BALAD

(Beginners Assembly Language and Debugger)

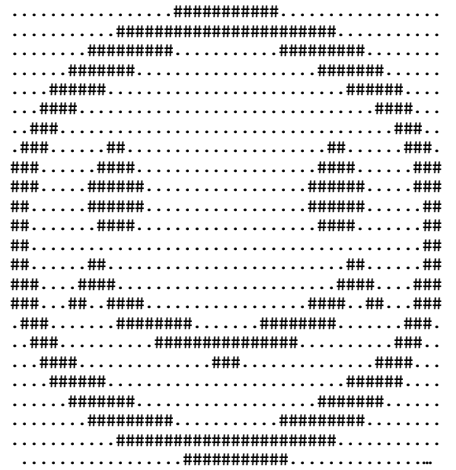
1 INTRODUCTION

BALAD is an assembly-level programming language for an emulated virtual computer with a 15-bit word length. It is combined with a comprehensive debugging system that allows online program assembly, execution of programs and the insertion of breakpoints to allow suspension of programs during execution.

Instructions are machine-oriented, using integer and logical operations only. As a concession to the beginner, extra input/output instructions are provided for automatic Decimal to Binary conversion and the printing of Text Strings. These facilities will enable students to obtain a reasonable printout of their results quickly while concentrating their efforts on developing algorithms.

https://github.com/JohnWulff/balad

`$Id: BALAD.odt 1.8  2021/09/26`



generated with the BALAD program smiley.bl

Table of Contents

2 THE BALAD COMPUTER

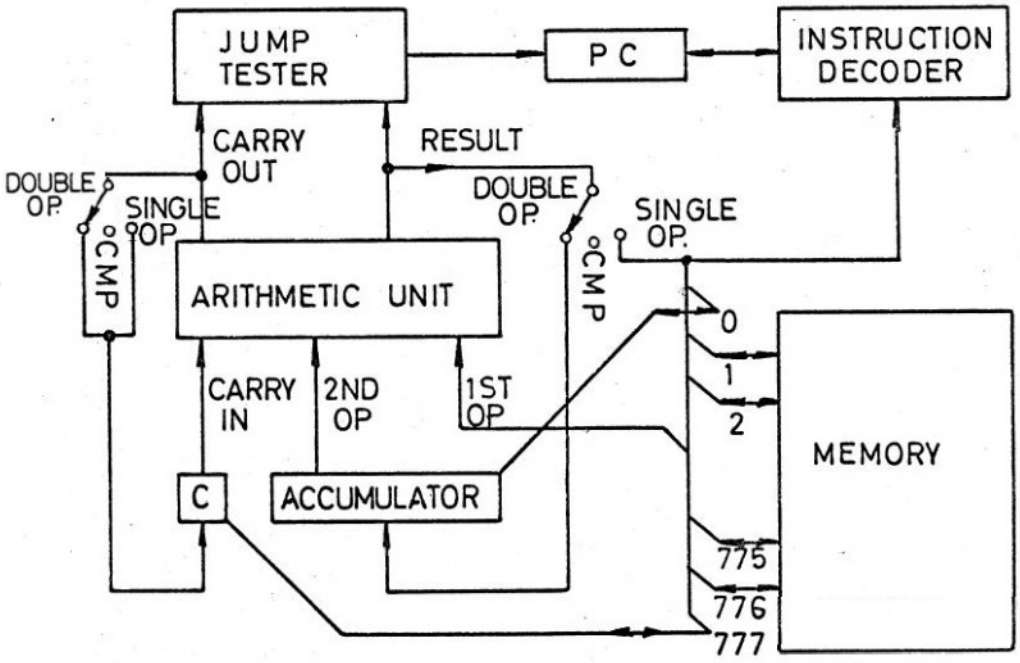


Fig. 1.

The BALAD computer is a one accumulator machine with an arithmetic unit for performing logical operations and 15 bit two's complement arithmetic. The Memory has 512 15 bit words with addresses 0 to 777 (octal numbering). Memory words may contain instructions, addresses of other memory words or data.

At the start of every instruction cycle, the address in the Program Counter register (PC) is used by the computer to fetch an instruction from memory into the Instruction Decoder. The Program Counter is normally incremented by 1 at the end of an instruction cycle so that the next instruction is fetched from the next location in Memory. Only a jump instruction can break this sequence by loading the Program Counter with a new address which points to some arbitrary point in the memory from where the next instruction will then be fetched.

The Instruction Decoder isolates bits 1 to 5 of the instruction and uses this as a number to distinguish between one of the 32 possible instructions. The instruction that has been identified is then executed. In a hardwired computer, this process involves setting various switches to allow data to flow from various registers to other registers.

For illustration two such switches are shown in the schematic above, which guide the result and carry values of an operation from the arithmetic unit either to the Accumulator and Carry for double operand instructions, ignore them for *compare* instructions or back to Memory and Carry for single operand instructions. The last 10 bits of the instruction are used to determine a memory reference address. This can again be thought of as setting a big switch that connects one data path to one of 512 words in memory.

2.1 THE INSTRUCTION FORMAT

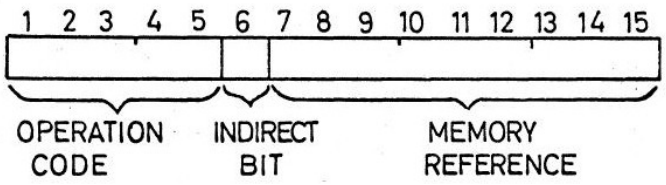


Fig. 2.

Instructions are stored in 15-bit memory registers. Bits 1 to 5 are the operation code, which selects 32 possible operations. Bit 6 is the Indirect bit. If it is zero (0) the value in bits 7 to 15 is taken as the address of the operation of the instruction. If the Indirect bit is a one (1), the address in bits 7 to 15 is used to locate another word whose value in bits 7-15 is used as the address of the first operand of the instruction. Before this is done the Indirect bit of this new word is first checked, and if it is a one (1) the process is repeated. Normally only one level of indirect addressing is used. If more than four levels of indirect address are attempted the computer will stop. This is to break up infinite indirect addressing chains which can happen accidentally if an indirect reference points to itself.

Of the 32 instructions, 4 are double operand arithmetic or logical operations, 2 are data moving operations, 10 are single operand operations, 8 are conditional or unconditional jump operations and the remaining 8 are Input/Output operations. Each instruction has a memory reference part, which addresses a word in memory.

2.1.1 Double Operand Operations

For double operand operations, the 1st operand is taken from the addressed memory location, while the second operand is the Accumulator.

The result of the operation is stored in the Accumulator and the Jump Tester as described later. One exception is the *compare* (CMP) operation for which the result is only stored in the Jump Tester. The result in the Jump Tester is used by conditional jump instructions which follow an operation.

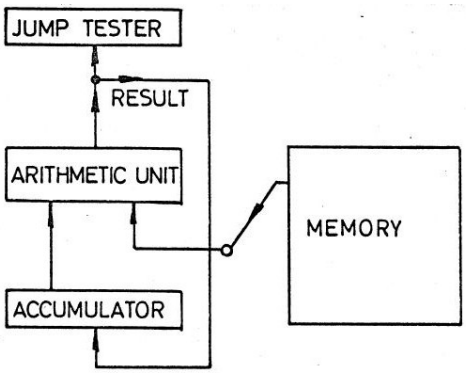


Fig. 3.

2.1.2 Single operand operations

For single operand operations, the one operand is taken from the addressed memory location and the result is stored back in that same location. The result is also stored in the Jump Tester.

Six different conditions of the result in the Jump Tester can be tested and if the condition is True a jump to the memory reference address in the jump instruction is executed. If False, the next instruction is executed. Note that the result used in the test is the result of the last operation executed before the test in the following jump instruction. It does not matter whether this result was also stored in the Accumulator, a memory location, or not stored at all, as in a CMP or TST instructions.

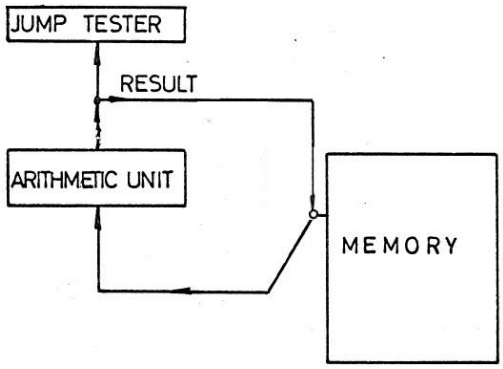


Fig. 4.

Most arithmetic operations are characterized by the fact that information is carried from one-bit position to the next. All computers have an arithmetic unit of limited length (15 bits in this machine). Therefore arithmetic operations will sometimes overflow. So that this case can be catered for, the output of the most significant bit (bit 1) is available to the programmer. The arithmetic unit is extended by one bit to 16 bits, and for practical purposes, the 1st operand is also extended by 1 bit. Storage for this bit is in the "Carry" register. The value of the Carry register also serves as input for most arithmetic operations, and the output of the "Carry" position of the arithmetic unit is stored back in the Carry register for both single and double operand operations. Only for the *compare* operation (CMP) and the *test* operation (TST) is the new Carry not stored. The output of the Carry position is also stored in the jump tester for all instructions, including CMP and TST for subsequent conditional jump tests for the state of "Carry".

For programmer convenience, the Accumulator and the Carry register are both addressable by a memory reference. This means that all the single operand operations can also be carried out on the Accumulator and Carry. Since Carry is only a one-bit register some of the operations may not be very significant. The Accumulator has memory address 0 and the predefined label **ACC**; the Carry register has address 777, the highest address and the predefined label **C**.

2.2 THE INSTRUCTIONS

2.2.1 Double Operand Operations

AND MR with Accumulator (OP CODE 20)

The logical **and** function of each bit of the contents of the Memory Reference and each bit of the Accumulator is stored in the Accumulator and the Jump Tester. Carry is not affected.

ADD MR to Accumulator (OP CODE 22)

Add the contents of the Memory Reference and Carry to the Accumulator and store the Result in the Accumulator. If the unsigned sum is $> 2^{15}$, set the Carry register, else reset it. Accumulator and Carry are also stored in the Jump Tester.

SUB MR from Accumulator (OP CODE 24

Subtract by adding the two's complement of the contents of the Memory Reference and Carry to the Accumulator and store the Result and Carry-out in the Accumulator, Carry and the Jump Tester.

CMP Accumulator with MR (OP CODE 26)

The same operation as SUB except that Carry-in is set to *zero* and the Result and Carry-out are only stored in the Jump Tester. Accumulator, Carry and Memory Reference are left unmodified.

2.2.2 Move data instructions

LDA Load Accumulator from MR. (OP CODE 30)

Load the contents of the Memory Reference into the Accumulator and the Jump Tester. The contents of the Memory Reference and Carry are unaffected. The original contents of the Accumulator are lost.

STA Store Accumulator at MR. (OP CODE 32)

Store the contents of the Accumulator at the Memory Reference location and the Jump Tester. The contents of the Accumulator and Carry are unaffected. The original MR contents are lost.

2.2.3 Single Operand Instructions

CLR Clear MR (OP CODE 34)

Zero is stored in the Memory Reference location and the Jump Tester. Carry is not affected.

TST Test MR (OP CODE 36)

Move the contents of the memory reference to the jump tester without changing the MR. Carry is not affected.

COM Complement MR (OP CODE 40)

Store the logical complement of the contents of the Memory Reference in the Memory Reference location and the Jump Tester. Carry is not affected. The logical complement is also the bitwise complement or the One’s complement.

NEG Negate MR (OP CODE 42)

Store the Two's complement of the contents of the Memory Reference in the Memory Reference location and the Jump Tester. If the original contents were zero, complement Carry.

INC Increment MR (OP CODE 44)

Add 1 to the contents of the Memory Reference and store the result at the Memory Reference location and the Jump Tester. If the original contents were 2^{15}-1 (signed -1), complement Carry.

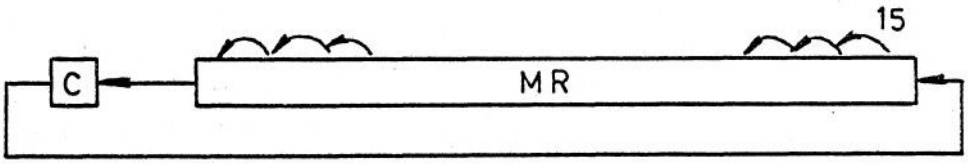
DEC Decrement MR (OP CODE 46)

Subtract 1 from the contents of the Memory Reference and store the result at the Memory Reference location and the Jump Tester. If the original contents were 0, complement Carry.

