# BALAD
**(Beginners Assembly Language and Debugger)**

## 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.

## THE BALAD COMPUTER

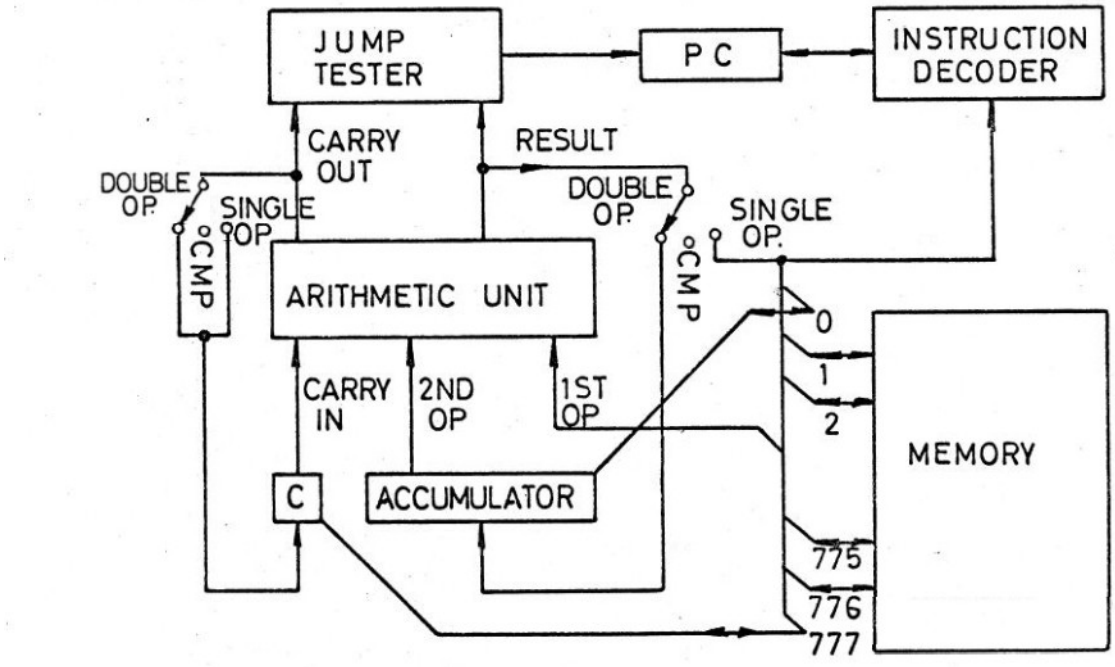



Fig. 1.

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.

# THE INSTRUCTION FORMAT

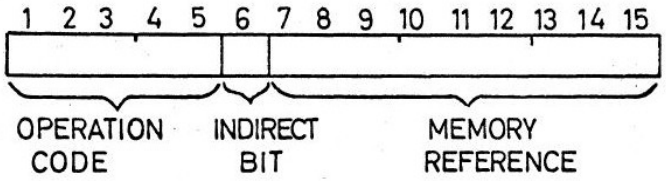



Fig. 2.

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.

## 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.

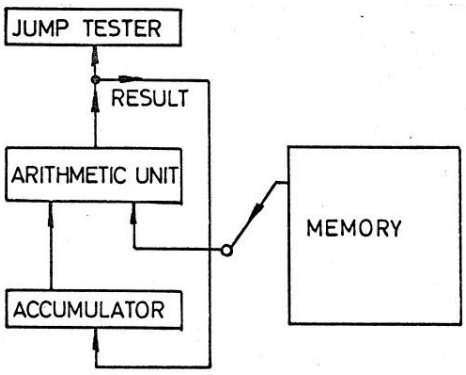



Fig. 3.

