***Operation Codes*** There are complete descriptions in the *Principles of Operation* section of the TBC web site: https://net/tbc/docs.html

| | |
|---|---|
| ADD | Add operand to accumulator |
| BR | Branch unconditional |
| BNN | Branch on not negative |
| BRZ | Branch on zero |
| CALL | Call subroutine |
| DAT | Define data |
| EQU | Define equivalence |
| HLT | Stop program |
| IN | Read number |
| INC | Increment accumulator |
| LDA | Load accumulator from address |
| LDI | Load accumulator immediate |
| LIA | Load indirect accumulator |
| NEG | Negate accumulator |
| ORG | Set instruction origin |
| OUT | Display number in accumulator |
| OUTC | Display one character |
| OUTN | Identical to OUT but no new line |
| PIO | Programmer I/O |
| POP | Pop stack to accumulator |
| PUSH | Push accumulator onto stack |
| RET | Return from subroutine |
| SSWP | Swap top two stack items |
| SIA | Store indirect accumulator |
| STA | Store accumulator to address |
| SUB | Subtract operand from accumulator |

***Assembling a Program*** You can type your program directly into the source pane of the Assembler page, paste a program edited elsewhere into the source code pane, or upload a program stored on your computer.

When you have the source code in the source pane, type a name in the block next to “Save as” and click or tap “Save as.” The program is stored in your browser’s local storage.

Click or tap “Assemble.” If there are no errors, the binary object code is also stored. You can load the Virtual Machine page and test.

***Operation*** Load the assembled program by selecting its name in the load control, then click or tap the “Load” button. You can run the program by clicking/tapping “Run.” The “Clock” slider adjusts the clock speed.

The “Pause” control stops execution at the end of the current instruction. “Step” runs one instruction. “Cycle” runs one clock cycle.

“Reset” resets the stack pointer and program counter, but *does not* reinitialize values initialized by DAT.

“Show output” and “Show trace” switch the display contents between program output and trace.

***The DAT Pseudo-Instruction*** DAT is not an operation code; it directs the assembler do reserve space and potentially to initialize it. The simplest use is:

```
nbr   dat
```

That allocates one word of space with the label “nbr.” The contents of such storage are unpredictable until a value is stored there. Storage can be initialized by giving the initial value as an operand to DAT.

```
nbr   dat   0x0a
```

Initial values can be decimal or hexadecimal in range −2,048 to +2047. Hexadecimal values are indicated by a leading 0x.

The DAT pseudo-instruction can take a quoted string or a comma-separated list of values.

```
hello dat “Hello, World!”, 0x0a, 0
```

In the example above, 0x0a is the new line character, and the 0, a null character, by convention, terminates the string.

You can get *some* Unicode characters into string literals by using backslash-U and four hex digits. Only code points from \U0000 to \U00FF are supported because only eight bits of the 12-bit word are used to store a character. You must have exactly four hex characters even though the first two will always be zero.

.

***Comparison*** High-level languages have IF statements, but TBC, with only 16 possible operations, does not. Instead, subtract comparands.

Comparison for Equality:

```
lda   a     // First comparand
sub   b     // subtract second
brz   equal // branch if equal
```

If variables “a” and “b” are equal, the branch will be taken. If not, the instructions following the branch will be executed. This opposite the usual “if equal” statement.

Comparison for Greater Than or Equal:

```
lda   a  //Load first comparand
sub   b //Subtract second
bnn   gteq // Branch if A >= B
```

The branch is taken if variable “a” is greater than or equal to variable “b.”

Comparison for Strictly Greater Than:

```
lda a //Load first comparand
sub b //Subtract second comparand
brz not //Equal so not strictly >
bnn gteq // Branch if A > B
```

We must take into the case where the comparands are equal. That happens in the third line of the example. The last line branches to the case that “a” is strictly greater than “b.”

**Comparison for Less Than:** To check for l*ess than*, use the patterns above, but reverse the comparands.

***Address Arithmetic*** The assembler can perform addition and subtraction on address fields. Operands can be decimal or hex constants, defined labels, or the symbol * which represents the current location. There must be ***no spaces*** in the expression. To find the length of a data structure at assembly time:

```
data    dat     1, 2, 3
datalen equ     *-data
        ldi     datalen
```

The expression on line 2 calculates the length of structure “data.” It can be used as a constant, as in the Load Immediate on the last line.

***Iteration*** Looping over a string or array is accomplished using a pointer variable and the LIA (Load Indirect Accumulator) instruction.

```
      ldi  data  Get add of data
      sta    ptr     save it
loop  lia    ptr
// check for end of data
// Do something with data
      Lda    ptr
      inc
      sta    ptr
      Br    loop
```

Checking for the end of data depends on whether you are looking for a terminating null or counting characters. For a terminating null, a BRZ instruction right after the LIA will work. If you are counting, check the counter to the ending value.

# At a Glance

**https://tbc.kennesaw.edu/**

***Instruction Format*** Each line of assembler code must have the following format:

```
[label]  opcode  [address]  [comments]
```

Items in square brackets are optional. The label, if present, must begin at the left, with no preceding white space. The operation code must be preceded by white space regardless of whether a label is present.

Anything following the address, or for operation codes that do not take an operand, anything following the operation code, is treated as a comment.

Labels, operation codes, and symbolic addresses are case-insensitive. Labels and symbolic addresses are not length limited. For valid operation codes, see Instruction Set, below.

The first executable instruction must be at location zero, as it will be by default because assembly starts at location zero.

***Organizing a Program*** Because the first executable instruction must be at location zero, organize your programs with instructions first and *data at the end.*