2.2.4 Rotate and shift instructions

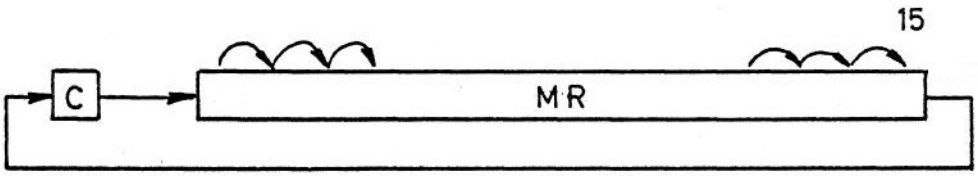
These are single operand operations for bit manipulation, scaling data by factors of 2 and byte manipulation. Rotates are used for testing sequential bits of a word. In each, the Carry register and the contents of the Memory Reference are manipulated in different ways, and the result is stored in the Memory reference location, Carry and the Jump Tester.

ROL Rotate left MR and Carry (OP CODE 50) Fig. 5.



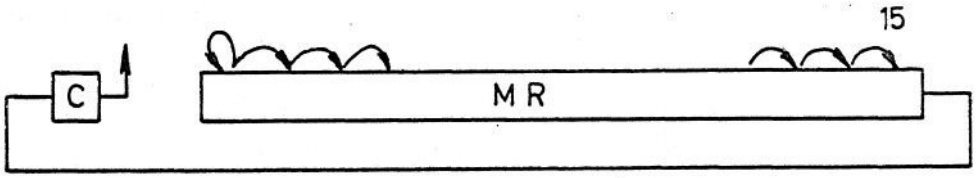
Rotate the contents of the Memory Reference and the Carry Register one bit left as shown in Fig. 5.

ROR Rotate Right MR and Carry (OP CODE 52) Fig. 6.



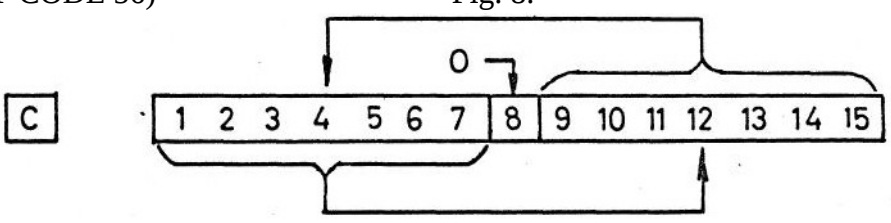
Rotate the contents of the Memory Reference and the Carry Register one bit right as shown in Fig. 6.

ASR Arithmetic Shift Right MR and Carry. (OP CODE 54) Fig. 7.



The sign bit (bit 1) is replicated and also shifted right. All other bits are also shifted right and bit 15 is shifted into Carry. The old value of Carry is lost.

SWP Swap Bytes in MR. (OP CODE 56) Fig. 8.



Bits 1 to 7 and bits 9 to 15 of the contents of the Memory Reference are Swapped. Bit 8 is set to zero. Carry is not affected. The process is illustrated in Fig. 8. A 7-bit BYTE is normally used to store a 7 bit ASCII character. The **SWP** operation allows 2 7 bit ASCII characters to be stored and retrieved from one word. NOTE: no 8 bit extended ASCII characters can be used in BALAD.

2.2.5 Jump Instructions

These instructions allow the alteration of the normal program sequence by jumping to an arbitrary location in memory. Conditional jump instructions also test the copy of the last result or Carry stored in the Jump Tester. If the condition tested is *True*, a jump is performed If the condition is *False*, the next instruction in the word sequence is executed. No registers, except the Program Counter, are modified in the operations.

A special case is a jump to location 0, which is also the Accumulator. This instruction is interpreted as a *HALT* (**HLT**) instruction and execution of a program stops and control is returned to the calling OS or the Debugging System. (A conditional halt can be implemented by making the memory reference of one of the conditional jump instructions 0 although this is deprecated).

JMP Unconditional Jump. (OP CODE 00)

Load the Memory Reference of the instruction (not its contents, unless the memory reference is indirect) into the Program Counter (PC). This has the effect, that the next instruction is fetched from the new location now pointed to by the PC

JZR Jump if Zero Result. (OP CODE 04)
JEQ Jump if equal. jump if **ACC == MR** following a **CMP** instruction

Load MR into PC if the last Result stored in the jump tester was zero. Otherwise, execute the next instruction.

JNR Jump if Non-Zero Result. (OP CODE 06)
JNE Jump if not equal. jump if **ACC != MR** following a **CMP** instruction

Load MR into PC if the last Result stored in the jump tester was non-zero. Otherwise, execute the next instruction.

JZC Jump if Zero Carry. (OP CODE 10)
JLT Jump if less than. jump if **ACC < MR** following a **CMP** instruction

Load MR into PC if the last Carry stored in the Jump Tester was *Zero*. Otherwise, execute the next instruction.

JNC Jump if Non-Zero Carry. (OP CODE 12)
JGE Jump if greater than or equal. jump if **ACC >= MR** following a **CMP** instruction

Load MR into PC if the last Carry stored in the Jump Tester was *One*. Otherwise, execute the next instruction.

JEZ Jump if ether zero. (OP CODE 14)
JLE Jump if less than or equal. jump if **ACC <= MR** following a **CMP** instruction

Load MR into PC if either the Result or the Carry stored in the Jump Tester were *Zero*. Otherwise, execute the next instruction.

JBN Jump if both non-zero (OP CODE 16)
JGT Jump if greater than. jump if **ACC > MR** following a **CMP** instruction

Load MR into PC if both the Result and the Carry stored in the Jump Tester were *Non-zero*. Otherwise, execute the next instruction.

JMS Jump to Subroutine. (OP CODE 02)

Load the contents of the incremented Program Counter (PC) into the Memory Reference location. This is the address of the next instruction in the normal program sequence. Then load **`MR + 1`**. (not its contents) into the PC. Thus a jump has been made to the location **`MR + 1`**.

Subroutines are written in this system with the first location free to store the "Return Address" (PC+1 as above). The first instruction in the subroutine follows this location. To return from a subroutine, an indirect jump is made via the first location of the subroutine. Then control is transferred to the location following the one from which the call was made. Indirect Memory references are written in the assembler language by preceding a location number by the symbol "**`@`**" e.g. **`JMP @400`**, which is jump indirect contents of location 400 for a subroutine at LOC 400.

2.2.6 Input instructions

KDN Key Decimal Number to MR (OP CODE 60)

When this instruction is executed the string Enter a short number: is output to indicate to the operator that a single-precision decimal number is to be typed on the keyboard. The first character may be "+" or "-" or a decimal digit. If no sign is typed the number is assumed to be positive. A number is terminated by the Enter key and execution of the program continues. The number that was typed is converted to a 15-bit binary number and stored at the Memory Reference location. If the number is to be interpreted as signed the absolute magnitude must be less than 8,192 (2^{14}). If unsigned, the input must be positive and less than 16,384 (2^{15}). If input exceeds these limits, the converted number will be reduced modulo 2^{15}.

KDD Key Double Decimal to MR and MR+1. (OP CODE 62)

When this instruction is executed the string Enter a long number: is output to indicate to the operator that a double-precision decimal number is to be typed on the keyboard. The first character may be "+" or "-" or a decimal digit. If no sign is typed the number is assumed to be positive. A number is terminated by the Enter key and execution of the program continues. The number that was typed is converted to a 30-bit binary number and stored at MR and MR+1. If the number is to be interpreted as signed the absolute magnitude must be less than 536,870,912 (2^{29}). If unsigned, the input must be positive and less than 1,073,741,824 (2^{30}). If input exceeds these limits, the converted number will be reduced modulo 2^{30}.

KCH Key Character to MR. (OP CODE 64)

When this instruction is executed the first character that is typed on the keyboard is stored at the memory reference location as a 7 bit ASCII character in bits 9 to 15. Bits 1 to 8 are made zero. This is normally the only form of input from a keyboard on a simple computer. The character is not echoed. All the other input instructions echo the characters typed.

KCS Key Character String to MR+. (OP CODE 66)

Key in characters and store them two bytes to a word starting at MR. The Enter key terminates entry and stores a NULL to terminate the string. Care is taken not to overflow memory.

When *q* or *Ctrl-D* is entered at the keyboard when executing any of the Key input instructions **KDN**, **KDD** or **KCS**, this will stop the running BALAD program and return to the debug input **>>** if it was started from the debugger. Otherwise, the program will terminate. **KCH** will only stop with *Ctrl-D*. *Ctrl-C* will always terminate a BALAD program.

2.2.7 Output Instructions

Print and Type are the same in BALAD

PDN Print Decimal Number at MR. (OP CODE 70)
TDN Type Decimal Number at MR.

Convert the contents of Memory Reference, interpreted as a 15 bit two's complement number, to a decimal character string and type this string on the screen.

PDD Print Double Decimal at MR and MR+1. (OP CODE 72)
TDD Type Double Decimal at MR and MR+1.

Convert the contents of MR and MR+1, interpreted as a 30 bit two's complement number, to a decimal character string and type this string on the screen.

PCH Print Character at MR. (OP CODE 74)
TCH Type Character at MR.

Type the character corresponding to the ASCII code represented by bits 9 to 15 of the contents of the Memory Reference Bits 1 to 8 are ignored.

PRF Print Character String starting at MR. (OP CODE 76)
TCS Type Character String starting at MR.

A Character String is a sequence of characters terminated by a NULL character (ASCII 0). A convention for this machine and many other computers is that character strings are stored 2 characters per word in the following format:

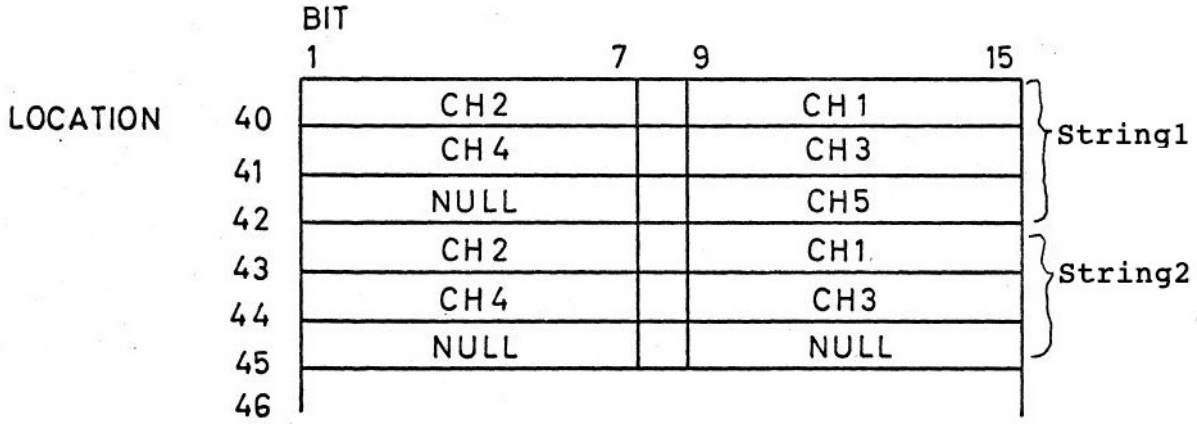


Fig. 9

Character strings may contain page formatting characters of the ASCII alphabet, such as a NEW LINE entered as `\n`, or a TAB entered as `\t`. A real `\` must be entered as `\\`. In practice, character strings are entered into the memory by the Debugging system during Program Assembly as "a character string in parenthesis". The address of the first character in each string is then known and an instruction **`PRF`** for that address will cause that character string to be printed at execution time. This facility can be used to provide spaces between numbers, to start printing on a new line, and to precede key number instructions with a printout of a short message to indicate what the number represents. Printout of results can also be preceded or followed by messages to produce a reasonably workmanlike output of a program and to aid in the identification of results, which is very important.

