z-80, Texas Instruments TMS1000. It is about a day's work to customize BCA for a new machine with an 8-bit-wide instruction word that doesn't have very many peculiar features. It is possible, although not straightforward at present, to produce code for 16-bit machines.

BCA produces relocatable binary code in a slight generalization of the Micro binary format. MLDR then combines several such files to produce one file in standard Micro binary format, suitable for Prom programming, downloading via Midas, or what-have-you. Each assembly-language module contains one absolute block (whose name is "$absolute") and can contain an arbitrary number of relocatable blocks. At any point in the assembly the assembler is emitting code into only one of these blocks. However, each of these blocks has its own relative location counter, and one can switch back and forth easily among blocks using the .LOC directive. MLDR must place bytes in the absolute block in the exact addresses specified by BCA, but can place each of the relocatable blocks independent of the others. This implies some restrictions on expressions whose values must be computable at assembly time, because BCA does not know what the final value of a symbol pointing into a relocatable block will be.

BCA is a line-oriented assembler. Only one operation may appear per line, and lines are terminated by carriage-returns. Comments begin at a semicolon and end at the end of the line.

```
; This entire line is a comment

        JMP     ExitIfRed       ; this part of this line is a comment
```

Empty lines can be sprinkled around to improve readability. You can also put in form feeds (control-L) to break up the listing intelligently.
Numeric constants are evaluated in the current radix, except when followed by a "o" or "O" (meaning octal), an "x" or "X" (meaning hexadecimal), or a "." (meaning decimal). A numeric constant that starts with a non-decimal digit must be preceded by a 0.

```
175 ; will be evaluated according to the "current" radix
300o    ; 300 octal
35.  ; 35 decimal
100X    ; 100 hexadecimal
0fx  ; NOT fx, because the scanner would read that as a symbol
```

One can also specify the Ascii code for a character using " or $. Usually the two are the same, but $ employs the same conventions as Bcpl and is slightly more civilized when specifying control characters (*n for carriage return, *l for line feed, *t for tab, *s for space).

```
$A      ; 101 octal, the Ascii code for A
"a      ; 141 octal, the Ascii code for a
$*s  ; 40 octal, the Ascii code for space
"    ; also 40 octal, but it's hard to tell there's a space there
```

A symbol is a string of alphabetic or numeric characters whose first character is alphabetic. In addition to the usual upper- and lower-case Roman alphabet, BCA considers the following characters to be alphabetic:

```
[ ] @ # ~ _ < > { } ’
```

Symbols can be very long (up to 50 characters or so). Two symbols are considered to be the same only if they are identical including capitalization.

```
ShrtSym
thisIsIndeedAVeryVeryVeryLongSymbol
[Garbage]9
@Foo
```

The single-character symbol "." means the value of the active location counter as it was at the beginning of the line.

Labels are symbols followed by a colon and appearing as the first item on a line. They can be the only item on the line, if you like.

```
ThisIsALabel:    NOP

ThisIsAlsoALabel:
    JMP     ExitIfGreen
```

Expressions can be put together using +, -, *, /, & (logical and), % (logical or), | (remainder after division by), and ^ (left-shift; x^y means x left-shifted y bits, where y can be negative to indicate right-shift). Expressions are evaluated from left to right except where parentheses indicate otherwise. Item delimiters (space, comma, tab) are not allowed inside expressions.

```
-1
1+3
Arg+Boo-1*3      ; = (((Arg+Boo)-1)*3)
Arg+(Boo-(1*3))
```

Expressions must generally be evaluable at assembly time. There is one important exception to this rule. If Rel1 and Rel2 are symbols that point into relocatable blocks, or imported symbols, then

```
Rel1+constant
Rel1-constant
Rel1-Rel2+constant
Rel1-Rel2-constant
```

are all legal expressions. In addition, if Rel1 and Rel2 are both symbols defined in this assembly, and if both point into the same relocatable block, then the sub-expression

```
Rel2-Rel1
```

is a constant, and can be used in the same context as any other constant.

