8bc – a B Compiler for the PDP-8

Robert Clausecker ⟨ *fuz@fuz.su*⟩

Zuse Institute Berlin

ABSTRACT

8bc is a mostly feature-complete B-compiler for the PDP-8 minicomputer. In contrast to contemporary B implementations, *8bc* generates native PDP-8 assembly in a single pass. A deferred instruction generator omits unnecessary instructions and feeds information about the current program state back to the code generator, allowing it to make a situation-dependent choice about the code generated for high-level language constructs.

Using this compiler as an example, we explore historical and contemporary approaches to compiler design and discuss the mutual influence of and on available computing hardware with a special focus on the PDP-8 that guided the evolution of modern procedural languages.

12 October 2019

8bc – a B Compiler for the PDP-8

Robert Clausecker ⟨ *fuz@fuz.su*⟩

Zuse Institute Berlin

1. Introduction

The B programming language [2] was designed in 1969 by Ken Thompson and Dennis Ritchie as a simplified version of BCPL [3] for use with the PDP-7 minicomputer [1]. Being the predecessor of C, the B programming language remains an important milestone in the continued evolution of procedural languages from first works by Rutishauser over ALGOL 60 and BCPL to B and C and finally to modern incarnations like Limbo, Go, and Rust.

The goal of this work is to give an overview over this evolution and to outline the constraints and features of historical machines as well as the programming techniques known in the day and how they influenced the design of contemporary procedural languages and their implementations. We contrast this with *8bc,* a rather straightforward implementation of the B language for the PDP-8 minicomputer using modern industry standard tools and a design approach that started to become commonplace in the 1980s.

The design of *8bc* and the PDP-8 target architecture was chosen to highlight the characteristics of the PDP-8, what design concessions had to be made to bring B to it, and how computers evolved to satisfy the demands of procedural languages.

2. Background

2.1. The B language

The B programming language was designed by Ken Thompson and Dennis Ritchie of Bell Labs as a simplified variant of the BCPL language for use on the PDP-7. The simplifications encompassed a removal of all language features programmers could make do without as well as a change from BCPL's wordy ALGOL-like syntax to a syntax built around an extensive use of punctuation, allowing the programmer to write terser programs and the parser to be less complex.

With these simplifications, a self-hosting B compiler was written for the PDP-7. B code is compiled to a form of threaded code operating mainly on a stack of data, conveniently using the same instruction encoding as the PDP-7 [4]. The stack comprises of lvalue/rvalue pairs where the lvalue contains the address of a datum. If the object on the stack is not a true lvalue (e.g. because it is an intermediate value), the lvalue field points to the rvalue field containing the objects rvalue.

While slightly wasteful, the *take-address* operator & and the *dereference* operator * can be implemented easily in this scheme and lvalues do not need to be distinguished from rvalues for most parts. In the B runtime for the PDP-11, this scheme was altered to push rvalue slots to the stack manually whenever needed. While making the compiler more complicated, both memory usage and program performance is improved. *8bc* uses a similar concept born out of the PDP-8's lack of useful addressing modes.

2.1.1. Syntax and features

Being the predecessor of C, the feature set of B is best described by explaining what features where added when B evolved into C. A more detailed description can be found in [2].

- B is a typeless language; all objects are machine words
- structures, unions, enumerations, type definitions, and type qualifiers are absent
- consequently, the declaration syntax is simple and does not mention any of these

- all local variables are automatic; the **static** keyword does not exist
- B does not have scopes
- explicitly declared as **extrn** (sic!) before use; as a popular extension, some compilers did so implicitly when an undeclared name was operand to the `()` operator
- though popular extensions, the **continue** statement and **do-while** loops are absent
- the syntax for combined assignment operators is =op, not op=
- the short-circuit logic operators && and `||` are absent; as a popular extension, many B compilers implemented & and `|` with their semantics when used in a control expression
- the operators ~ and ^ where absent in early versions of B; the token `,` is a separator, not an operator
- as with K&R C, the content of the standard library depends on the system with wildly varying contents

2.2. The PDP-8

The PDP-8 is a family of 12 bit minicomputers introduced by the Digital Equipment Corporation (DEC) in 1965. Being among the cheapest computers on the market while having a large variety of available peripherals an expansion modules, the PDP-8 and enjoyed a wide usage with over 300,000 units sold over the lifespan of the architecture. DEC being originally a supplier of laboratory equipment, intended the PDP-8 to be used as a low-cost computer for controlling and processing data generated by such laboratory equipment. However, its low-price, reliability, and versatility made computers from the PDP-8 series very popular in other applications such as health care, industrial process control, time sharing, and text processing.