## 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.

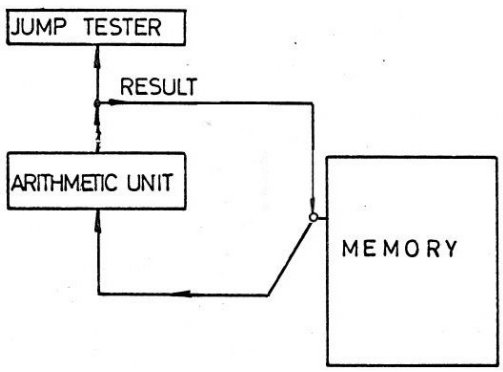



Fig. 4.

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 $1^{st}$ 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 pre-defined label ACC; the carry register has address 777, the highest address and the pre-defined label C.

# THE INSTRUCTIONS

## DOUBLE OPERAND INSTRUCTIONS

**AND** (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** (OP CODE 22)

Add 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. If the unsigned sum is $> 2^{15}$, complement the current value of the Carry register.

**SUB** (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** (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.

## MOVE DATA INSTRUCTIONS

**LDA** Load Accumulator. (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 Accumulatar. (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 contents of the Memory Reference is lost.

## SINGLE OPERAND INSTRUCTIONS

**CLR** Clear (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 (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.

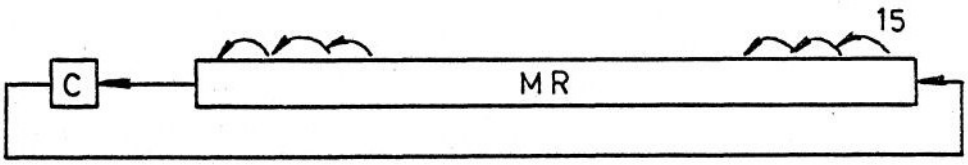
**DEC** Decrement MR (OP CODE 46)

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.

## ROTATE AND SHIFT INSTRUCTIONS

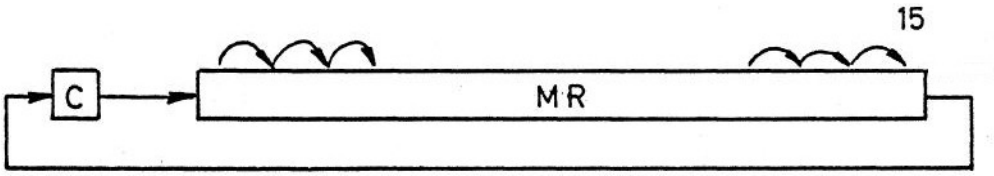
These are single operand operations for bit manipulation, scaling data by factors of 2 and byte manipulation. Rotates are used for testing sequential bits of a word. In each the Carry register and the contents of the Memory Reference are manipulated in different ways, and the result is stored in the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

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



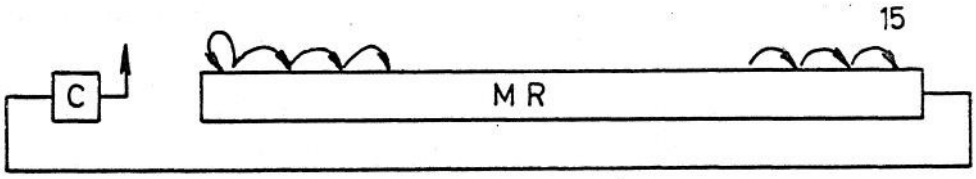

Rotate the contents of the Memory Reference and the Carry Register one bit left as shown in Fig. 5.

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



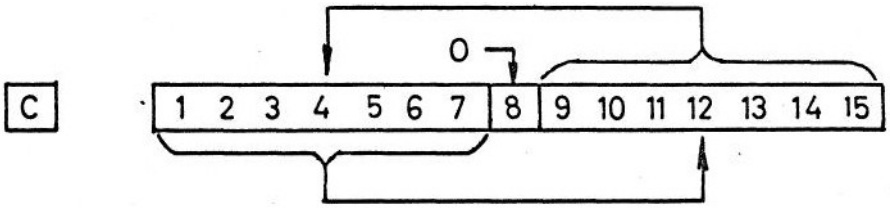

Rotate the contents of the Memory Reference and the Carry Register one bit right as shown in Fig. 6.