You can equate a symbol to an expression using the equals sign. This is frequently done to improve the readability of a program. For example,

```
B = Arg+(Boo-(1*3))
Key = 0c42x      ; a memory-mapped peripheral device

LDA     Key
```

Between labels and comments, an assembly line consists of a sequence of items separated by tabs, spaces, or commas. If the first such item is not an operator symbol, then it is evaluated and the assembler emits a byte of code containing its value. If it is an operator symbol, then an operator-specific routine in the assembler is invoked to process the rest of the line and emit whatever bytes of code are appropriate. When it starts up, BCA only knows a few operator symbols and operator classes.

    .RDX   y

This sets the current radix to 10, evaluates y, and sets the current radix to y.

    .LOC   y

This does one of two things, depending on y. If y is an expression that can be evaluated to a constant, then "$absolute" is made the current assembly block and its location counter is set to y. If y is a symbol that has not previously been defined, then a new relocatable block is set up named y, y becomes the current assembly block, and its relative location counter is set to 0. If y is already the name of a relocatable block, then y again becomes the current assembly block but its previous relative location counter value is left unchanged.

    .BLK   y

This evaluates y and adds it to the location counter of the current assembly block. The most common use of .BLK is to reserve Ram space for variables without specifying any value to be put in that space.

    .END

This terminates the assembly

    .TXT   "Any old string*n"

This emits the string in Bcpl string format: a byte of length followed by the Ascii codes for the string charcters, one per byte.

    .ADR   y

This evaluates y and then emits it in two bytes. Whether the high-order byte appears first or second is determined by the value of the symbol @HIORDFIRST.

    .ADRH y

This evaluates y and emits the high-order byte of the value.

    .ADRL   y

This evaluates y and emits the low-order byte of the value.

    .GET   "filename"

This interrupts the reading of the current file, reads the file filename, and then resumes reading the current file.

    .GETNOLIST   "filename"

This is the same as .GET, except that filename is excluded from the listing (for shorter listings).

    .PREDEFINE   "filename"

This is the same as .GETNOLIST, except that filename is only read during pass 1 (and is therefore not allowed to emit code, among other things). This saves assembly time, and is generally used to read in the customization file.

    .DEF   x,y,z

This defines the symbol x to be an operator symbol of class y with value z. This is the usual mechanism by which the assembler is customized to assemble code for particular machines. For a simple example, the class @immclass is a class that emits the value of its operator symbol followed by the value of its first operand. This class is known to BCA at startup time. The statement

    .def ldai,@immclass,0a9x

would cause a succeeding statement

```
ldai 44x
```

to result in the emission of the two code bytes {a9},{44}.

```
.EXPORT    y,z,...
```

This declares that the symbols y,z,... are defined in this assembly module and are to be available for use by other assembly modules.

```
.IMPORT    y,z,...
```

This declares that the symbols y,z,... may be used in this module but will not be defined here and are intended to refer to identically-named symbols exported from some other module.

```
.SHORT     y,z,...
```

Many machines have two opcodes corresponding to the same mnemonic. One is to be used when the operand lies in page 0; the other is to be used otherwise. y,z,... can be symbols pointing into relocatable blocks, or imported symbols, or the names of relocatable blocks. This is a declaration that those symbols, or all symbols in the block, will eventually lie in page 0, and therefore the page 0 opcodes are appropriate.

BCA is invoked with a line like

```
BCA/switches [customfile.br/C] sourcefile.ext
```

The L or l switch causes a listing to appear under the name sourcefile.ls. The E or e switch causes an error file to appear under the name sourcefile.er. The S or s switch appends a symbol table listing to the listing file. The U or u switch maps all alphabetic characters to upper case. The c or C switch applied to a file name indicates that that file contains Bcpl relocatable binary code to customize BCA for a particular microcomputer.

MLDR is invoked with a line like

```
MLDR   outputfile/f listingfile/l binfile1 binfile2 ...
       octalLocCtr/p block1/m block2/m ...
       hexLocCtr/x block100/m block101/m ...
       dummyBlock/d
       overlaySymbol/o block200/m block201/m ...
```

The binfile’s are the .mb files produced by BCA. The block’s are the exact (including capitalization) names of relocatable blocks contained in the .mb files. The /m switch actually causes the block to be loaded, symbols to be resolved, and the location counter advanced by the length of the block. The /d switch is the same except that the data is not actually loaded. Only the $absolute block is contained in the output file.