The PDP-8's design is an incremental upgrade over the PDP-5 architecture from 1963 with only minor changes, like an improved interrupt handling process and more ways to microcode `OPR` instructions. Peripherals for the original PDP-8 "straight eight" were originally taken from the largely compatible pool of PDP-5 peripherals, but were quickly replaced by newly designed PDP-8 peripherals. Starting with the PDP-8/E (for which *8bc* was designed), many machines of the PDP-8 family were built around the OMNIBUS system bus, enabling the construction of sophisticated expansion cards that would work on any OMNIBUS system. Many such cards were designed for a wide range of applications, greatly contributing to the popularity of the PDP-8.

2.2.1. Memory reference instructions

The PDP-8 has an address space of 4'096 words of 12 bits each. Later models extend the address space to 8 *fields* of 4'096 words each for a total of up to 32'768 words. As extended memory is not used by *8bc*, we do not furthe describe it in this article. Most instructions operate on a 12 bit accumulator AC and a 1 bit L register that is connected to the accumulator's carry-out are provided for operation. It is often useful to think of L and AC as a single 13 bit accumulator L:AC. Being a*von Neumann* machine, the PDP-8' s program is stored in the same 4'096 words of memory as the data. Each instruction comprises a 3 bit operation code followed by a 9 bit operand.

The instructions are:

0	`AND`	and
1	`TAD`	two's complement add
2	`ISZ`	index, skip if zero
3	`DCA`	deposit, clear AC
4	`JMS`	jump to subroutine
5	`JMP`	jump
6	`IOT`	IO transfer
7	`OPR`	operate

The first six instructions are called *memory reference instructions*. Their operand is made of an indirection bit, a page bit, and a 7 bit address. If the page bit is set, the address is completed with the high 5 bits of the program counter, otherwise zeros are used. If the indirection bit is set, that address is used to look up the actual address from memory. As an example, consider the instruction 3245 located at address 1757. The operation code is 3, indicating a `DCA` instruction. The P bit is set, so the address 0045 is completed with

1600 from the program counter, yielding an operand of 1645. The encoded instruction is thus `DCA 1645`, which deposits the content of AC into the memory at address 1645 and then clears AC.

2.2.2. Microcoded instructions

The *operate* instruction `OPR` interpretes its operand as a bitmask of operations to execute. Three groups of operate instructions exist and within each group, an arbitrary set of instructions can be *microcoded* together to execute at once by computing the bitwise or of their operation codes. The first group provides ways to set up the L and AC registers, increments, and rotations.

7200	`CLA`	clear AC
7100	`CLL`	clear L
7040	`CMA`	complement AC
7020	`CML`	complement L
7010	`RAR`	rotate L:AC right
7012	`RTR`	rotate L:AC twice right
7004	`RAL`	rotate L:AC left
7006	`RTL`	rotate L:AC twice left
7002	`BSW`	byte swap AC
7001	`IAC`	increment L:AC

When multiple group 1 instructions are microcoded together, first `CLA` and `CLL` are executed, then `CMA` and `CML`, then `IAC`, and finally the rotate instructions `RAR`, `RTR`, `RAL`, `RTL`, and `BSW`, of which at most one can be microcoded into any given operate instruction. Operate instructions from group 1 are commonly used to aid in implementing complex arithmetic operations. For example, to compute $A \vee B$, one needs to compute $A + B - (A \wedge B)$:

```
CLA      / clear AC
TAD A    / compute 0 + A i.e. load A
AND B    / compute A ∧ B
CMA IAC  / compute ¬(A ∧ B) + 1, i.e. −(A ∧ B)
TAD A    / compute A - (A ∧ B)
TAD B    / compute A + B - (A ∧ B), i.e. A ∨ B
```

Another common purpose is the generation of small immediate constants, removing the need to place a literal into the current or zero page.