The **PRF** instruction has been extended so that the parameter string pointed to by **MR** is interpreted like the format string of a *printf* instruction in the Perl or *C* language. The **%** character in such a string starts a conversion defined as follows:

%d	word at next **ADR n** after the **PRF** is output as short signed decimal.
%u	word at next **ADR n** after the **PRF** is output as short unsigned decimal.
%o	word at next **ADR n** after the **PRF** is output as short unsigned octal.
%x	word at next **ADR n** after the **PRF** is output as short unsigned hexadecimal.
%b	word at next **ADR n** after the **PRF** is output as a short unsigned binary.
%s	string starting at next **ADR n** after the **PRF** is output as an embedded string.
%c	character at next **ADR n** after the **PRF** is output as an embedded character.
%%	print a single %.

When the letters **d**, **u**, **o**, **x** and **b** are preceded by a letter '**l**', the word **ADR n** after the **PRF** is interpreted as a long double precision word at **ADR n** and **ADR n+1**. The letters **D**, **U** and **O** are aliases for **ld**, **lu** and **lo**. Field width numbering follows *printf* conventions in Perl or C.

ADR is a pseudo instruction, which tells the assembler that the following **MR** is an address.

2.3 THE ASSEMBLY LANGUAGE

Computer programs are stored in the computer Memory as binary numbers. This is the way a computer reads instructions. As human beings, we devise shortcuts to make what is often referred to as a binary machine language program more tractable. The first step is to divide every binary number mentally into several groups of three bits. In our case for a 15-bit machine, we would have five 3 bit groups.

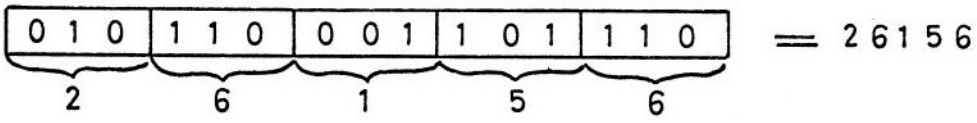


Fig. 10

Then each group of 3 bits can easily be converted into a number between 0 and 7. The 5 numbers together constitute the OCTAL representation of the binary number. OCTAL representations can be used for instructions or data. Certain sub-fields of a Memory location, e.g. the 9 bit Memory reference can be expressed as a 3 digit OCTAL number.

OCTAL representations are easier to handle in the long run than DECIMAL representations because they preserve the regularity of binary groups. OCTAL counting is easy as long as you remember that, you always stop at 7 and then go to the next highest position say 10 or 17 to 20, etc. The main use for OCTAL numbers in this machine is for addressing Memory locations[1].

Another simplification is to break up a binary number into arbitrary fields and to give each possible combination of bits in a field a label. This is a little like labelling 3-bit fields with digits 0 to 7 for the 8 possible bit combinations. In this assembly language, the OP CODE field of instructions has been treated this way. The OP CODE field is bits 1 to 5 and there are 32 labels each of 3 characters to distinguish these 32 codes. The characters are chosen to convey the name of the operation mnemonically eg. **ADD** for *add*, **SUB** for *subtract* etc.

1 NOTE: in modern assembly languages it is more common to break up binary numbers into 4-bit fields, which are expressed as HEXADECIMAL digits, which are 0-9 and a-f for 10-15. This is appropriate for word lengths of 16, 32 and 64 bits, which do not divide neatly by 3. In 1970, when BALAD was first implemented word lengths of 12 and 18 bits were more common.

The indirect bit of an instruction, bit 6, is expressed with the symbol "**@**" if the bit is a 1. No character implies bit 6 is 0. Numerical Memory references in an instruction may be written as OCTAL numbers. Memory locations can also be labelled with an arbitrary text, which is an alias for that numerical address. These labels can then be used instead of absolute OCTAL addresses anywhere in the assembly code, where a Memory reference address is required. The advantage of labels is, that code and data can be moved in memory without changing that code. Simply the value expressed by a label is changed automatically in the symbol table during program assembly.

An address can also be defined as the address of the current instruction with the symbol "**.**" (FULL STOP). The **.** or a label can be followed by + or – followed by a decimal displacement to evaluate an address.

To write a program or data block, the address of the first location must be defined. This is done with the LOC or BLK pseudo instructions. LOC followed by an OCTAL, symbolic or relative address defines the starting address of the block to follow. BLK followed by a DECIMAL displacement will leave a block of memory locations initialized to 0. The next location follows the end of the block.

When typing program code into the computer follow an address defined by LOC or BLK with the 3 character mnemonic of the operation code followed by at least one space and/or the @ symbol if the memory reference is indirect. Lastly type the memory reference of the instruction, either as an absolute OCTAL address, a symbolic address or a relative address using "**.**" optionally followed by a simple displacement. If no memory reference is typed 0 is assumed which means the operation refers to the Accumulator. Lastly type Enter, which will cause analysis of the instruction and cause the next address to be typed on the screen, ready for more program input, unless the instruction contained an error, in which case an appropriate error message will be output on the screen and the previous address is output on the screen again, ready for correct input. This ensures that only syntactically correct programs or data can be entered[2].

Alternatively to writing a block of program statements, we can initialize a block of numerical data. If the first character of a new entry is numeric the rest of the word must be filled by a single number. For convenience, this number may be expressed as decimal, octal, hexadecimal or binary. The convention for modern programming languages is that a number starting with 1 to 9 is a decimal integer. A number starting with 0 is interpreted as octal with only digits 0 to 7. A number starting with 0x followed by digits 0 to 9 and a to f is interpreted as hexadecimal. A number starting with 0b followed by digits 0 or 1 only is interpreted as binary. Only decimal numbers may be preceded by an optional + or – sign. A number followed immediately by the letter l or L is considered to be a 30-bit double-precision number using up two consecutive words.

Character strings are another form of constant that may be initialized during assembly. A character string is entered into memory by typing a parenthesis symbol **"** as the first character instead of a statement or a number. Any character following except another parenthesis **"** is regarded as another character in the string. Strings may contain two control characters written as **\n** for *new-line* and **\t** for *tab* (space to the next column of 8). A real **"** character in a string is written as **\"** and a real \ as \\. A closing parenthesis terminates the string with a NULL byte. The following ENTER key sets the next address to the word address after the end of the string.

2.3.1 Comments

In all cases, before typing ENTER, zero or more spaces followed by "**;**" or "**#**" followed by any text is a comment. For other variations, see the Debugging System.

2 This does not mean a program is semantically correct – the algorithm may not do what it is intended to do.

Here is a small BALAD program example:

```
########################################
#  Comment block
########################################

        LOC 10      ; initialized data block
op1:    99
op2:    81
sum:    BLK 1       ; uninitialized data block

        LOC 100     ; code block
main:   CLR C       ; clear Carry before addition
        LDA op1
        ADD op2
        STA sum
        PDN sum
        HLT
```

This produces the following assembler listing when run with *balad -l*

```
                ########################################
                #  Comment block
                ########################################

                        LOC 10  ; initialized data block
010     00143   op1:    99
011     00121   op2:    81
012     00000   sum:    BLK 1   ; uninitialized data block[3]

                        LOC 100 ; code block
100     34777   main:   CLR C   ; clear Carry before addition
101     30010           LDA op1
102     22011           ADD op2
103     32012           STA sum
104     70012           PDN sum
105     00000           HLT
```

The first two columns are the 3 digit octal memory address and the 5 digit octal memory contents, which may be code instructions or data numbers or strings.

3 Any memory not specifically initialized by the assembler is set to 00000

2.3.2 List of instructions in op code order

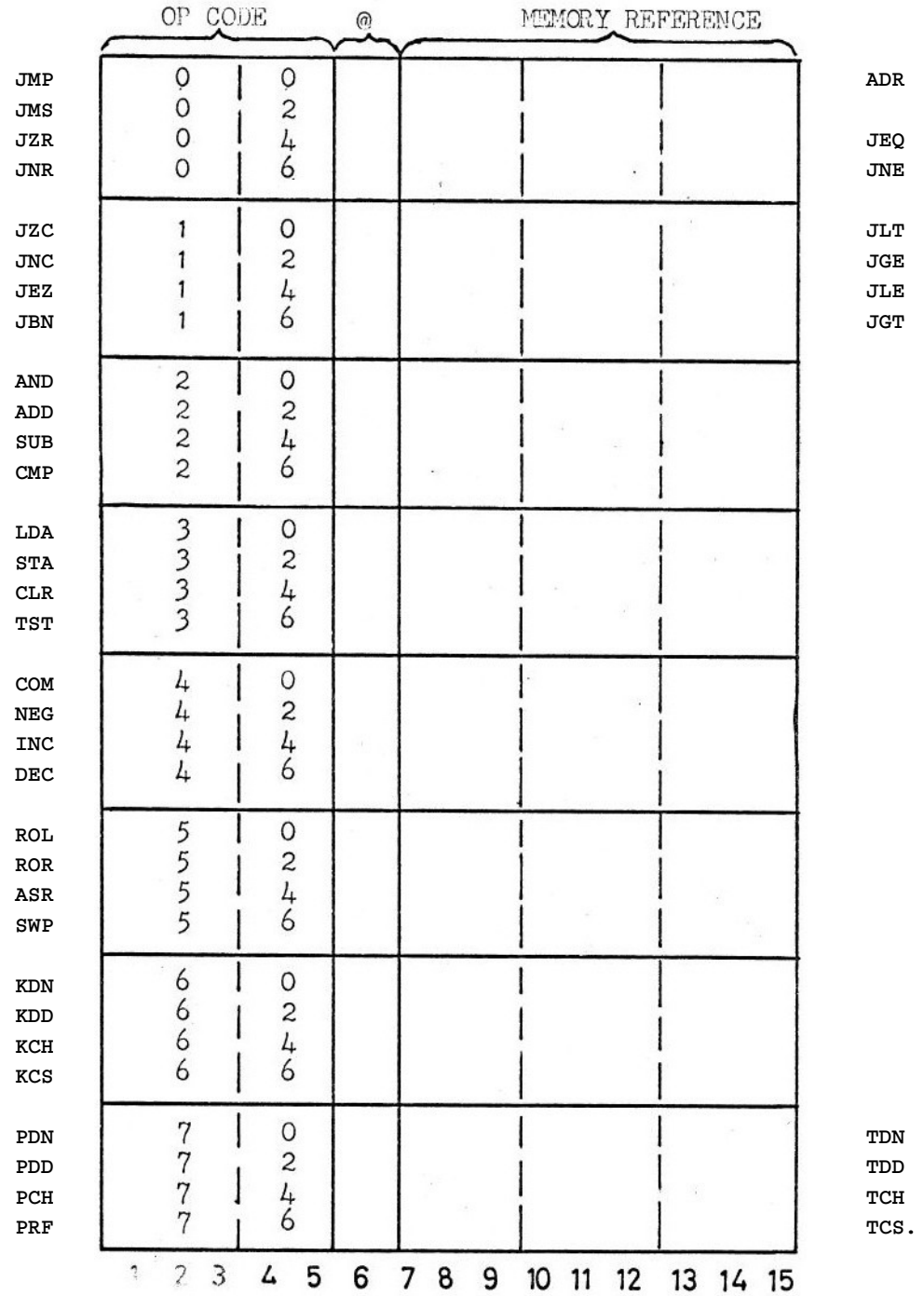


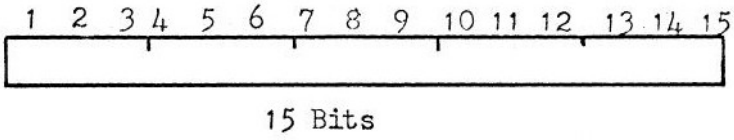
Table 1

Each 15-bit binary word can be expressed as a 5 digit OCTAL number. For instructions, the first two digits are the OP CODE, which is written in assembler code as a three upper case letter mnemonic shown in the left-hand column. A 1 is added to the second digit if MR is indirect (**@**). The Memory Reference is the remaining 3 OCTAL digits. Some op codes have alternate mnemonics useful for **unsigned** numerical comparisons with the CMP instruction.