**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.

## 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** 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. when following a CMP instruction

Load MR into PC if the last value 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. when following a CMP instruction

Load MR into PC if the last value 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. when 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. when 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. when following a CMP instruction

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

**JBN** Jump if both non-zero (OP CODE 14)
**JGT** Jump if greater than. when following a CMP instruction

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

**JMS** Jump to Subroutine. (OP CODE 12)

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.

## 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 Ctrl D is entered at the keyboard when executing any of the Key input instructions KDN, KDD, KCH or KCS, this will terminate the running BALAD program and return to the debug input >>. Ctrl C will always terminate a BALAD altogether.

## 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 74)
**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:

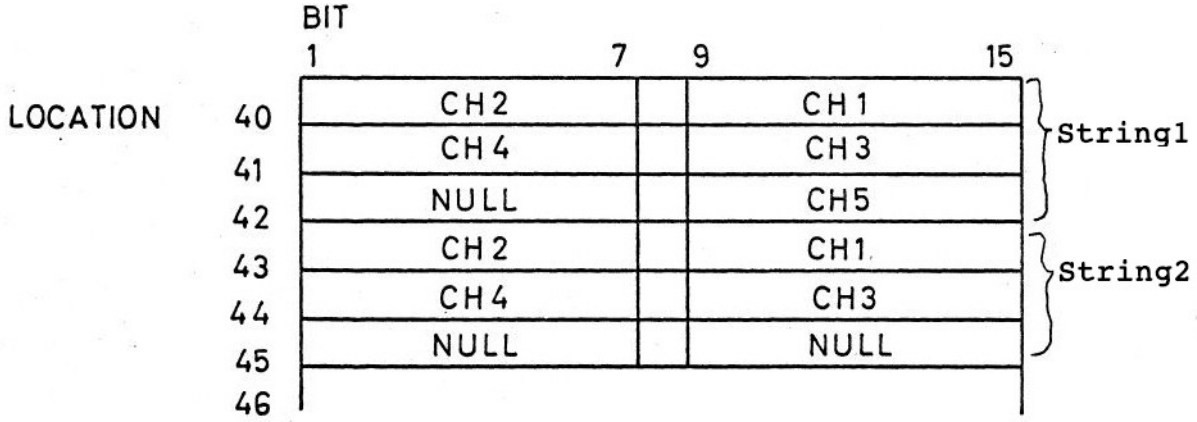



Fig. 9

Character strings may contain page formatting characters of the ASCII alphabet, such as a NEW LINE entered as \n, 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 ADR n after the PRF is output as a short signed decimal. |
| %u | word at ADR n after the PRF is output as a short unsigned decimal. |
| %o | word at ADR n after the PRF is output as a short unsigned octal. |
| %x | word at ADR n after the PRF is output as a short unsigned hexadecimal. |
| %b | word at ADR n after the PRF is output as a short unsigned binary. |

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.

| | |
|---|---|
| %% | print a single %. |

# 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.

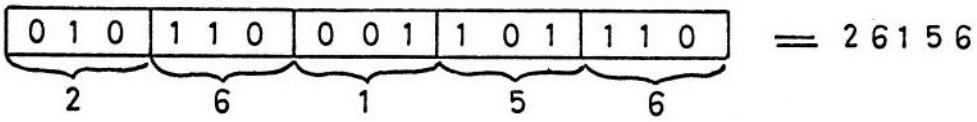



Fig. 10

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 subfields 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].

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.

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.

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 must 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 s symbol table.