Most of operate group 2 provides instructions for conditional execution depending on the content of L:AC. Instead of performing conditional jumps, the next instruction is skipped if the condition holds. The following microcodable instructions are available:

7600	`CLA`	clear AC
7500	`SMA`	skip on minus AC
7440	`SZA`	skip on zero AC
7420	`SNL`	skip on non-zero L
7410	`SKP`	reverse skip condition
7404	`OSR`	or switch registers
7402	`HLT`	halt

Additionally, the mnemonics

7510	`SPA`	skip on positive AC
7450	`SNA`	skip on non-zero AC
7430	`SZL`	skip on zero L

are provided for skip conditions micro-coded with `SKP`. All of these instructions can be microcoded with each other. First,`SMA`, `SZA`, and `SNL` are e xecuted and the next instruction skipped if any of the conditions holds. If`SKP` is microcoded in, the skip condition is flipped. Then, `CLA` is e xecuted and finally `OSR` (sensing the state of the front panel's switches) and `HLT`. With these instructions, unsigned comparisons can be implemented easily. Due to the lack of an overflow flag it is however rather hard to program signed (two's complement) comparison and support for such has been omitted from *8bc* in favour of unsigned comparisons.

A third group of operate instructions exists but remains unused by *8bc*. This group of instructions manipulates the *extended arithmetic element* (EAE), an add-on for the PDP-8/E that provides extra arithmetic instructions. We do not consider it any further in thid document.

2.2.3. Accessing peripherals

The *IO transfer* instruction `IOT` is used to communicate with peripherals. Its operand is split into a 6 bit device number and a 3 bit field each device can interpret as it wants. These three bits are commonly used to either provide 8 device specific operation codes or 3 operation codes that can be *microcoded* as desired. Many devices occupy multiple device numbers with each device number performing a different set of operations. Some rare peripherals even use parts of the device number as additional operation bits.

As an example, the serial communication module is implemented as two devices, *keyboard*, and *teleprinter*. The keyboard responds to device 03 and provides the following microcodable operations:

6031	`KSF`	keyboard skip if flag
6032	`KCC`	keyboard clear and read character
6034	`KRS`	keyboard read static
6036	`KRB`	keyboard read and begin next read

When a character is received through the serial line, an internal flag is set. This flag can be queried for with a waiting loop around Once`KSF`. be reset to allow for the next character to be received and the current character needs to be transferred to AC. This can be done with a`KCC` and a `KRS` instruction microcoded into one, giving `KRB`. This yields the following idiom to read a character from the keyboard:

```
KSF       / skip if character is ready
 JMP .-1  / if not, loop until it is
KRB       / transfer character to AC
```

Likewise, the teleprinter responds to device 04 and provides the following microcodable operations to send characters:

6041	`TSF`	teleprinter skip if flag
6042	`TCF`	teleprinter clear flag
6044	`TPC`	teleprinter print character
6046	`TLS`	teleprinter load and start

The operation is similar to the keyboard. An internal flag is set once a character has been transmitted and must be manually cleared so the transmission of the next character can be detected. This can be done by microcoding `TCF` with `TPC`, a combination for which the mnemonic `TLS` is provided. This yields a common idiom to send a character:

```
TSF       / skip if previous character transmitted
 JMP .-1  / if not, loop until it has been
TLS       / send character in AC
```

3. The design of 8bc

The design of *8bc* was driven by the desire to generate native code for the PDP-8 with acceptable performance while limiting the size, resource consumption, and programming techniques of the compiler to the state of the art in the early 1980's. This way, we can not only give a good picture of how to cope with the quirks and constraints of the PDP-8 but also explore compiler design from a historical perspective.

3.1. Runtime environment and ABI

The *8bc* runtime makes some concessions to deal with the PDP-8's restricted addressing mode, lack of stack and archaic behaviour of the `JSR` instruction. Instead of generating a stack frame, each B function has a dedicated *call frame* that stores a template for the zero page, space for the function's parameters, the function’s automatic variables, and the previous content of the zero page to be restored on return.

