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 which allows on line 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 reasonable print out of their results quickly, while concentrating their efforts on developing algorithms.

Table of Contents

https://github.com/JohnWulff/balad

`$Id: BALAD.odt 1.3  2020/12/14`

2 THE BALAD COMPUTER

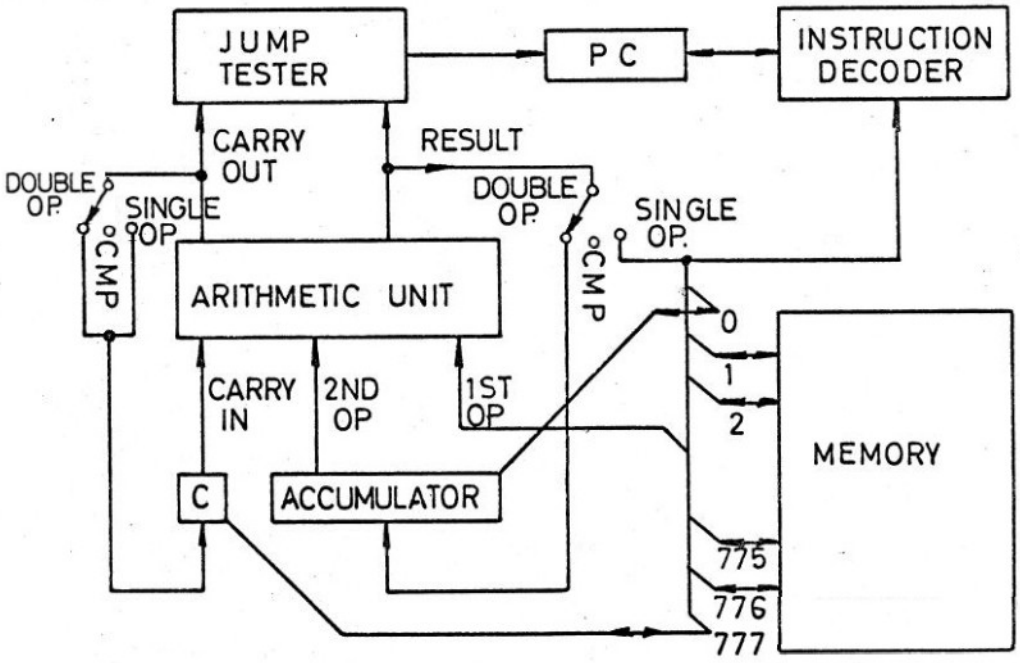


Fig. 1.

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 the next 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 (PC) 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 Fig. 1, which guide the result and carry out 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 which connects one data path to one of 512 words in memory.

2.1 THE INSTRUCTION FORMAT

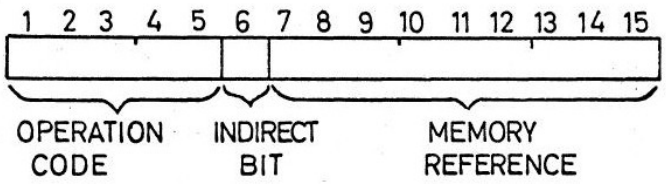


Fig. 2.

Instructions are stored in 15 bit memory registers. Bits 1 to 5 are the operation code, which select 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 in 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.

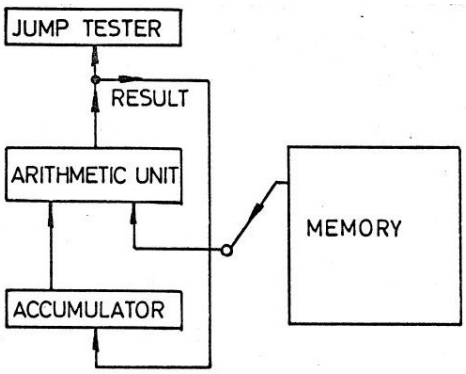


Fig. 3.

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.