It is also possible to refer to another location as a displacement relative to the location occupied by the current instruction. For this purpose the assembler recognizes the symbol **.** (FULL STOP) as the address of the current instruction. The **.** is followed by + or **–** followed by a decimal displacement (all written without spaces). A displacement greater than 7 is inadvisable, because octal arithmetic is involved and inserted instructions should alter the displacement leading to errors. Labels are much better, because they look after all this.

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 terminated with ENTER defines the starting address of the block to follow, which will by typed on the screen followed by 2 spaces. 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. 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, strings or address data for use by indirect addressing instructions[3]. If the first character of a new entry is numeric or an **@**, + or - character the rest of the word must be filled by a single number. For convenience this number may be expressed as decimal, octal or hexadecimal. The convention for modern programming languages is, that a number starting with 0 is interpreted as octal with only digits 0 to 7. If the number starts with 0x or 0X followed by digits 0 to 9 and a to f or A to F, the number is interpreted as hexadecimal. Otherwise it is decimal with digits 0 to 9 only. 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 must 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

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

3 In higher level languages indirect addresses are called pointers.

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.

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

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             sum:    BLK 1   ; uninitialized data block

                        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 numbers (data) or instructions.

## LIST OF INSTRUCTIONS IN OP CODE ORDER

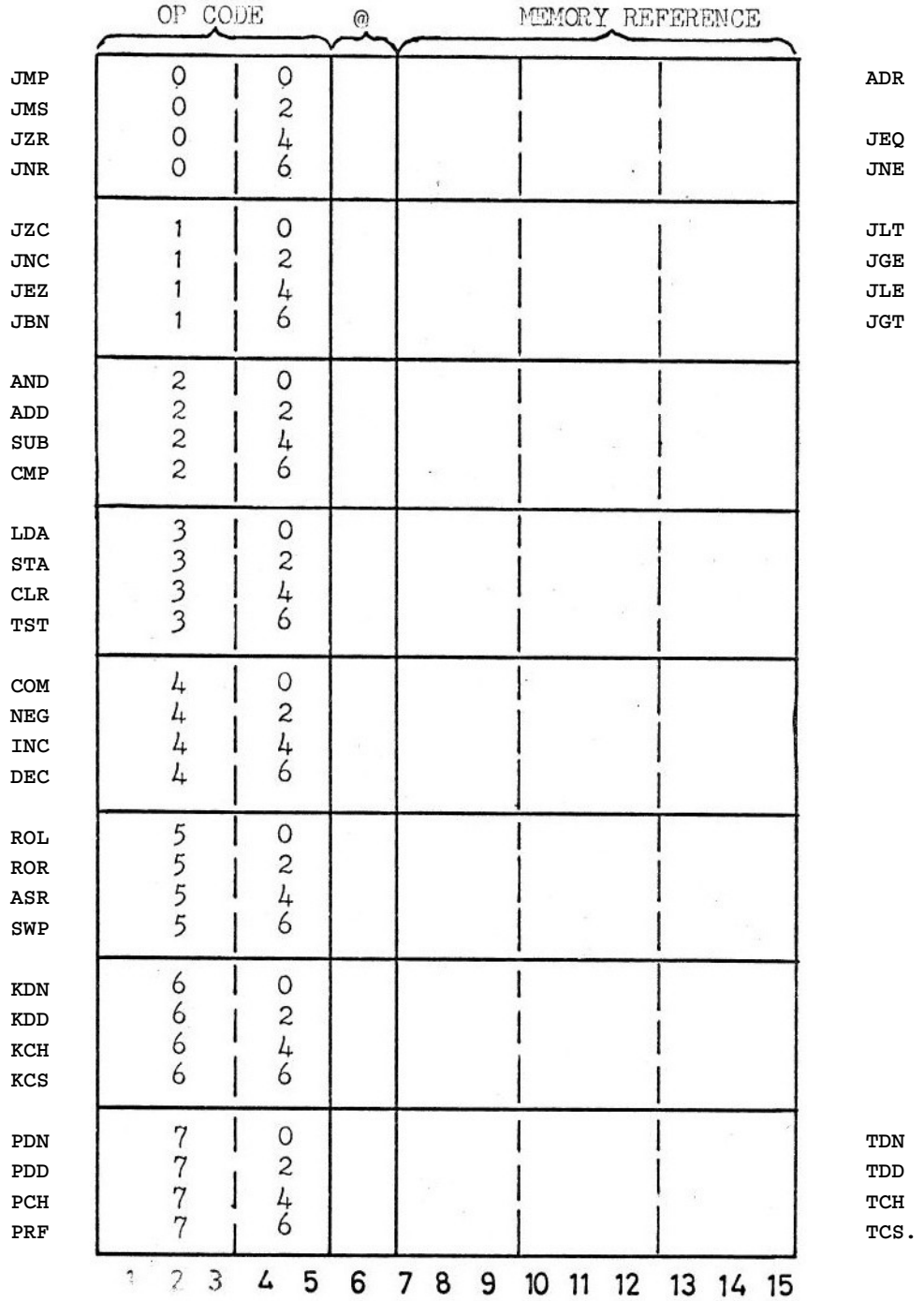
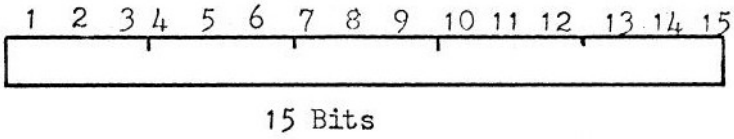