Appendix A

Micro binary format

This appendix describes that part of the Micro binary format used by BCA, including the generalization that handles relocation. A Micro binary format file consists of a sequence of items. Each item is a sequence of 16-bit words. Each item has a one-word header that gives its type. The headers used by BCA are:

0 - End of file
1 - Data Word
2 - Set Location Counter
4 - Define Block
5 - Define Symbol

The Define Block header is followed by a one-word block number, a one-word bit width, and an Ascii string, expressed as a sequence of Ascii characters terminated by one or two Null (0) characters as necessary to reach the next word boundary. The block number is used in all subsequent references to the block. The bit width is the width of a word in the block, in bits. Normally this is 8, but we shall see a special case where this is at least 96. If the bit width is 0, this is not really a block, but rather an imported symbol. The string is the name of the block.

The Set Location Counter header is followed by a one-word block number (specified in a previous Define Block item) and a one-word offset in that block.

The Data Word header is followed by one word containing the source line number that generated this data word and then a sequence of bits as wide as the word in the block last specified in a Set Location Counter item. This sequence is left-aligned in an integral number of 16-bit words. The data word is intended for that block at the address currently in the location counter. Afterward, the location counter is assumed to advance by 1.

The Define Symbol header is followed by one word containing the number of a block, another containing an offset within that block, and finally a string as with Define Block.

A special block called "$relocation" has a word at least 16 bits wide, and usually 96 bits wide. The first 16 bits of each word in this block specify a relocation operation to be performed by MLDR. The remaining bits are interpreted according to the value in the first 16 bits. The value 0 in the high-order 12 bits causes the values 1-4 in the low-order 3 bits have the following meanings:

1 - High byte
2 - Low byte
3 - Entire address (use the low-order byte, and the entire value must lie in the range [0, 255])
4 - Any byte (use the low-order byte, and the entire value must lie in the range [-128, 255])

In these four cases, the second 16 bits are interpreted as the number of a target block in which the result of this relocation operation is to go, and the third 16 bits are an offset within that target block. The fourth 16 bits are a numeric constant, and the fifth 16 bits are the number of a block. The final address of this block is added to the numeric constant, one byte is extracted from the result according to the table above and placed in the target block at the target offset.

If bit 12 (or fourth from least significant) of the relocation operation is 0, then after the arithmetic above is performed, the final address of the block number in the sixth 16 bits is to be subtracted from the result before byte extraction.

Appendix B - Customization

There are two standard ways of customizing BCA to produce code for new machines, and most machines use a synthesis of both. The most straightforward method is to set up a text file containing mostly .def statements, and to .predefine that file into every source file. To give you the flavor of this, the following is taken from the beginning of the MCS6502 predefs file:

```
; The following predefinitions are for the MOS Technology
;       MCS 6502

;       E. McCreight
;       last modified June 16, 1978  5:09 PM to add relocation

#Pg0Addr = 0
#HighAddrByte = 1
#LowAddrByte = 2

        .rdx 16

@OPTOFFSET = -8
@HIORDFIRST = 0

;       There are six classes into which you can put
;       opcodes:

;               @noparclass
;               @pcrelclass
;               @immclass
;               @pgzclass
;               @extclass
;               @optclass

        .def brk @noparclass 00
        .def clc @noparclass 018
        .def cld @noparclass 0d8
...
        .def bcc @pcrelclass 090
        .def bcs @pcrelclass 0b0
        .def beq @pcrelclass 0f0
...
        .def adci @immclass 069
        .def andi @immclass 029
        .def cmpi @immclass 0c9
...
        .def adcz @pgzclass 065
        .def andz @pgzclass 025
        .def cmpz @pgzclass 0c5
...
        .def adce @extclass 06d
        .def ande @extclass 02d
        .def cmpe @extclass 0cd
...
        .def adc @optclass 06d
        .def and @optclass 02d
        .def cmp @optclass 0cd
```

There are several noteworthy points above. First, the 6502 is one of a reasonably large class of machines in which the same generic operator (say, adc, or add with carry) can be applied either to an operand in page 0 of memory or to an operand in general position. Two different operation codes apply to these different cases. The programmer might want to be able to force one case or the other, but in general he wants the assembler to choose the right opcode. The assembler must be able to make this decision in its pass 1, because the length of the assembled instruction (and therefore the placement of subsequent instructions) depends on this decision. This is the motivation for the .short/.long operators. The symbol @OPTOFFSET is used by the operator class @optclass to decide how to adjust the opcode if the short form is required.