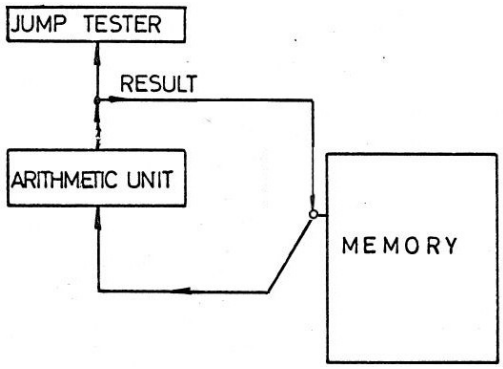


Fig. 4.

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 accounted for the "carry" out of the most significant bit (bit 1) is available to the programmer. In fact the arithmetic unit is extended by one bit to 16 bits, and for practical purposes the 1st operand is extended by 1 bit. Storage for this bit is in the "carry" register. The contents 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 value not stored. The output of the carry position is also stored in the jump tester for subsequent tests for the state of "carry". The Jump Tester is affected by CMP and TST.

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 be carried out on the accumulator and the carry. Since the 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 is 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 is 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 bit wise 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 was 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 was 2^{15}-1 (signed -1), complement carry.

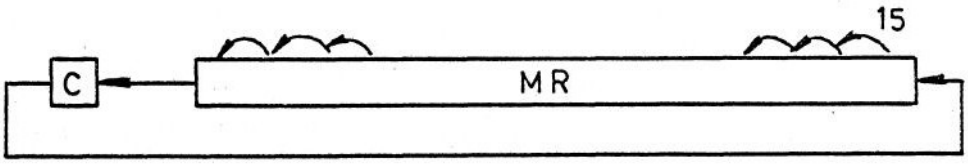
DEC Decrement MR (OP CODE 46)

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 was 0, complement Carry.

2.2.4 Rotate and shift instructions

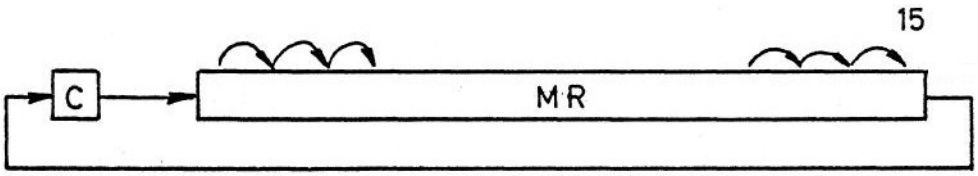
These are single operand operations for bit manipulation, scaling data by factors of 2 and byte manipulation. Rotates are used for testing sequential bits of a word. In each the Carry register and the contents of the Memory Reference are manipulated in different ways, and the result is stored in the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

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



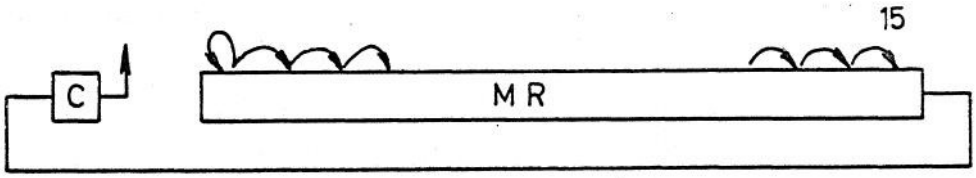
Rotate the contents of the Memory Reference and the Carry Register one bit left as shown in Fig. 5.

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



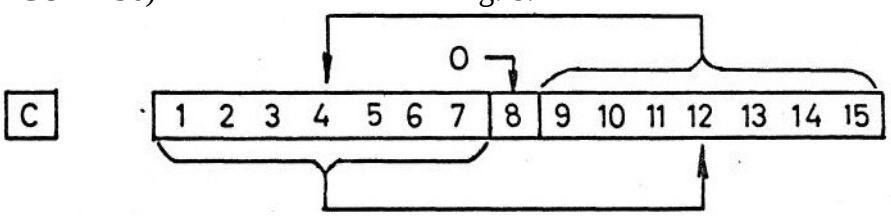
Rotate the contents of the Memory Reference and the Carry Register one bit right as shown in Fig. 6.

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 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, 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, 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.