Table 1

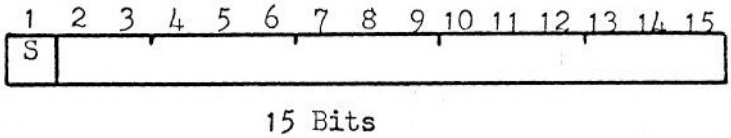
Each 15 bit binary word can be expressed as a 5 digit OCTAL number. For instructions the first two digits are the OP CODE, which is written in assembler code as a three letter mnemonic shown in the left hand column. Some op codes have alternate mnemonics useful for unsigned numerical comparisons. A 1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

## DATA FORMATS

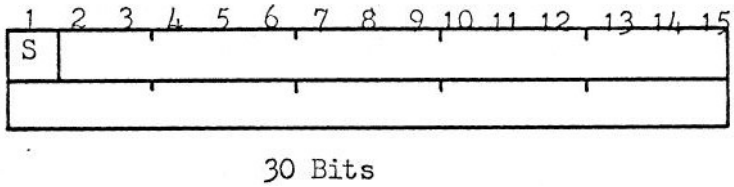
UNSIGNED Integer Number or Logical Unit



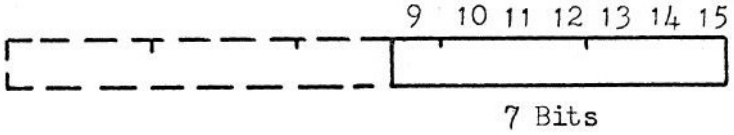

SIGNED Number



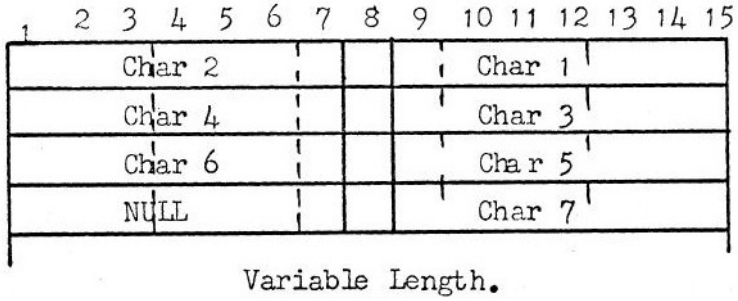

DOUBLE PRECISION SIGNED Number




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.

## 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 and 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.