3.1.1. Zero page usage

The zero page is special because it is the only page that can be addressed directly. B programs use the zero page as follows:

0000–0007	interrupt handler
0010–0017	indexed registers
0020–0027	runtime registers
0030–0177	scratch registers

As interrupts are unsupported by B, the interrupt handler is a single `HLT` instruction at address 0001. Index register 0010 is used to store one of the factors when the `MUL` routine is called. While the B compiler does not otherwise use the index registers, they are used by the B runtime routines.

Scratch registers much be preserved by the callee, indexed registers need not. The runtime registers are used to store pointers to important B runtime functions and as scratch space for those runtime registers. The runtime registers are used as follows:

0020	pointer to the `ENTER` routine
0021	pointer to the `LEAVE` routine
0022	pointer to the `MUL` routine
0023	pointer to the `DIV` routine
0024	pointer to the `MOD` routine
0025	runtime scratch register
0026	runtime scratch register
0027	runtime scratch register

3.1.2. Function call sequence

A function is called with a `JSR` instruction followed by pointers to the function’s parameters. The number of parameters must match the number of parameters in the function’s definition, the function returns to the first instruction after the arguments.

The call frame looks as follows. The numbers of registers to save, function arguments to copy, and registers to initialise are negated to simplify the `ENTER` and `LEAVE` runtime routines.

negated number of registers to save
space to save the registers
negated number of parameters
function parameters
negated number of register templates
register templates
space for automatic variables

The first instruction of every B function calls `ENTER`, a runtime function responsible for setting up the environment such that the function can do its job. To return, the B function calls `LEAVE`, a runtime function that restores the zero page to its previous state and then returns from the function that called it.

The `ENTER` routine first copies all zero page registers that are going to be used into the call frame. Then, the arguments are grabbed from the call site and copied into the call frame. The return address is adjusted to skip over them. Lastly, the register template is copied to the zero page. The `LEAVE` routine is simpler: it copies the saved registers back into the zero page and returns to the caller.

3.2. Program structure

Like modern and historical C compilers, *8bc* is split into a compiler driver *8bc* that passes the source file through compiler and assembler, interpretes options, prepends the B runtime *brt.pal*, and finally deletes intermediate files, and an actual compiler*8bc1* that translates B source into P AL assembly. This compiler is a one pass compiler written in C using **lex**(1) and **yacc**(1) to generate lexer and parser. Contrary to historical

B and C compilers (but not compilers for other languages such as Pascal), no intermediate representation of the source code is used. Instead, code is generated at each parser action. This makes for a very memory and time efficient design, but greatly restricts the amounts of optimisations possible.

Apart from a few global variables, the majority of the state remembered by the compiler is found in a *definition table* for variables and functions defined at the top level and a *declaration table* for names declared within a function. While the declaration table is vital for the compiler to find out about the storage class about local names, the definition table is only needed due to shortcomings of the PAL assembler: as it is limited to symbols of up to 7 alphanumeric characters, we cannot always use B names as symbol names. Instead, the B compiler translates all names to numbered labels with the association between name and number being kept in the declaration and definition tables.

3.3. Character set

8bc compiles B source files written in ASCII. To allow for source files to be composed on a real PDP-8 using an ASR 33 teletype, a 6 bit ASCII representation is used internally, mapping ASCII codes 0140–0176 to 0100–0140, yielding the character set:

normal								**alternative**							
	!	"	#	$	%	&	'								
(	)	*	+	,	-	.	/								
0	1	2	3	4	5	6	7								
8	9	:	;	<	=	>	?								
@	A	B	C	D	E	F	G	`	a	b	c	d	e	f	g
H	I	J	K	L	M	N	O	h	i	j	k	l	m	n	o
P	Q	R	S	T	U	V	W	p	q	r	s	t	u	v	w
X	Y	Z	[	\	]	^	_	x	y	z	{	|	}	~	

The alternative characters are treated equally to the normal characters except inside character or string literals. Names and keywords are case insensitive. There is no alternative character for the _ character. For example, the program

```
main() {
    extrn putchar;
    auto i 0, hello "Hello, World!*n";

    while (hello[i] != '*e')
        putchar(hello[i++]);
}
```

could equally be written as

```
MAIN() [
    EXTRN PUTCHAR;
    AUTO I 0, HELLO "Hello, World!*N";

    WHILE (HELLO[I] != '*E')
        PUTCHAR(HELLO[I++]);
]
```

While case is honoured inside string and character literals, the ASR 33 teletype is unable to read or print ASCII characters from the *alternative* characters list and prints the corrsponding *normal* characters instead. To provide UNIX-like semantics, the *getchar*() function translates CR to LF and clears the parity bit; the function *putchar*() inserts a CR before each LF.