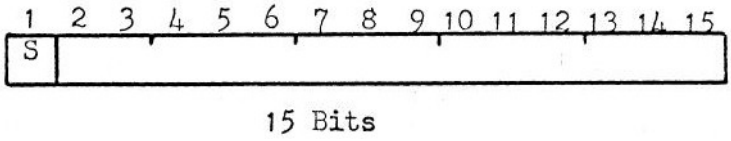
```
CMP dest
JLT .+2     ; jump to .+2 if ACC < contents of location dest
```

2.3.3 Data Formats

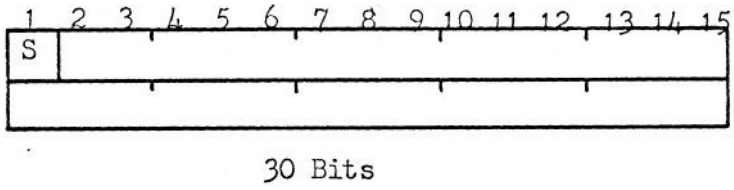
UNSIGNED Integer Number or Logical Unit



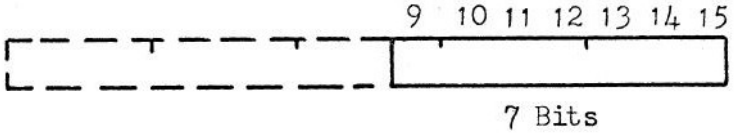
SIGNED Number



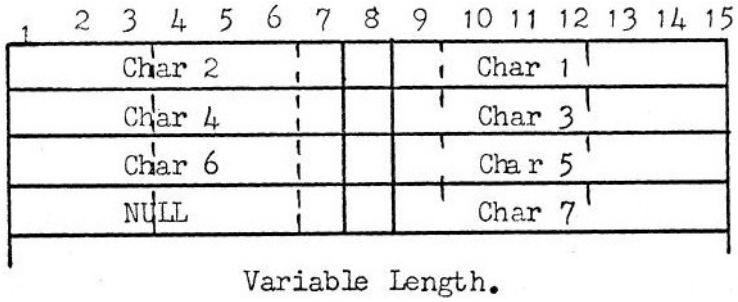
DOUBLE PRECISION SIGNED Number



7 BIT ASCII CHARACTER or BYTE



CHARACTER STRING
A NULL character marks the end of the string



Other Data formats can be devised and these are only limited by the programmers' ingenuity.

2.4 HISTORY

The Balad system was developed in 1970 as an aid for teaching electronics technicians and their teachers the basics of how a computer works internally. Even at that time, the RISC instruction set and the use of 4 Accumulators in the Data General NOVA computer available to them in hardware were confusing the fundamental simplicity of a computer based on the original *von Neumann* architecture.

This is a much greater problem in the 21st century, with CPUs with very extended instructions sets and a much greater reliance on higher-level languages, which hide what is happening inside the computer. **This very simple BALAD virtual computer should give students some insights into how the very core of a computer works. The details can easily be taught in one 45 minute lesson and from then on students can test their skills in manipulating machine instructions to develop higher-level functionality, like a multiply routine, which is not part of the basic instructions set.** HINT: a very simple multiply algorithm is to add the *multiplicand* to the accumulator and decrement the *multiplier* in a loop. Terminate the loop when the *multiplier* is zero.

3 THE BALAD ASSEMBLER

The BALAD assembler is always executed first in two-pass mode when BALAD is called with one or more source file arguments. Two-pass mode means that the source(s) are read once to identify all symbolic labels followed by a colon ":", which builds a Symbol Table. Then the files are read a second time and all instructions, numeric data and strings are converted into 15-bit machine code and stored in the correct memory locations, ready to run. Any errors in the program are reported on the console and in an optional listing file. If there are errors or the call is made with the **-c** option the file(s) are only assembled. Otherwise, the assembled program is started at **`main:`** if there is a label **`main`** – else the debugger is entered to allow starting the program manually with the **`r`** command.

```
balad george.bl
```

Linux and Unix operating systems allow automatic starting of a BALAD program when the following line is the first line of a program:

```
#!/usr/bin/balad
```

The file location **`/usr/bin/balad`** must contain the BALAD executable or a link to it. For this to work your BALAD application source must first be made executable with:

```
chmod +x george.bl
```

Then the following simple call will assemble and start the program automatically without mentioning *balad*:

```
george.bl
```

Any listing, help or debugging switches can follow that direct call.

3.1.1 BALAD Help

When BALAD is called with the **-h** option a Usage help output on the console describes all the command line switches and file parameters. This is followed by a detailed description of the Debugger commands.

```
balad -h

Usage: balad[-lt[doxb]cmh
            [ -L[ <list_file>]][ -O <out_file>]
            [ -B[ <batch_file>]][ <file> …]
       -l   list code output during assembly
       ...
```

The user can also add his own help output to a BALAD program by terminating the normal code and data with a line starting with **`__END__`**. Any text following this terminator is interpreted as a help text, which can be displayed on the console with the BALAD call of the program followed by the **-h** switch. The following program has such a *help* text:

```
george.bl -h

  Reverse Polish Notation Calculator 'GEORGE'
     0-9   Enter a number terminated by white space or an operator,
           at which point the number is pushed on the stack.
  ...
```

3.1.2 Interactive Input Mode

If BALAD is called without any parameters at all, *interactive input mode* is entered, which allows direct entry of BALAD labels, instructions and data from the keyboard. Each line is assembled as soon as the Enter key is typed. This is a quick way to try out a short program. This program entry is

terminated with **q** on a line by itself, which causes entry into the debugger from where the newly entered program can be run. Care must be taken to terminate code with an unconditional **JMP** or **HLT** instruction before entering data, going to a new memory location with **LOC** or **BLK** or quitting interactive entry with **q**. Not doing so causes an Error message. *Interactive input mode* can also be entered from the debugger with the **e** command. This makes it possible to change existing code or enter extra code while debugging if the semantics of the program is not correct. The debugger also allows single memory locations to be modified directly with the **<** command.

A program entered in *interactive mode* can be output to a file from the debugger with the **O <file>** command. If the **<file>** chosen already exists you will be asked if you want to overwrite it. You can also create a list file from the debugger with the **L <file>** command. *Interactive mode* is a quick and easy way for students to try their hand with BALAD code, without having to learn to enter code in a text editor. It was the only way that was available with the original 1970's BALAD version, which had a teleprinter as a terminal on a Data General NOVA minicomputer. Programs could only be printed and saved on paper tape at 10 characters/second – there was no disc.

3.1.3 Listing Options

Calling BALAD or a BALAD application with the **-l** switch will produce a complete listing of the program. Each line of the program which generates a value for a memory location will be preceded by the 3 digit octal address of that memory location and the 5 digit octal contents generated for that location. The listing of the main part of the program george.bl starts as follows:

```
balad -l george,bl
                        LOC 100

  100   30023   main:   LDA stackA      ; top of stack - clear the stack
  101   32014           STA stackP      ; stack grows downwards towards 'expr'

                ##########################################################################
                #  Get a new RPN expression string
                ##########################################################################

  102   66700   loop:   KCS expr        ; grow expression upwards towards stack
  103   30022           LDA expADR      ; expr[] address
  104   32013           STA expPTR      ; word pointer first
  105   02373           JMS iniDig      ; clear digits, hexFl -1, decimal base, C
  106   00110           JMP .+2         ; skip first ROR - start with even byte of expr[0]
```

The listing output is useful for debugging. It correlates with the tracing output at breakpoints and normal tracing.

The listing can also be stored in a file specified with the **-L** switch. Normally a <**listing_file**> name follows the **-L** switch. If no <**listing_file**> is specified the base name of the <**source_file**> with the extension **.bll** is used. The **-L** switch should be put at the end of the command line, because if it is put just before the <**source_file**> name that is assumed to be the listing file. The -- switch before the <source_file> marks the end of all switches and fixes this also. The following calls are all equivalent:

```
balad -l george.bl -L
balad -l george.bl -L george.bll
balad -l -L george.bll george.bl
balad -l -L -- george.bl
```

A listing file can also be generated from the debugger with the **L <file>** command.

4 THE BALAD DEBUGGER

The debugger is an integral part of the system, which is invoked with one of five command-line switches, either alone or in combination. The **-t** switch starts the debugger in *instruction tracing mode*. The normal debugger prompt is the current octal debugger location followed by **>>**.

```
balad -t george.bl
100 >>
```

The debugger can be in one of three modes with different prompts and individual commands:

- program is not running `100 >>` **r** or **e** command allowed
- is stopped at a breakpoint `100 B>` **n s u c** or **a** command allowed
- is stopped at a watchpoint `100 W>` **n s u c** or **a** command allowed

The following additional commands work in all modes:

```
O         output the source text to STDOUT
O <file>   output the source text to <file>
L         output the listing text to STDOUT
L <file>   output the listing text to <file>
S         output the Symbol Table         (sorted by symbols)
Si        output the inverse Symbol Table (sorted by address)
H         output the application Help text if any
h         output the BALAD debugger help text
q         QUIT balad

*         set a BREAK or watch point at current location
<n>*      set a BREAK or watch point at location <n>
<n>,<m>*  set BREAK or watch points at locations <n> to <m>
#         CLEAR all break or watch points
.#        CLEAR a break or watch point at current location
<n>#      CLEAR a break or watch point at location <n>
<n>,<m>#  CLEAR break or watch points at locations <n> to <m>
=         list all break or watch points
<n>,<m>=  list break or watch points in range <n> to <m>
< <code|data>     assemble <code> or <data> at current location
<n> < <code|data> assemble <code> or <data> at location <n>
          this makes the modification of single locations possible

    List commands show an octal memory address and the contents
/c        list current location as code
<n>/c     list location <n> as code
<n>,<m>/c list locations <n> to <m> as code
    Similarly list data in different list modes
/d    short signed decimal       /D    long signed decimal
/u    short unsigned decimal     /U    long unsigned decimal
/o    short unsigned octal       /O    long unsigned octal
/x    short unsigned hexadecimal /X    long unsigned hexadecimal
/b    short unsigned binary      /B    long unsigned binary
/s    text string up to next NULL
    these apply to all other commands starting at * (set a BREAK)
/ or enter    list location(s) in current list mode

-t    trace code during execution.
  -d  additionally  trace C, ACC and MR in decimal
  -o  alternatively trace C, ACC and MR in octal
  -x  alternatively trace C, ACC and MR in hexadecimal
  -b  alternatively trace C, ACC and MR in binary
          In each case trace values before and after execution.
```

```
            Without -t trace only at break or watch points.
   -     stop tracing code and data during execution.
```

Memory addresses shown as <n> or <m> above can be entered in several ways:

- An octal number. The BALAD memory is only 512 words long, which makes 0 – 777 the only valid addresses.
- A defined symbolic address followed by an optional + or – decimal offset. If the command following a symbolic address is a letter it must be separated from the address by a space.
- The symbol "**.**", which stands for the current location, followed by an optional + or – decimal offset.
- Any of the above, preceded by the symbol **@**, which means the address is the contents of the chosen location.

4.1.1 Running a program

A program is started from the debugger with the **r** command. Without a preceding address, the program is started at the label **`main:`**. If there is no such label the program is started at location `100`, which is the default location where code is normally started. But it is safer to start with a specific location preceding the **r** command in this situation.

```
100 >> 155r
```

Specifically start execution at location `155`. The **r** command can only be called when a program has not been running and stopped at a break or watch point.

4.1.2 Instruction Tracing

When a program is run after starting with the **-t** switch, every instruction that is executed is traced by printing the address and then the instruction code as an octal number and as a symbolic assembler instruction.

```
balad -t george.bl
100 >> r
        100 *****  run *****
main:   100 30023  LDA stackA
        101 32014  STA stackP
loop:   102 66700  KCS expr
RPN:
```

Other switches which start the debugger are **-to**, **-td**, **-tx** and **-tb** or without the tracing option as **-o**, **-d**, **-x** and **-b**. Calling the program with **-to** or **-o** additionally traces the contents of the Carry and Accumulator as well as the Memory Reference address of the current instruction [in square brackets] and the contents of the Memory Reference - all before and after the instruction is executed. Lastly, the binary values of the Jump Tester Carry register **jC** and Result register **jR** are traced. The Jump Tester registers are the only output of the **CMP** and **TST** instructions. The values of **jC** and **jR** influence subsequent conditional Jump instructions.

```
balad -to george.bl
100 >> r
        100 *****  run *****
main:   100 30023  LDA stackA    C 0 ACC 00000 [023] 00776 ==> C 0 ACC 00776 [023] 00776  jC 0 jR 1
        101 32014  STA stackP    C 0 ACC 00776 [014] 00000 ==> C 0 ACC 00776 [014] 00776  jC 0 jR 1
loop:   102 66700  KCS expr      C 0 ACC 00776 [700] 00000
RPN:
```

The example shows how the first instruction **`LDA stackA`** loads the stack address 00776 from location [023] into the Accumulator, which was previously 00000 and stores it with the **`STA stackP`** instruction in the stack pointer, location [014], which was also previously 00000. Any

trace will stop when encountering an input instruction, which is the `KCS` instruction in this example with the prompt `RPN:` . Entering a value at the prompt and typing Enter will continue the trace. Note: only the data before the instruction is shown for an input instruction, because the instruction has not been fully executed yet. The trace data after execution is shown after typing Enter.