Second, the 6502 is a machine where 16-bit addresses are elaborated low-order byte first. This makes sense if you think about indexing and how an 8-bit ALU must handle 16-bit numbers. But some machines don't do it that way. The symbol @HIORDFIRST is used by the FULLADROUT routine to decide which way to elaborate addresses. Non-zero means high-order first, 0 means low-order first.

Each operator class is represented by one Bcpl routine that must respond in each of three assembly "passes". In pass 0, the subroutine is responsible for entering its class name in the assembler's symbol table. In pass 1, the subroutine is responsible for deciding how many bytes of code it will generate in pass 2. In pass 2, it is responsible for the code generation. Let us look at a few typical operator class routines.

```
...

and NoParClass(pass, value) = valof

    [ switchon pass into
        [ case 0: AddOpcodeClass("@noparclass", NoParClass)
            endcase

        case 1:   resultis 1      // byte of code

        case 2: ValueGen(value)
            endcase

        default:
        ]

    resultis 0
    ]

..

and PCRelClass(pass, value) = valof

    [ switchon pass into
        [ case 0: AddOpcodeClass("@PCRelClass", PCRelClass)
            endcase

        case 1:   resultis 2      // bytes of code

        case 2:
            [ ValueGen(value)
            let Expr = vec size E/WORDSIZE
            MustHaveKExpressions(1, 1, Expr)
            test Expr>>E.relative eq currentLC
            ifso ValueGen(FORCEVALUE(
                Expr>>E.value-(currentLC>>S.value+1),
                -128, 127, 8, 15))
            ifnot
```

```
                [
                    REPORT("Cannot do PC-rel op to different loc ctr")
                    ValueGen(0)
                ]
            ]
            endcase

        default:
        ]

    resultis 0
    ]

...

and OptClass(pass, value) = valof

    [ switchon pass into
        [ case 0: AddOpcodeClass("@optclass", OptClass)
            endcase

        case 1:
        case 2:
            [ let Expr = vec size E/WORDSIZE
            MustHaveKExpressions(1, 1, Expr)
            let Short = IsShortForm(Expr)
            if pass eq 1 then resultis Short? 2, 3 // bytes of code

            ValueGen(value+(Short?
                GETABSSYM("@OPTOFFSET", 0), 0))
            test Short
            ifso ExprGen(Expr, EntireAddr)
            ifnot    FULLADROUT(Expr)
            ]
            endcase

        default:
        ]

    resultis 0
    ]
```

At initialization time, BCA calls the subroutine OpcodeDefRtn(), which in turn is expected to call each class routine with pass=0 so that the classes define themselves into the symbol table. The /C switch in BCA’s command line causes BCA dynamically to load and link to a Bcpl relocatable binary file (or a concatenation of several) that you supply. This file re-defines any of BCA’s symbols that it also defines, so if it defines the static OpcodeDefRtn, then BCA will call your OpcodeDefRtn instead of its own. BCA knows that you’ve done that, and reclaims the space occupied by its own opcode class routines, so you’ll have to define all of your own routines. The following is the current list of variables and/or subroutines belonging to BCA that you can use and/or redefine:

Subroutines:
OpcodeDefRtn
AddOpcodeClass
ValueGen
ExprGen
MustHaveKExpressions

UpToKExpressions
FORCEVALUE
REPORT
FORCEFIELD
FULLADROUT
GETABSSYM
AddToPC
SKIPDELIMS
EXPRESSION
ISEXPRESSION
GETSTRING
GETSTRINGCHAR
NEWCHAR

Variables:
PCAtBegin
currentLC
CURCHAR