3.4. Storage classes

8bc recognises 7 *storage classes*. The storage class is used by the code generator to figure out how to refer to an object. Each storage class exists as an *lvalue* and as an *rvalue* class. The difference is that the rvalue storage class has an additional level of indirection. For example, an object of storage class `RLABEL` is the value of a label. If we dereference an `RLABEL`, we get an object of storage class `LLABEL` which is the object located at that label. The lvalue storage classes are:

0	`LCONST`	object at absolute address
1	`LVALUE`	object pointed to by zero page register
2	`LLABEL`	object pointed to by label
3	`LDATA`	object in data area
4	`LSTACK`	object pointed to by stack register
5	`LAUTO`	object in automatic variable area
6	`LPARAM`	object in parameter area

The storage class `RCONST` is used for constants. *Stack register* refers to a register in the zero page used to spill temporary values. The first stack register follows the last register loaded from the zero page template. Since the size of the zero page template is only known after the function has been compiled, the compiler refers to stack registers through an offset from a label referring to the first stack register, necessitating a separate storage class.

A B object is converted from lvalue to rvalue through the & (take address) operator and back through the * (dereference) operator. When an object that is not of class`LVALUE`, `LSTACK`, `RVALUE`, or `RSTACK` is used as an operand to a memory instruction (one of `AND`, `TAD`, `ISZ`, `DCA`, `JMS`, or `JMP`), the object is *spilled* by templating a zero page register with the object's rvalue and substituting an object of type `LVALUE` or `RVALUE` referring to said zero page register to render the original object accessible. Objects of the various storage classes are otherwise created as follows:

`RCONST`	numerical or character constant
`LLABEL`	external variable, function, or label
`LDATA`	string constant
`RSTACK`	value of an expression
`LAUTO`	automatic variable
`LPARAM`	function parameter

3.5. Optimisations

8bc is an optimising compiler. Even though the lack of an intermediate code representation makes many optimisations hard to perform, peephole optimisations are still possible. To implement these optimisations, the compiler uses three layers of abstraction in code generation:

In the *parser* layer, each production rule's action generates an instruction sequence that pops the operands to the implemented operator from a virtual stack, computes the result, and pushes that result onto the virtual stack. For example, a parser action for the + operator could be:

```
expr = expr '+' expr {
        lda(&$3);
        pop(&$3);
        tad(&$1);
        pop(&$1);
        push(&$$);
}
```

The function *lda*() requests for its argument to be loaded into AC and *tad*() requests for a `TAD` instruction to be generated, implementing the behaviour of the + operator. The function*push*() allocates a ne w memory cell on the virtual stack and writes AC to it, leaving its contents undefined. After loading a datum from the stack, *pop*() is used to mark the top of the virtual stack as unused. Care must be taken to only pop the top element off the stack. This is ensured by always popping operands from right to left.

Most parser actions are a bit more complicated than this example and provide multiple instruction sequences for different situations, e.g. special casing constant operands.

These functions *lda*(), *and*(), *tad*(), *isz*(), *dca*(), *jms*(), *jmp*(), and *opr*() are implemented in the *stack management* module to request the generation of the equivalent instructions.[1] The module watches the contents of AC and eliminates stack allocations that can be satisfied by a constant or existing memory location, turning the virtual stack into stack registers. When a newly allocated stack register is immediately loaded back into AC and popped, the entire *push*(); *lda*(); *pop*(); sequence is discarded, generating no code at all. Some strength reductions are performed as well.

Except for `JMS` instructions (which are directly emitted), each instruction is then passed into the *instruction selection* state machine. The state machine simulates the effect of the requested instructions to the extent possible and defers all instructions whose effect is known at compile time until their effect can no longer be simulated.