When tracing instructions in an arithmetic algorithm the contents of the Accumulator and the memory reference can be shown as decimal numbers with the **-td** switch. For completeness these contents can also be traced in hexadecimal with the **-tx** switch and binary with the **-tb** switch. The tracing modes can be changed from within the debugger using the same mnemonics **-to**, **-td**, **-tx** and **-tb**. In the debugger, the mnemonic - switches tracing off altogether. The tracing modes **-o**, **-d**, **-x** and **-b** only output trace data just before and just after the occurrence of a **breakpoint** or **watchpoint**, which will be covered next.

4.1.3 Breakpoints

Locations in a block of program code can be marked as a breakpoint. For this debugger, any number of breakpoints can be set. Breakpoints are set with the ***** command. The following sets the current location `100` as a breakpoint:

```
100 >> *
  100  *  30023   main:   LDA stackA
100 >>
```

Setting the breakpoint echoes the listing line of that instruction with a ***** symbol after the memory address to show that a breakpoint has been set at that location. All subsequent listings will show that star until the breakpoint has been cleared with the # command, which will clear all breakpoints:

```
100 >> #
  100     30023   main:   LDA stackA
100 >>
```

A list of all breakpoints can be output with the = command. Most debugger commands can be preceded by an address or a range of addresses. The following sets breakpoints on 2 consecutive locations:

```
100 >> 103,104*
  103  *  30022           LDA expADR
  104  *  32013           STA expPTR
100 >>
```

Ranges are two addresses separated by a comma. The example shows the use of octal addresses but symbolic addresses can also be used. One gets pretty used to octal addresses, which are available from any listing.

Breakpoints come into play when a program is executed. When the instruction at location `103`, marked as a breakpoint, is about to be executed the instruction and trace data (if one of the data-trace switches -o -d -x or -b is set) is output followed by the prompt `103 B>`

```
100 >> r
RPN: 5 6 +
  103 30022  LDA expADR   C 0 ACC 00776 [022] 00700
103 B>
```

This shows that we are about to execute the `LDA expADR` instruction at location `103` with the values `0`, `00776` and `00700` in Carry, Accumulator and location `[022]`. The most common command at a breakpoint is to **continue** execution to the next breakpoint, which is the **c** command:

```
103 B> c
                                                    ==> C 0 ACC 00700 [022] 00700  jC 0 jR 1
       104 32013  STA expPTR   C 0 ACC 00700 [013] 01601
104 B>
```

The first output after the **c** command is the trace values after execution of the just-completed breakpoint instruction. That shows that the value `00700` from location `[022]` has indeed been loaded into the Accumulator. The next breakpoint is at location `104`, which again shows the appropriate

trace information before executing that instruction. If no breakpoint had been set at location **104**, the same effect would have been achieved with the **step** command **s**, which causes a break at the very next instruction, even if that instruction steps into a subroutine. Alternatively, the **n** command breaks at the **next** instruction but will step over subroutine calls – treating the **JMS** instruction as if it were just one machine instruction. Once inside a subroutine, the **u** command will continue execution in that subroutine **until** it leaves that subroutine. The **abort** command **a** will take the system from a breakpoint to the *not running* state with the >> prompt, effectively stopping the program and returning to normal debug mode.

4.1.4 Watchpoints

Watchpoints are very similar to breakpoints, but their operation during the execution of a program is slightly different. Watchpoints are also set with the ***** command, but instead of marking an instruction, a data location is marked. Such a data location is normally not executed as an instruction, which means it is not a breakpoint. Instead, data locations are read and sometimes modified by various instructions. Each time this happens for data at a watchpoint location, which is marked by a *, a trace line for the instruction referencing that data is output with the word **watch** appended. When the data at a watchpoint is modified, execution of the program stops and the debugger is re-entered with the watchpoint prompt **012 W>**. The address shown is the address of the data location being watched. Unlike with a breakpoint, the instruction must be fully executed before it can be determined that a modification of the data has occurred. The following example shows a watchpoint in operation on a variable **sign**, which happens to be at location **012**:

```
100 >> sign*
        012  *  00000   sign:   0
100 >> r
RPN: 3 4n +
  502 32012  STA sign     C 0 ACC 00000 [012] 00000 ==> C 0 ACC 00000 [012] 00000  jC 0 jR 0  watch
  505 26012  CMP sign     C 0 ACC 40000 [012] 00000 ==> C 0 ACC 40000 [012] 00000  jC 1 jR 1  watch
  507 44012  INC sign     C 0 ACC 40000 [012] 00000 ==> C 0 ACC 40000 [012] 00001  jC 0 jR 1  watch
012 W>
```

For the two instructions at location **502** and **505**, the value of **sign** [012] has not changed. The **INC sign** instruction at location **507** does change **sign** from **00000** to **00001**, thus causing a watchpoint break. Intermediate instructions **503 504** and **506** operating on other memory reference locations are not traced unless in full tracing mode with the **-t** switch.

This particular watchpoint on the variable **sign** allowed me to find a subtle bug during the development of the *Reverse Polish Notation Calculator* GEORGE. **sign** is used in four different subroutines **add**, **sub**, **mul** and **div**, which execute the arithmetic operations in the calculator. **sign** is used to adjust the result of these double-precision operations. I had forgotten that I also called **sub** in the **div** routine. This led to the value of **sign** for the **div** result being altered by the embedded call to **sub**, leading to erroneous results. The use of a watchpoint on **sign** showed the change in **sign** in the routine **sub** during the execution of **div** very clearly.

To continue from a watchpoint the same commands **s**, **n**, **c** or **u** as for breakpoints are used.

The way breakpoints and watchpoints work in BALAD are similar to their functionality in other assembler and higher level-language Debuggers and *Integrated Development Environments*. In particular, these other Debuggers all use the letters **s**, **n**, **c** and **u** as keyboard accelerators for the **step**, **next**, **continue** and **until** operations. Thus learning to debug simple programs or algorithms in BALAD should be a good learning experience for using Debuggers with other language systems. The main skill in debugging is seeing how variables vary by setting appropriate breakpoints and

interpreting the values in the accumulator and the instruction memory reference locations. Other locations can be listed at any time in different list modes, which will be discussed next.

4.1.5 Listing or viewing memory locations

Memory locations or groups of memory locations in any computer are fixed-length bytes or words, which are identified by an address. In BALAD the word length is 15 bits. The interpretation of memory words depends on which part of the computer is accessing a particular memory location. There are three main ways of interpreting computer memory and several sub-categories:

1. Instructions. In BALAD these are very regular consisting of a three upper-case letter op-code followed by an optional '@' symbol and then an octal or symbolic memory reference.
2. Numbers. These may be 15-bit single-precision integers or 30-bit double-precision integers. BALAD instructions can only work directly on single-precision numbers, but many function algorithms deal with double-precision numbers, which makes it useful to be able to list pairs of memory locations as double-precision numbers while debugging. Numbers may be interpreted as signed or unsigned. Numbers can also be displayed as octal numbers, which are identified by a leading '0', hexadecimal numbers with a leading '0x' and binary numbers with a leading '0b'. The '-' symbol is only be displayed with a decimal output. All octal, hexadecimal and binary numbers are displayed as unsigned. If signed, the first bit of these numbers is the sign bit. Other computers also have floating-point numbers, which are not supported by BALAD. In principle a group of functions to carry out floating-point arithmetic is feasible, but the amount of memory available in this implementation of BALAD is not sufficient to do it. So the debugger does not support them either.
3. Strings. In BALAD strings are stored two 7 bit ASCII characters per word, terminated by a zero or NULL byte. Strings are variable-length data structures.

The debugger command to start a list of one or more memory locations is `/`. The `/` is optionally preceded by a memory address (octal, symbolic or relative). The `/` is optionally followed by a single letter list mode, which determines how this location or group is to be interpreted:

```
/c    list current location as code

/d    single-precision or short signed decimal
/u    single-precision or short unsigned decimal
/o    single-precision or short unsigned octal
/x    single-precision or short unsigned hexadecimal
/b    single-precision or short unsigned binary

/D    double-precision or long signed decimal
/U    double-precision or long unsigned decimal
/O    double-precision or long unsigned octal
/X    double-precision or long unsigned hexadecimal
/B    double-precision or long unsigned binary

/s    text string up to next NULL
```

A subsequent `/` command or an Enter without a command – either optionally preceded by an address or address range will list memory locations in the most recently used list mode.

```
100 >> main,loop /c
  100   30023   main:   LDA stackA
  101   32014            STA stackP
  102   66700   loop:   KCS expr
```

```
103 >> /
  103   30022           LDA expADR
104 >>
```

In the following we inspect the contents of a memory block pointed to by **stackP**, with **@stackP** which is where double precision numbers are stored in the GEORGE RPN calculator:

```
776 >> stackP/o
  014   00774   stackP:   0774
015 >> @stackP/O
  774   04000             025004000
  775   00250
776 >> @stackP/D
  774   04000              5507072
  775   00250
776 >>
```

The **@stackP** address uses the contents of **stackP** as the location to display the two-word octal number. The next display shows the same number as a decimal. As with all listing modes, the 5 digit octal memory values for both words are also shown, and any labels if defined.

4.1.6 Modifying memory locations

An important aspect of debugging is being able to modify the contents of memory locations, either before a program is run or at a break or watchpoint. The debugger command to do this is **<**. Any text after the **<** command is passed to the assembler to interpret and convert the text to a correct binary value to store in the memory location preceding the **<** command. That location may be the current location if no new address is typed, an octal or symbolic address or a range, which will all be modified with the same value. Values passed may be BALAD instructions if modifying code, numerical short or long numbers (double precision numbers have a trailing l or L) in any of the input bases – decimal, octal, hexadecimal or binary. When modifying a string with a new "string" several words may be involved, which can be tricky. In all cases, the first line of output shown after the **<** command is a listing of the previous contents, which is followed by the assembly line of the new contents. No listing mode letter may follow a **<** command. The temporary listing mode is determined by the type of value to be assembled – code, number or string.

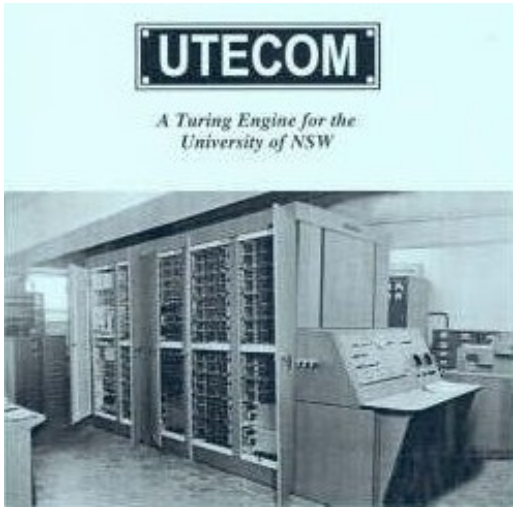
```
111 >> 103<LDA expPTR
  103   30022           LDA expADR
  103   30013   LDA expPTR
104 >> 770<55
  770   00041               33
  770   00067   55
771 >> 772<1234567L
  772   00041             1081377L
  773   00041
  772   53207   1234567L
  773   00045
774 >> 720<"Hello world\n"
  720   00000                   ""
  720   62510   "Hello world\n"
  721   66154
  722   20157
  723   67567
  724   66162
  725   05144
  726   00000
727 >>
```

5 "*GEORGE*" a Reverse Polish Notation Calculator

To show the power of the BALAD computer, I developed the program ***GEORGE*** *(george.bl)*, which is a complete implementation of a Reverse Polish Notation (RPN) Calculator for double-precision integer arithmetic (30-bit precision). I did this with two aims in mind:

1. To demonstrate what can be done with a minimal Turing complete instruction set computer and limited memory of 512 words (1024 bytes). Most of the demonstration programs I have supplied are toy programs to demonstrate various aspects of the instruction set. ***george.bl*** is a full application with many of the capabilities of the calculator ***dc*** developed for early UNIX systems and still available for Linux (***dc*** is a reverse-polish desk calculator which supports unlimited precision arithmetic). Other well-known RPN calculators are the ***HP-42S*** Scientific calculator introduced in 1988 and ***PCalc*** available on iPads and other mobile devices, which has an RPN mode. The main limitation of ***george.bl*** compared with these systems is the lack of floating-point arithmetic and the ability to use named variables. Only the lack of memory stops one from adding these capabilities.