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 st ring type the 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 for many other computers is that character strings are stored 2 characters per word in the following format:

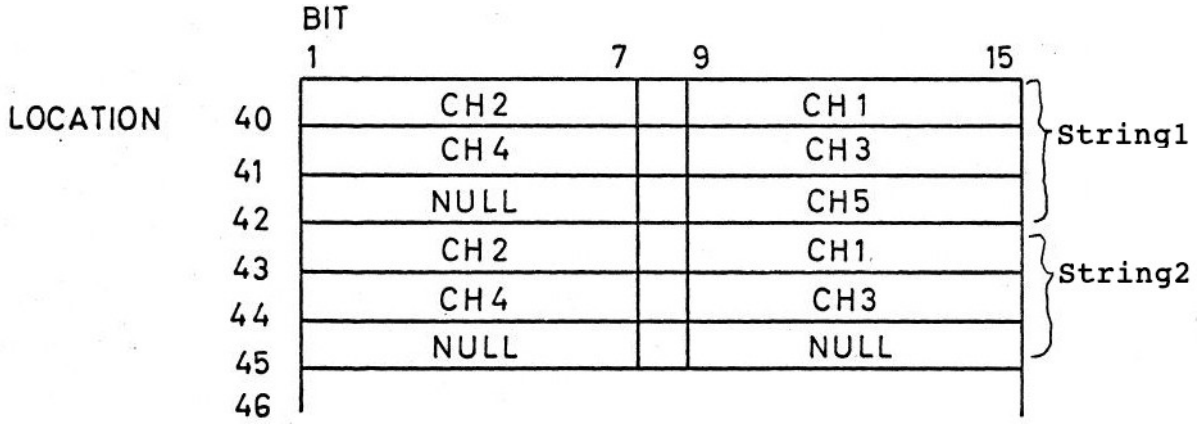


Fig. 9

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 addresses 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 make a reasonably workmanlike end result of a computation, 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 a short signed decimal.
%u	word at next **ADR n** after the **PRF** is output as a short unsigned decimal.
%o	word at next **ADR n** after the **PRF** is output as a short unsigned octal.
%x	word at next **ADR n** after the **PRF** is output as a 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 short-cuts 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 a number of groups of three bits. In our case for a 15 bit machine we would have five 3 bit groups.

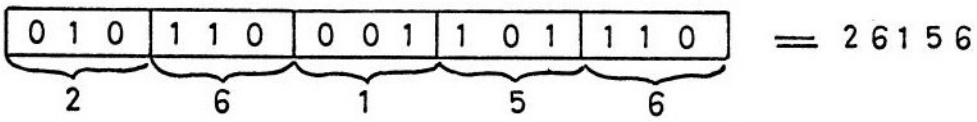


Fig. 10

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 labeling 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 length of 16, 32 and 64 bits, which do not divide neatly by 3. In 1970, when BALAD was first implemented word length 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 labeled 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.

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 address defines the starting address of the block to follow. BLK followed by a DECIMAL displacement will leave a block of memory locations uninitialized, compute the new address and output that as before.

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 optional spaces or the @ symbol if the memory reference is indirect. Lastly type the memory reference of the instruction, either as an absolute OCTAL address, a relative address using **.** or a label 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 starts with 0x followed by digits 0 to 9 and a to f, the number is interpreted as hexadecimal. If the number starts with 0b followed by digits 0 or 1, the number 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.

Another form of constant that may be initialized at Assembly are character strings. 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. After the following ENTER key, the next address after the string is typed.

2.3.1 Comments

In all cases, before typing ENTER, zero or more spaces followed by a **;** 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 shows as 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