The state machine keeps track of the contents of AC and L and continuously replaces the deferred instructions with the shortest sequence of instructions needed to achieve the same effect; sequences that compute constants are replaced by sequences of up to two `OPR` and `TAD` instructions, statically known skips are eliminated, and skips setting AC to 0 or 1 followed by `SZA` or are merged into one.

Summarised, the following optimisations are performed:

3.5.1. Strategy Selection

An operation is translated into a sequence of instructions depending on which operands are constant, on the stack, or already in AC. For example, a subtraction normally generates the sequence

```
expr = expr '-' expr {
        lda(&$3);
        pop(&$3);
        opr(CMA | IAC);
        tad(&$1);
        pop(&$1);
        push(&$$);
}
```

which adds the minuend to the two’s complement of the subtrahend. If the subtrahend is known to be a constant and the minuend is known to already be in AC, the sequence

```
expr = expr '-' expr {
        lda(&$1);
        pop(&$1);
        $3.value = RCONST | −val($3.value) & 07777;
        tad(&$3);
        push(&$$);
}
```

is emitted instead, adding the two’s complement of the subtrahend to the minuend already in AC, saving the minuend from begin deposited on the stack and then reloaded.

3.5.2. Stack forwarding

When the content of AC is known to be a constant value or the result of loading another value, a call to *push*() does not allocate a new stack register but instead returns whatever is currently in AC. This eliminates useless stack registers and paves the way for constant folding.

3.5.3. Reload elimination

When the content of AC is pushed to the stack and then immediately loaded into AC and popped, the entire *push*(); *lda*(); *pop*(); sequence is discarded, leaving the contents of AC untouched. This eliminates all

[1] `IOT` instructions are never requested and not implemented.

unnecessary stack operations during expression evaluation that are not already caught by stack forwarding.

3.5.4. Double load elimination

When AC is known to contain the content of a memory location and a load from that same location is requested, the duplicate load is discarded. The same optimisation is performed for constants through the constant folding optimisation.

3.5.5. Strength Reduction

Instructions which have no effect or can be replaced with `OPR` instructions are discarded or replaced. For example, a *tad*() call that attempts to add 1 to AC is replaced with an `IAC` instruction.

3.5.6. Constant folding

Sequences of instructions resulting in a constant value in AC are deferred. The entire sequence is then replaced by one or two instructions loading the desired value into AC. If possible,`OPR` instructions are used to reduce the size of the register template.

3.5.7. Skip elimination

Skip instructions that can be predicted at compile time are discarded. If the instruction is known to skip, the skipped instruction is discarded as well.

3.5.8. Skip forwarding

A skip instruction that clears AC and is followed by `IAC` is recognised as setting AC to the result of the condition. If such a sequence is followed by a `SZA` or `SNA` microcoded with `CLA`, the two skip instructions are merged into one and the `IAC` is discarded.

3.6. Restrictions

Recursion is not supported. Due to time constraints, the **switch** statement was left out of the implementation. Implementations for the / and % operators are missing in*brt.pal*, b ut can easily be added. Many common B extensions such as **do-while** loops, the **continue** statement, or implementations of & and | with short-circuit behaviour for control expressions were omitted. Redefinitions and use of undefined functions or variables are not detected by the compiler but will lead to failure during assembly.

8bc directly generates a complete PAL program by concatenating the B runtime *brt.pal* and the compiler output. This runtime contains a rudimentary standard library comprising the functions *exit*(), *getchar*(), *putchar*(), and *sense*(). No further library functions are provided. It is not possible to link two or more B source files into a single binary and there is no way to write parts of the program in another language.

4. Literature

1 RITCHIE, Dennis M. *The development of the C language.* ACM SIGPLAN Notices, 1993, vol 28, no 3, p 201–208.

2 THOMPSON, Ken. *Users' Reference to B.* Bell Laboratories, 1972, MM-72-1271-1.

3 RICHARDS, Martin. *The BCPL Reference Manual.* Multics Repository, 1967, Memorandum M-352.

4 PAPENHOFF, Angelo. *The B Programming Language.* http://squoze.net/B, 2018.