2. To show 21st-century students how computer languages developed in the early days of computing and the importance of the stack concept in computer languages. *Charles Leonard Hamblin* was the professor of philosophy at the NSW University of Technology, which later became the University of NSW in 1955. Among his most well-known achievements in the area of computer science was the introduction (some sources also say invention) of *Reverse Polish Notation* and the invention of the stack in computing. This was arguably independent of and about a year before the work of *Friedrich Ludwig Bauer* and *Klaus Samelson* on the invention of the push-pop stack. In the second half of the 1950s, he became active with UTECOM, the third computer available in Australia, which was a DEUCE computer produced by the English Electric company. DEUCE was based on the ACE computer which Alan Turing had designed. For UTECOM *Charles Hamblin* sketched one of the first higher-level programming languages, GEORGE, which was based on *Reverse Polish Notation*, including the associated compiler (language translator), which translated the programs formulated in GEORGE into the machine language of the computer. In 1957 the GEORGE compiler was operational, and I had the pleasure of attending *Charles Hamblin's* philosophy lectures during the early part of my Electrical Engineering studies at UNSW. He explained the workings of GEORGE to us and the use of Reverse Polish Notation. I wrote my first computer program in this language and had it run on this very large machine. An incidental memory I have of that computer run was being allowed to take a pocket full of vacuum tubes out of a bucket – they had been discarded as being below specs. I used them to make radios at the time.



   About a year later *Friedrich Ludwig Bauer*, *Klaus Samelson* and *Peter Naur* specified the computer language ***ALGOL58***, which also used push-pull stacks for variable manipulation and subroutine nesting – a strategy which has been maintained via ***PASCAL*** and ***C*** through to all modern computer languages.

Here are some extracts from the original GEORGE Programming and Operation Manual.

GEORGE, or the 'General Order Generator', is a program for DEUCE permitting mathematical problems to be presented to the machine in a simple "address-less" instruction language, here called "G-Code". To use this code the programmer must learn a special method of writing mathematical formulae, known as "reverse Polish" notation. Once this is mastered, however, programming is considerably easier and quicker than by other methods. "G-Code" is a highly simplified and condensed instruction language. The program in G-Code in fact resembles a mathematical formula for the result required more than it does an orthodox machine program. In particular, the programmer never has to specify any "addresses" for numbers or instructions inside the machine.

As written, a program in G-Code is a sequence of mathematical symbols such as numerals, variables and arithmetical and other special signs; the symbols are transcribed to cards in a numerical code (on today's computers a program is stored in a file in ASCII code – which is also a numerical code). Broadly, each symbol may be regarded as an individual "instruction" of the program.

5.1 PROGRAMMING GEORGE

5.1.1 Reverse Polish Notation

The majority of the symbols used in programming are those normally used in mathematics, but the order in which they occur is different from the usual order: formulae are written in "reverse Polish" notation. This notation has several advantages which fit it for use with machine computation.

In this preliminary description let us take for granted that we can use the letters "`a`", "`b`", "`c`", … as ordinary algebraic variables. A mathematical formula is a prescription for operating on the numbers such variables represent.

> IMPORTANT NOTE for the BALAD implementation ***george.bl***: variables in formulae have not been implemented because of lack of memory. Integer numbers are put directly into the formulae, which are then executed just like the 1957 version of GEORGE.

We can classify the operators involved as:

i. monadic operators, or operators on a single number, such as the minus-sign in "`-a`".

ii. dyadic operators, or operators on a pair of numbers, such as the operators for addition, subtraction, multiplication and division.

In ordinary mathematical notation, a monadic operator is most frequently written in front of the number concerned, as in "`-5`", and a dyadic operator between two numbers, as in "`7 + 9`". Because the sign "`-`" is used indifferently for subtraction and as a minus-sign (subtraction from zero), brackets are needed to distinguish, say, "`-4 + 6`" and "`-(4 + 6)`".

In reverse Polish notation the operator-signs, whether monadic or dyadic, are written after the numbers concerned, as follows:

For	`7 + 9`	write	`7 9 +`
"	`7 - 9`	"	`7 9 -`

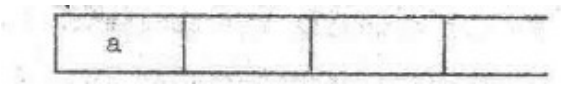
"	`7 * 9`	"	`7 9 *`	
"	`774 / 9`	"	`774 9 /`	
"	`-5`	"	`5 n`	and so on.

One result of this is that brackets are never needed: any of the above expressions may be used directly and without ambiguity as an element in a longer expression. For example:

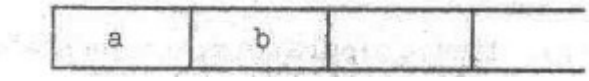
For	`-(7 + 9)`	write	`7 9 + n`	
"	`7 + 9 + 15`	"	`7 9 + 15 +`	
			or `7 9 15 + +`	
"	`(7 + 9) * 15`	"	`7 9 + 15 *`	
"	`7 + 9*15`	"	`7 9 15 * +`	
"	`7*9 + 15/3`	"	`7 9 * 15 3 / +`	

In understanding the use of this notation in GEORGE it will be of assistance to have in mind a picture of the internal logic of operation. The pseudo-machine into which DEUCE (BALAD in our case) is converted by the basic GEORGE program can be envisaged as equipped with a "running stack": this is a set of storage locations (known as "cells") arranged in linear order and operated on the "last-in-first-out" principle. When a number occurs in a program, its "value", as a binary number, is placed in the first vacant cell of the stack, i.e. in cell 1 if this does not already contain a number, otherwise in cell 2 etc.: and when an operator-symbol occurs, the specified operation is carried out on the contents of the last occupied cell or the last two occupied cells, depending on whether the operator is monadic or dyadic. The operation of the formula "**`a b +`**" may for example be pictured as follows: (I am using variables "**`a`**" "**`b`**" "**`c`**" of the original GEORGE handbook, which have to be numerals for the BALAD version ***george.bl***)

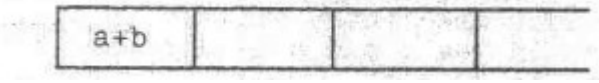
i. "**`a`**": number is transferred to cell 1.



ii. "**`b`**": number is transferred to cell 2.

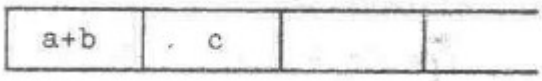


iii. "**`+`**"" contents of last two occupied cells added together; cells cleared and result replaced.

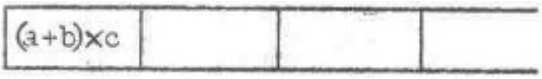


The result is exactly the same as if a single number, the sum of **`a`** and **`b`**, had been specified. In consequence, this number is available for further operations if required. For example to calculate "**`(a + b) * c`**" we can write "**`a b + c *`**". The first three steps are as above, after which:

iv. "**`c`**": number c is transferred to cell 2.



v. "**`*`**": contents of last two cells multiplied; cells cleared and result replaced.



Alternatively, this calculation could have been carried out in the form "**`c a b + *`**". In this case, three cells would first have been filled with numbers before any calculations were carried out: the result of the addition would go into cell 2, and the final result as before into cell 1.

Quite generally, the overall effect of any calculation of a number is to place the number concerned in the first vacant cell of the stack.

5.1.2 Other Operators

There are two special operators which, although not strictly necessary, are frequently useful:

i. "**`dup`**" this "duplicates" the contents of the last cell of the stack in the next cell. In ***george.bl*** duplication is executed by "**`E`**" the Enter operation in conformity with ***HP-42S*** and ***PCalc***.

   For **`(a + b)`**^{2} write **`a b + dup *`** or for ***george.bl*** **`9 7 + E *`**

ii. "**`rev`**" this "reverses" (i.e. interchanges) the contents of the last two occupied cells. In ***george.bl*** "**`rev`**" is executed by "**`S`**" for Swap in conformity with ***HP-42S*** and ***PCalc***.

   For **`a + b + c / (a + b)`** write **`a b + dup c rev / +`** or for ***george.bl*** **`9 7 + E 100 S / +`**

iii. "**`n`**" is negate in ***george.bl*** instead of "**`neg`**" in GEORGE.

iv. "**`(a)`**" in GEORGE means store the contents of the last cell in variable **`a`**. Since ***george.bl*** has no variables, it has no such operator.

The following useful stack manipulation operators from ***HP-42S*** and ***PCalc*** were not available in GEORGE but have been implemented in ***george.bl***:

v. "**`R`**" for Roll. Move the last cell on the stack to the first cell of the stack after moving all other cells across one position.

vi. "**`D`**" for Drop. Free the last cell on the stack, making it available for a new number.

vii. "**`C`**" for Clear the whole stack. This allows starting a new set of calculations.

White space, which is a SPACE, a TAB or a CR (enter) character, as well as any other operator (including E) immediately after entry of a new number, will push that number on the stack. Only then is the operation executed if it was an operator. After that white space is ignored until a new number is entered and further operators carry out their function on numbers already on the stack.

Numbers in ***george.bl*** can be input and output in different bases, namely decimal, octal (base 8), hexadecimal (base 16) and binary (base 2). This is a very useful feature for supporting machine-level programming to do address arithmetic. Assemblers (including the BALAD assembler) present memory addresses and their contents either in octal or hexadecimal. Binary presentation is useful

for bit manipulation. All the calculators which served as a model for ***george.bl***, namely ***dc***, ***HP-42S*** and ***PCalc*** have input and output in the four number bases.

For ***george.bl*** the format of different base numbers follow the strategy of modern computer languages like ***C*** and ***Perl***:

- A number starting with 1-9 followed by digits 0-9 is decimal.
- A number starting with 0 followed by up to 10 digits 0-7 is octal.
- A number starting with 0x followed by up to 8 digits 0-9 and a-f is hexadecimal.
- A number starting with 0b followed by up to 30 digits 0 and 1 only is binary.

Numbers may only be integers. Therefore no decimal point is allowed. The result of a calculation is shown in decimal at the end of a reverse Polish expression string. Normally there is only one result, but some expression strings leave values on the stack. For this reason, the whole of the current stack is output at the end of the expression string. For debugging your reverse Polish expression this information is very useful. Most calculators only show the last value on the stack or at most the last two values. The ability to see the whole stack was a feature I always wanted.

george.bl has four extra operators to output numbers in the different bases. These are **`d`** for decimal, **`o`** for octal, **`x`** for hexadecimal and **`b`** for binary. These operators can be put at the end of an expression, in which case the result will be shown in the requested number base. The operators **`d`**, **`o`**, **`x`** and **`b`** serve a dual purpose. They can also be placed more than once in the middle of a reverse Polish expression. This has the effect that each time one of these display operators occur in the expression the stack, as it is at that point of the expression evaluation, will be output in the requested base. On top of that, if the display operator is not at the end of the expression the remainder of the expression will be output after the stack results. The net result is a tracing feature for debugging more complicated reverse Polish expressions. Let me demonstrate this feature with an example taken from the original GEORGE handbook:

> To evaluate **`e = ay`**^{2} **`+ by + c`**, one wrote **`a y dup × × b y × + c + (e)`**.
> For values of **`a = 15`**, **`y = 30`**, **`b = 12`** and **`c = 45`** the ***george.bl*** RPN expression is:
> `15 30 E E R * * R 12 * + 45 +` which evaluates as follows:

```
RPN: 15 30 E E R * * R 12 * + 45 +
 13905
```

> With extra trace output **`d`** before every value input and operator the output is as follows:

```
RPN: 15 d30 dE dE dR d* d* dR d12 d* d+ d45 d+
 15              30 dE dE dR d* d* dR d12 d* d+ d45 d+
 15 30           E dE dR d* d* dR d12 d* d+ d45 d+
 15 30 30        E dR d* d* dR d12 d* d+ d45 d+
 15 30 30 30     R d* d* dR d12 d* d+ d45 d+
 30 15 30 30     * d* dR d12 d* d+ d45 d+
 30 15 900       * dR d12 d* d+ d45 d+
 30 13500        R d12 d* d+ d45 d+
 13500 30        12 d* d+ d45 d+
 13500 30 12     * d+ d45 d+
 13500 360       + d45 d+
 13860           45 d+
 13860 45        +
 13905
```

A similar effect can of course be achieved by entering every input value and operator on a separate line, but the output is not quite as compact and informative:

```
RPN: 15
 15
RPN: 30
 15 30
RPN: E
 15 30 30
RPN: E
 15 30 30 30
RPN: R
 30 15 30 30
RPN: *
 30 15 900
RPN: *
 30 13500
RPN: R
 13500 30
RPN: 12
 13500 30 12
RPN: *
 13500 360
RPN: +
 13860
RPN: 45
 13860 45
RPN: +
 13905
```

5.2 The overall strategy of GEORGE

It is important to remember that computers in the ‘50s like UTECOM were mainly seen as *Number Crunchers* – in other words evaluating numerical formulae. Up till then, the word *computer* had been a *job description* for workers, whose job it was to compute numbers for such projects as the dimensions of beams for building sites or statistical projections for businesses using pencil and paper with mechanical calculators as the only aid.