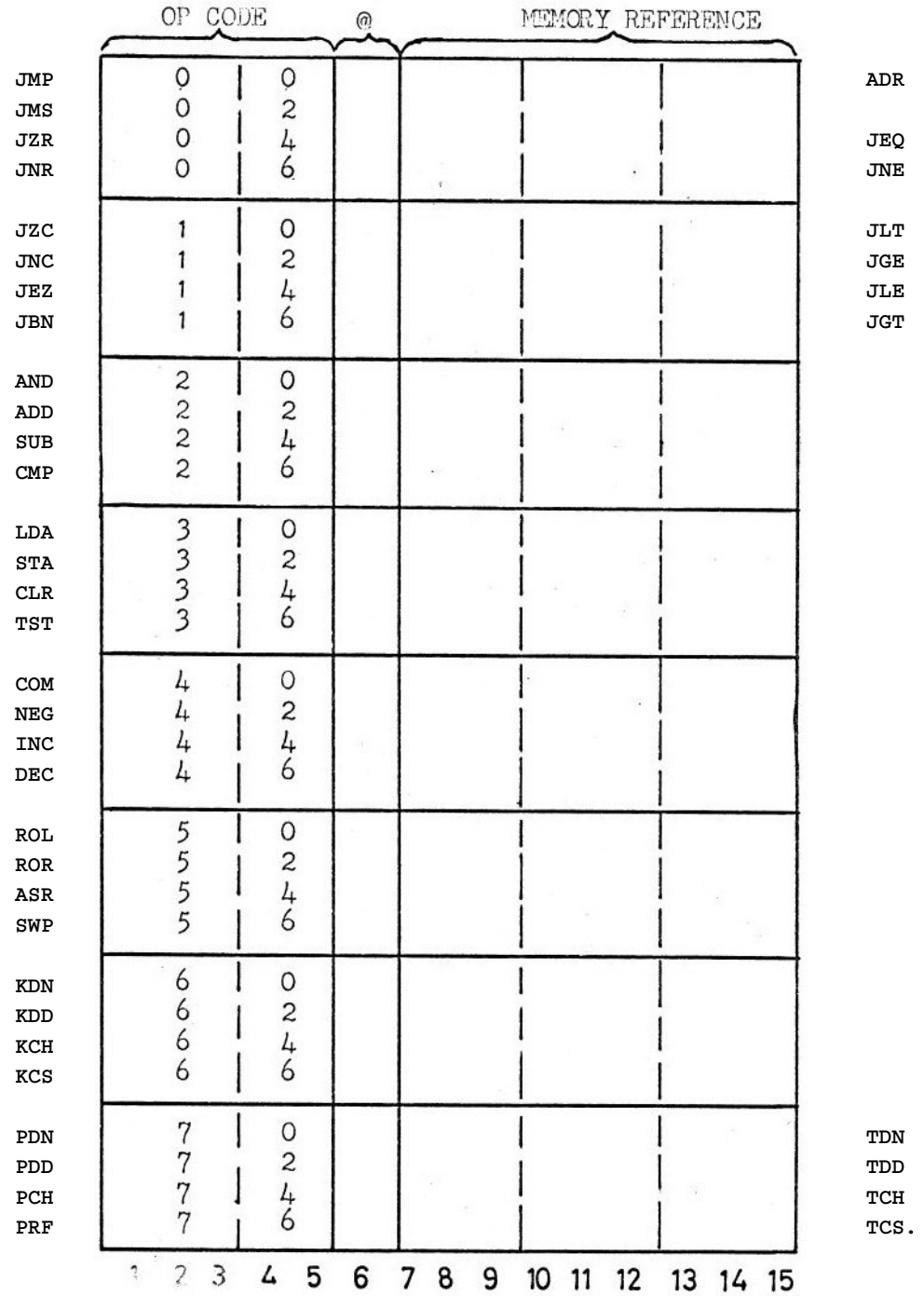


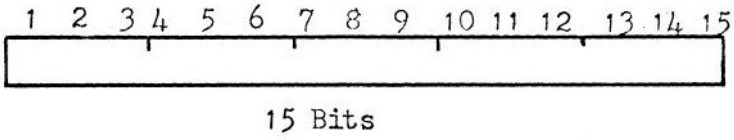
Table 1

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.

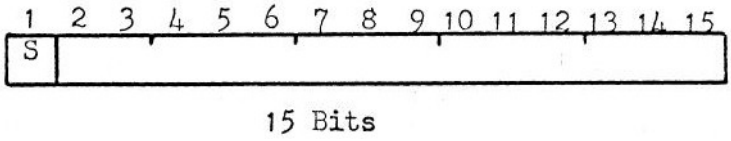
```
CMP dest
JLT .+2     ; jump to .+2 if ACC < contents of location dest
```