I did such a project with my father, who was a lecturer in Mechanical Engineering at UNSW. He had undertaken to produce a chart for selecting suitable steel tubes from those produced by the company *Tubewrights* in Port Kembla for use as columns for building projects. It was known that long columns fail by buckling. A Swiss mathematician named Leonhard Euler (1707 –1783) was the first to investigate the buckling behaviour of slender columns within the elastic limit of the column’s material. Euler’s equation shows the relationship between the load that causes buckling of a (pinned end) column and the material and stiffness properties of the column. The critical buckling load can be determined by the following equation.

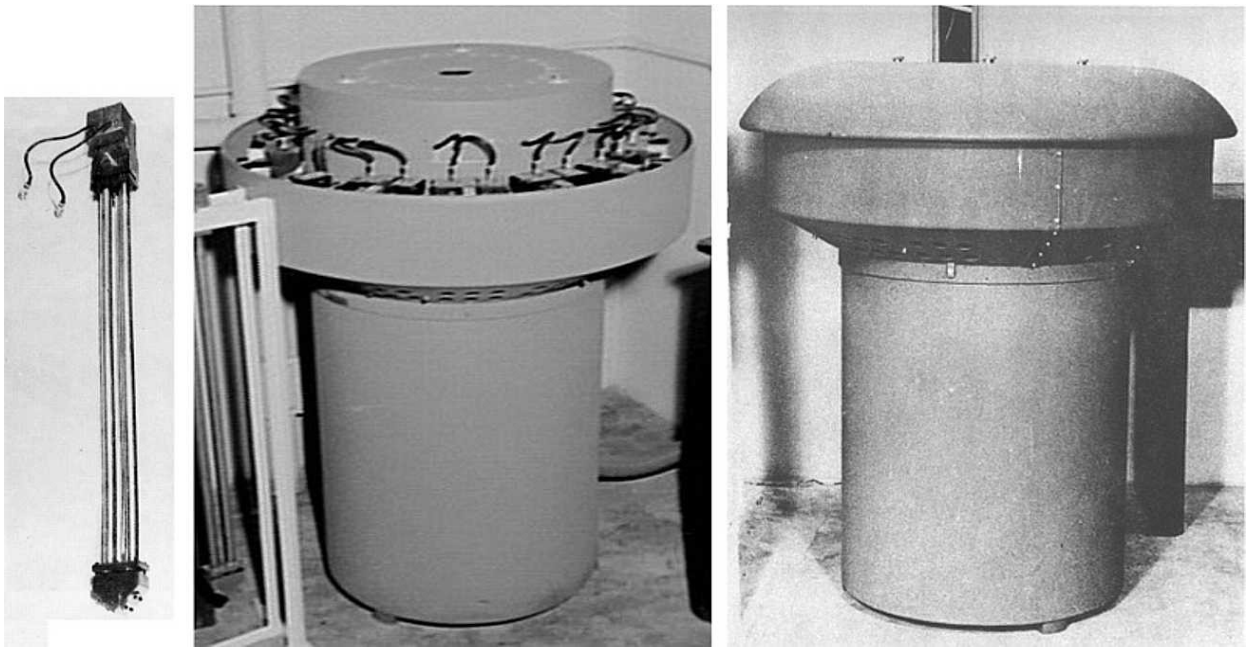
$$P_{critical} = \pi^2 E I_{min} / L^2$$

where

$P_{critical}$ = critical axial load that causes buckling in the column (pounds or kips)
E = modulus of elasticity of the column material (psi or ksi)
I_{min} = smallest moment of inertia of the column cross-section (in^2)
(Most sections have I_x and I_y; angles have I_x,I_y and I_z.)
L = column length between pinned ends (inches)

The chart my father made showed a curved line for each type of tube with load in pounds on the Y-axis and length in inches on the X-axis. For a tube to be suitable as a column the point of intersection of load and length had to be below the line for that tube on the chart. For each of about 50 types of tube, we had to compute 200 points, which meant 10,000 evaluations of Euler's equation. At the time I had just done my exercise with GEORGE on the UTECOM computer. My father tried to get time on UTECOM for his project, but it was rejected. So we worked for a fortnight with a large notebook and a FACIT mechanical calculator to do these 10,000 computations.

It was not until the 60's that computers started getting away from solely *number crunching*, because now they were byte-oriented, which made it much easier to do text manipulation. At the time it was realized that a larger proportion of actual computing time was being spent on manipulating text rather than on crunching numbers. Particularly compilers are very text-oriented.



But in 1957 this was not so. UTECOM was a general-purpose vacuum-tube digital computer, with a serial organization and a 1Mhz clock rate. The word size was 32 bits, and the machine's arithmetic units were capable of performing single, double, and mixed-precision binary integer arithmetic. Negative values were held in two's complement form. A hardware unsigned integer multiplier and a signed integer divider were included. Its primary memory consisted of 12 mercury-filled delay lines. These were "folded" with a quartz crystal transmitter and receiver at the top and a pair of acoustic mirrors at the bottom. Each delay line circulated 32 words of 32 bits each, making a total of 384 words (1536 bytes). Of the 384 words, 256 words (1024 bytes) were used for program memory and 128 words for data memory. This means that the program memory was of similar size as the BALAD memory of 512 x 15-bit words (1024 bytes). UTECOM just had an extra 50% of data memory, which made the storage of variables and other extras possible.

The overall strategy of GEORGE was to use the UTECOM library functions add, subtract, multiply and divide to do the actual arithmetic. But first GEORGE programs, coming in via punched cards, had to be compiled, either directly into machine code, or later into code, which was executed by an interpreter, which was more flexible and allowed calling more functions, like **LOG** and **SINE**.

george.bl uses the latter strategy, which will be looked at in detail in the next section.

5.3 Analysis of the BALAD implementation *george.bl*

The core of the RPN system are the five subroutines **add**, **sub**, **mul, div** and **negate**, which carry out signed double-precision integer arithmetic (very similar to the capabilities of UTECOM). The double-precision unsigned multiply function **mult** is used to multiply the value of incoming numerical digits to binary by multiplying them by the number base. It is also used to implement the signed double-precision function **mul**. These basic functions are supplemented by the functions **push** and **pop**, which implement a stack for double-precision numbers. Another stack function is **showSt**, which is called in two places to display the whole stack.

The program is very short on available memory, so manual optimization was undertaken to eliminate any groups of code that were duplicated and replace them with subroutines, which are called instead of executing duplicated code.

5.3.1 Variables and Constants

LOC 0 and **LOC 1** are the *Accumulator* and its extension for double-precision values. These locations are predefined with the label **ACC** and **ACC+1**. The following double-precision variables **dest, src** and **rem** are initialized with the double-precision constant **0L**. The variables **src**, **lambda** and **savACC** are shared in a number of different routines for quite different purposes. Great care must be taken when doing this, that the variables can be safely shared. The remaining variables carry global states and cannot be shared.

Variables whose contents start with the pseudo-operator **ADR** hold the address of other locations. **expADR** holds the start of the locations where RPN expressions are stored and **stackA** holds the top of the **stack**. A special case is **funPAd**, which hold the indirect address **ADR @funAdr**, which points to the array **funAdr** containing the four function pointers **add**, **sub**, **mul** and **div**. The rest are character constants for identifying operators, number constants and character strings.

5.3.2 Main Program

The first thing done in the main program is to initialize the stack. This point is also looped to when executing the Clear instruction "**C**".

```
100  30021 main:   LDA stackA   ; top of stack - clear the stack
101  32013         STA stackP   ; stack grows downwards towards 'expr'
```

Then we fetch a new RPN expression string:

```
102  66704 loop:   KCS expr     ; grow expression upwards towards stack
103  30020         LDA expADR   ; expr[] address
104  32012         STA expPTR   ; word pointer first
105  02407         JMS iniDig   ; clear digits, hexFl -1, decimal base, C
106  00110         JMP gettok+1 ; skip first ROR - start with even byte
```

We loop to this point every time the execution of the previous RPN expression is finished. The **KCS** instruction reads a character string from the keyboard (or from a batch file with STDIN). Next, the **expPtr** is initialized with the word address of the first character [0] of the RPN expression. This pointer is later turned into a byte pointer, which is incremented to point to the next character. To do

this with the BALAD instruction set the byte pointer is right-shifted and if the Carry is one, the bytes in the word are swapped with the **SWP** instruction. The subroutine **iniDig** (which is called in several other places) resets the counter **digits**, which counts the digits as a number is built from a series of input digits. When **digits** is zero no new number has been built yet. **hexFl** is used in hexadecimal conversion and **asBase** is set for decimal conversion. Lastly the Carry register **C** is set to zero. This is important for subsequent additions and subtractions. The initial entry into the **gettok** loop skips the initial **ROR** instruction.

The following loop picks up consecutive character tokens for numerical digits, operators and white space:

```
107  52012 gettok: ROR expPTR   ; change to word pointer and Carry
110  31012         LDA @expPTR  ; two bytes from the expression
111  10113         JZC .+2      ; skip SWP for even bytes
112  56000         SWP ACC      ; swap bytes for odd bytes
113  50012         ROL expPTR   ; change to byte pointer
114  44012         INC expPTR   ; increment expression byte pointer
115  20063         AND C177     ; mask to 7 bit character from expression
```

String characters are stored two to a word and the code above picks those up in consecutive order. Finally, the word is masked with **AND C177** to leave just the character in the word accumulator for comparisons. Next, we check whether the input character is a numerical digit, by comparing with **ASC_0** and **asBase** whose initial value is for decimal conversion.

```
116  26036         CMP ASC_0    ; "0"
117  10160         JLT notDig
120  26017         CMP asBase   ; "9"+1, "8", "2", "f"+1
121  12160         JGE notDig
```

If the character lies outside this range it is not a numerical digit and the program jumps to **notDig**. The code up to **num1:** is mostly concerned with setting up octal, hexadecimal or binary conversion, whose workings are left as an exercise for the reader. In each case, the variable **inBase** is set to 10, 8, 16, or 2 and temporarily saved in **savACC**.

```
132  24036 digFd:  SUB ASC_0    ; convert ascii to binary 0 - 9
133  32011 convrt: STA savACC   ; save accumulator

134  36014         TST digits   ; is this the first digit ?
135  06151         JNR num1     ; no
136  34002         CLR dest     ; yes - first digit
137  34003         CLR dest+1   ; clear DP result for new number
```

The actual code to do character string to binary conversion is the following:

```
151  44014 num1:   INC digits   ; number is growing
152  30016         LDA inBase   ; 10, 8, 16 or 2
153  32004         STA src      ; multiplier
154  34005         CLR src+1    ; multiplicand dest is previous number
155  30011         LDA savACC   ; restore accumulator - new binary digit
156  02572         JMS mult
157  00107         JMP gettok   ; get next digit or next operator
```

A double-precision binary number is generated from the next numerical digit by first multiplying the previously built number's low and high part by **inBase** (usually 10) and then adding the binary value of the new digit. The result is stored in **dest** and **dest+1**.

The code from **notDig** to **notNum** checks for **x** or **b** after an initial **0** to detect **0x** or **0b**, the leading markers for identifying hexadecimal or binary numbers.

Then the RPN token in the **ACC** is tested for the operations "**q**", "**C**" or "**E**", which are executed, even if no new number has been pushed on the stack.

```
202  26055 notNum: CMP ASC_q    ; "q"
203  04077         JZR errMsg+3 ; quit the program

204  26042         CMP ASC_C    ; "C"
205  04100         JZR main     ; Clear the stack

206  26044         CMP ASC_E    ; "E"
207  06214         JNR tstNum

210  02421         JMS push     ; Enter or duplicate number on stack
211  02407         JMS iniDig   ; clear digits, hexFl -1, decimal base, C
212  34406         CLR showFl   ; show stack after pushing a new number
213  00107         JMP gettok
```

For all other operators push a newly built number if a new number has been built. This includes white space, which is otherwise ignored.

```
214  36014 tstNum: TST digits   ; has a number been built ?
215  04221         JZR noNum    ; no - no new number

216  02421         JMS push     ; yes - push the new number
217  02407         JMS iniDig   ; clear digits, hexFl -1, decimal base, C
220  34406         CLR showFl   ; show stack after pushing a new number

221  30022 noNum:  LDA funPAd   ; address of function pointer array funAdr
222  32010         STA lambda   ; funAdr[0] add (shared)
223  30011         LDA savACC   ; restore accumulator
224  04374         JZR expEnd   ; end of expression - show result
                                ; unless just shown with 'd' 'o' 'x' or 'b'
225  26027         CMP space
226  04107         JZR gettok   ; ignore white space
227  26030         CMP tab
230  04107         JZR gettok
```

Now we come to call the arithmetic functions **add**, **sub**, **mul** and **div**, which are the main reason for having a calculator. They have all been designed to have a similar calling strategy namely to pop two values from the stack, the first to the double-precision variable **src** and the second to the double-precision variable **dest**. Rather than repeat this for each of the four arithmetic functions, it is done once at **execOp**. Leading up to this is a short section, which generates the values 0, 1 2 or 3 in the shared variable **mulplr** depending on whether the current operator is "**+**", "**-**", "*****" or "**/**".

```
231  34406         CLR showFl   ; all other tokens force showing the stack
                                ; funAdr[0] add
232  26032         CMP addOp    ; "+"
233  04261         JZR execOp
234  44010         INC lambda   ; funAdr[1] sub
235  26033         CMP subOp    ; "-"
236  04261         JZR execOp
237  44010         INC lambda   ; funAdr[2] mul
240  26034         CMP mulOp    ; "*"
241  04261         JZR execOp
242  44010         INC lambda   ; funAdr[3] div
243  26035         CMP divOp    ; "/"
244  04261         JZR execOp
```

Check for function **negate**, which has only one parameter **dest** popped from the stack.

```
245  26047          CMP ASC_n    ; "n"
246  06252          JNE test_r
247  02431          JMS pop      ; n negate last value on the stack
250  02460          JMS negate   ; DP negate destination
251  00270          JMP pushDs   ; push result from dest and dest+1
```

The arithmetic function div produces a remainder, which can be pushed on the stack with "**r**".

```
252  26050 test_r: CMP ASC_r    ; "r"
253  06272          JNE test_D
254  30006          LDA rem      ; get remainder of latest div
255  32002          STA dest
256  30007          LDA rem+1
257  32003          STA dest+1
260  00270          JMP pushDs   ; push result from dest and dest+1
```

At this point, we pop **src** and **dest** and call one of the subroutines **add**, **sub**, **mul** or **div**, namely the one whose pointer is in **lambda**. The address is a double indirect, which is not often used but is a very useful feature of the BALAD instructions set. This computed call of a subroutine is similar to *Lambda* functions in Python, which are used in various Python implementations of reverse Polish calculators. *Lambda* functions are called *anonymous* functions in Python, i.e. functions without a name.

```
261  02431 execOp: JMS pop      ; popped value to src and src+1
262  30002          LDA dest
263  32004          STA src
264  30003          LDA dest+1
265  32005          STA src+1
266  02431          JMS pop      ; popped value is in dest and dest+1
267  03010          JMS @lambda  ; execute the selected op add sub mul div
```

All six arithmetic functions return their double-precision result in **dest** and **dest+1**, which is pushed back on the stack.

```
270  02421 pushDs: JMS push     ; push result from dest and dest+1
271  00107          JMP gettok   ; next token
```

Three stack manipulation operations follow.

```
272  26043 test_D: CMP ASC_D    ; "D"
273  06276          JNE test_S
274  02431          JMS pop      ; Drop last value on the stack
275  00107          JMP gettok

276  26046 test_S: CMP ASC_S    ; "S"
277  06303          JNE test_R
300  30013          LDA stackP   ; Swap last two numbers on the stack
301  22060          ADD D4
302  00306          JMP swpNow

303  26045 test_R: CMP ASC_R    ; "R"
304  06337          JNE test_d
305  30021          LDA stackA   ; Roll last number on stack with top of stack
```

Drop simply pops the last number on the stack. This is equivalent to clearing the last number entry.

Swap and Roll share a common execution strategy. The only difference is the value in the accumulator, which is four words back from the contents of the current stack pointer **stackP** for Swap and the top of the stack **stackA** for Roll.

```
306  32010 swpNow: STA lambda   ; shared
307  30013         LDA stackP
310  32011         STA savACC   ; save stack pointer
311  02431         JMS pop      ; last number on the stack (checks empty stack)
312  30013         LDA stackP   ; last number now in 'dest'
313  26021         CMP stackA
314  10317         JLT swp1
315  76074         PRF errMsg   ; "? err\n"
316  00333         JMP swp2     ; restore stack

317  32004 swp1:   STA src      ; temporary stackP from (shared)
320  24057         SUB D2
321  32005         STA src+1    ; temporary stackP to   (shared)
322  30010         LDA lambda
323  32013         STA stackP

324  31004 swpLp:  LDA @src     ; move numbers down on the stack
325  44004         INC src
326  33005         STA @src+1
327  44005         INC src+1
330  30004         LDA src
331  26013         CMP stackP   ; until top for swap or roll is reached
332  10324         JLT swpLp

333  02421 swp2:   JMS push     ; push last number further up the stack
334  30011         LDA savACC
335  32013         STA stackP   ; restore stack pointer
336  00107         JMP gettok
```

Next, we deal with the four debugging operators "**d**", "**o**", "**x**" and "**b**", which show intermediate results on the stack in either decimal, octal, hexadecimal or binary. First, select the number base.

```
337  26051 test_d: CMP ASC_d    ; "d"    decimal
340  04347         JEQ show

341  26052         CMP ASC_o    ; "o"    octal
342  04347         JEQ show

343  26053         CMP ASC_x    ; "x"    hexadecimal
344  04347         JEQ show

345  26054         CMP ASC_b    ; "b"    binary
346  06371         JNE inpErr
```

This is followed by the common **show** code, which uses the characters "**d**", "**o**", "**x**" and "**b**" in the accumulator to modify the format string **fmtDP** to **" %#ld"**, **" %#lo"**, **" %#lx"** and **" %#lb"** for the **PRF fmtDP** formatted print call in the function '**showSt**, which is also called for decimal output when the RPN expression ends to show the result of an expression.

```
347  02446 show:   JMS showSt   ; " %#ld" " %#lo" " %#lx" " %#lb"
```

This is followed by the remainder of the RPN expression at that point, which is a useful tracing feature when debugging a complex RPN expression.

```
350  74030         PCH tab      ; "\t" show remainder of RPN expression
351  74030         PCH tab      ; "\t" separated by a wide space:w
352  30012         LDA expPTR
353  32010         STA lambda   ; shared
```

```
354  52010          ROR lambda
355  31010          LDA @lambda  ; two bytes from the expression
356  10365          JZC show1    ; skip SWP ACC ... CLR C for even bytes

357  56000          SWP ACC      ; swap bytes for odd bytes
360  20063          AND C177
361  04366          JZR show2    ; terminates on 2nd byte
362  44010          INC lambda
363  74000          PCH ACC      ; print first odd byte of rest of expression
364  34777          CLR C

365  77010 show1:   PRF @lambda  ; print rest of expression
366  44406 show2:   INC showFl   ; no need to show stack again if terminated now
367  74031          PCH nl       ; "\n"
370  00107          JMP gettok
```

Any other character token in the RPN expression is not recognized and produces a short but informative error message.

```
371  76071 inpErr: PRF inpEms   ; "? %c\n"
372  00000          ADR ACC      ; token is character in the accumulator
373  00107          JMP gettok
```

Finally, the **expEnd** code, which is executed when the **NULL** terminator of the RPN expression is detected near **noNum** above.

```
374  36406 expEnd: TST showFl   ; was output shown already
375  06102          JNR loop     ; yes - don't show again

376  30051          LDA ASC_d    ; no  - restore decimal output
377  02446          JMS showSt   ; " %#ld"
400  74031          PCH nl       ; "\n"
401  00102          JMP loop     ; get new RPN expression
```

This code is skipped, if the variable **showFl** was set at **show2** above, which means the stack has just been shown already with no RPN expression remaining. Otherwise, the display base is set to decimal and **JMS showSt** is called. The program then returns to **loop** to request a new RPN expression. This is the end of the main program.

5.3.3 Subroutines

The subroutines are relatively short and self-contained with hopefully informative comments. The **pop** function checks, that no attempt is made to pop a number when the stack is empty. If the stack is empty an error message is printed and a **JMP loop** is executed to request a new RPN expression. Such a direct **JMP** out of a subroutine is allowed in BALAD because nested subroutine returns are not organized on a stack like in most modern languages. The **JMP loop** out of the subroutine **pop** is similar to a **longjmp** in the ***C*** language.

Extra code at the beginning and end of each of the four arithmetic routines **add**, **sub**, **mul** and **div** make sure the result is correct for signed integers. Arithmetic overflow for **add** and **sub** as well as divide by zero are flagged as errors. There was not enough memory to detect arithmetic overflow for **mul**. A new unsigned double-precision multiply algorithm made the whole signed multiply much shorter and also shortened the multiplies for building numbers. A fast double-precision divide algorithm by Steve Auer sped up the division. The memory saved was used to provide the "**r**" operation to obtain the remainder of the last division.

5.3.4 Help text

As mentioned in section 3.1.1. the BALAD system makes it possible to extend the program code with a help text, which is output with the -h switch as follows:

```
george.bl -h
Reverse Polish Notation Calculator 'GEORGE'
      0-9     Enter a number terminated by white space or an operator,
              at which point the number is pushed on the stack.
      0-7     A number starting with 0 is octal (0-7 only).
      0-7 a-f A number starting with 0x is hexadecimal (0-9a-f).
      0-1     A number starting with 0b is binary (0-1 only).
      + - * / Arithmetic operators which act on the last two numbers
              popped from the stack. Result is pushed back on the stack.
              Error if there are less than two values on the stack.
      n       NEGATE the last number on the stack.
      r       push the REMAINDER of the last division on the stack.
      Enter   display numbers on the stack as signed decimal.
      d       display numbers on the stack as signed decimal numbers
              before any further operations.
      o x b   display numbers on the stack in octal, hexadecimal or
              binary notation before any further operations.
      E       ENTER a number on the stack if 'E' immediately follows
              a number; else duplicate the last number on the stack.
      D       DROP the last number on the stack.
      S       SWAP the last two numbers on the stack.
      R       ROLL the stack - move last number to the top of the stack.
      C       CLEAR the stack.
      q       QUIT the RPN calculator.
```

5.3.5 BALAD on Windows 10

BALAD is a ***Perl*** program and to run BALAD on Windows, you need to install either *Strawberry Perl* or *Active State Perl* from https://www.perl.org/get.html. BALAD works with both, although I prefer *Active State Perl* because it provides a more UNIX like environment. BALAD runs as a command-line program in a Command window. Both systems initially produce the following error message: *Unable to get terminal size*. To fix that, execute the following once:

```
set TERM=dumb
```

Windows makes no provision to start interpreted scripts automatically from a Command window. For that reason all BALAD programs must be called as follows eg. for george.bl:

```
perl balad <flags> george.bl
```

6 FINALLY

Best of luck with trying out some of your own algorithms using either your favourite editor or *interactive entry mode*. This handbook should provide enough information to do serious programs. Any suggestions or bug reports are very welcome. Please contact me – John Wulff – on:

immediatec@gmail.com with the Subject: BALAD …

The complete BALAD system can be cloned from https://github.com/JohnWulff/balad.