2.3.3 Data Formats

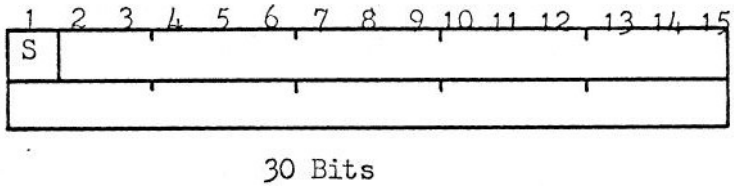
UNSIGNED Integer Number or Logical Unit



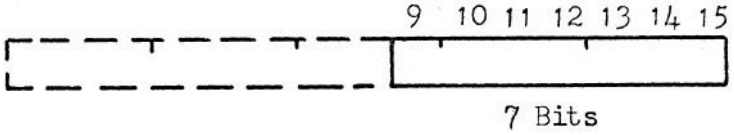
SIGNED Number



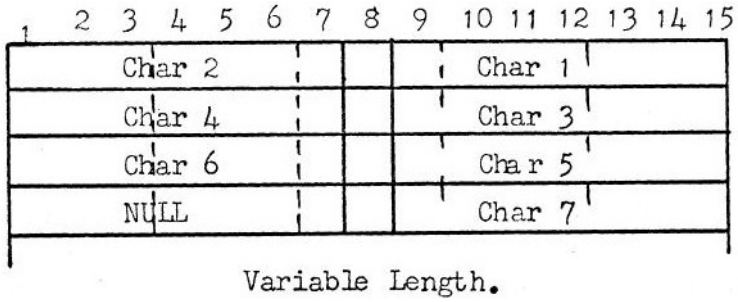
DOUBLE PRECISION SIGNED Number



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 instruction sets and the multitude of special registers 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 CPU's 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 functions. The details can easily be taught in one 45 minute lesson and from then on students can test their own 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 a **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 make it possible to make changes to existing code or enter extra code while debugging, if the semantics of the program are 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 entering 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 character/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 break points 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 BALAD system. which is invoked with one of five command line switches, either alone or in combination. The **-o** switch starts the debugger in *octal data tracing mode*. The normal debugger prompt is the current octal debugger location followed by `100 >>`

```
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 break point `100 B>` **n s u c** or **a** command allowed
- is stopped at a watch point `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
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>  assemble <code> or <data> at current location
<n> < <code>  assemble <code> or <data> at location <n>
              this allows the modification of single locations

    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 for 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 a number of 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**. Adding the **-o** switch, calling the program with **-to** 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. These 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 **break point** or **watch point**, which will be covered next.

4.1.3 Break Points

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 value `00700` in 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 Watch Points

Watchpoints are very similar to breakpoints, but their operation during 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 never executed as an instruction, which means it is not a breakpoint. Instead data locations are read and sometimes modified by different instructions. Each time this happens for data in a watchpoint location (marked by a *) a trace line for the instruction referencing that data is output with the word ***watch*** appended. When the data is actually modified, execution of the program stops and the debugger is re-entered with the watchpoint prompt **502 B>**. The address shown is the address of the instruction whose memory reference location is 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**:

```
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 first two instructions at **502** and **505** the value at **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 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 double precision **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 **n s u** or **c** as for a breakpoint are used.

The way breakpoints and watchpoints work in BALAD is 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 **n**, **s**, **c** and **u** as keyboard accelerators for the **next**, **step**, **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 viewed 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 locations. There are three main ways of interpreting computer memory and a number of 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 single-precision 15 bit integers or 30 bit double-precision integers. BALAD instructions can only work directly on single-precision numbers, but many function algorithms deal wit 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. The '-' can only be displayed with a decimal output. 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'. 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 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. Thus 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 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 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 >>
```

Here we inspect the contents of **stackP**, which is a memory address 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 number as a decimal. As with all listing modes the 5 digit octal memory value for each word is shown also. Any labels are also shown.

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 watch point. The debugger command to do this is **<**. Any text after the **<** command is passed to the assembler to interpret and convert to correct binary value(s) 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. Finally a number of locations can be modified by supplying a “string”. 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 CONCLUSION

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 debugging. Any suggestions or bug reports are very welcome. Please contact me – John Wulff – on:

immediatec@gmail.com with the Subject: BALAD ...