3AA - Assembler Interpreter

A manual for an interactive assembler interpreter

Prof. Dr. W. Kowalk

Fachbereich 10 Informatik, Carl-von-Ossietzky-Universität Oldenburg

Version 2.1 – August, 2002

This document describes usage and application of the interactive assembler interpreter 3AA – Three Address Assembler.

Introduction

An assembler is used to describe instructions of a processor. This assembler is a three address assembler which uses no registers. Thus quantities can be stored in memory only.

Its main application is to teach the behavior of a simple computer, which has been shown to be equivalent to a Turing machine.

The Interface

The interface consists of a single screen with several areas, the main area, the Control of Memory Values, and four other areas with one edit window each. An additional interface is a statistic window, which displays the number of statements accomplished during program execution.

Main area

The only edit window of the main area is used to display the file name path, which contains the program to be loaded or stored. Initially a name is inserted, which can be used as test file for easily loading and saving the program under development. Using the Search-Button allows to change the contents of this window by an interactive dialog.

The meaning of the buttons in the main area is as follows:

Compile	Compile the program as being placed in the edit window labeled ***Input program code***. If errors are detected they are reported one by one in a popup-window. The user can stop compiling immediately by pushing the appropriate 'NO'-button.
Load	Load the program into the assembler's memory. The program must have been compiled with no errors before. Loading is required to initialize the memory as being defined by the program text. If the user re-executes a program with no load, the results might be unexpected. Execution starts at the program line specified in the field ***Start at line***. If this entry is not adequate, the load command asks to set it to the first executable line of the program. It is strongly recommended to start programs at the first line of the program text (after some lines of comments, which are always ignored).

Step	Execution mode **step-by-step**. Runs the program by one step. The program counter is always displayed for the statement which will be executed next. The output of this popup-message is 1. the number of the statement (counted from 1 with the first statement of the program text), 2. the line number of the program text, 3. the label of this line, it there is any. 4. the latest change of a memory value by presenting this memory's old and new value, the location of this memory (using line number and label, if present) and the statement which changed this memory 5. Selectors for continuation. Pushing the appropriate button 'Yes' can continue program execution for one statement. Pushing the appropriate button 'No' can continue program execution with no interrupt. Selecting the appropriate button 'Cancel' can stop execution.
Run	Execution mode **run**. Runs the program until a 1. '`stop`' statement is reached, 2. a '`step`' statement changes execution mode to step-by-step, 3. or a runtime error occurs. Runtime errors are a. access of illegal memory space, b. trying to execute a line of code which is not a statement.
Format	If the user pushes this button all line numbers and indents are removed from the program. A popup-windows appears, which asks whether indents should be inserted, and after this whether line numbers should be inserted. The compiler ignores a number as first token in a line and any indents by blanks or tabulators.
Stats	This button opens an additional dialog where statistical data of the program runs are displayed. Each executed instruction is counted, where intermediate and total sums are displayed. Also the sum of all previous runs is cumulated, which can be cleared by the Clear cumulated button. Using the Transfer button, the data of the last run can be stored and so easily compared to data of previous runs. Using the Save button, the results are stored in a file with the same path as the program file, using the extension ".sta". The Close button shuts down this dialog.
Close	This button finishes the assembler program. Save your data before, otherwise it's lost.
Search	This button opens the windows interactive Load File dialog. A selected file name path is stored into the file window of the main area. No load or save of files is performed. For this the buttons in the corresponding areas are to be pressed.
Open Program	The program the file of which is specified in the edit window above this button is loaded into the window labeled ***Input program code***. All previous input is overwritten. No security question is made, so all previous content that is to be preserved must be stored, using the 'Save Program' button. The content of the clipboard is preserved.
Save Program	The program in the window ***Input program code*** is saved into the file which is specified in the edit window above this button. No security question is made, so the content of the file is overwritten and cannot be restored. The user should be careful not to destroy valuable data.

Control of Memory Values

The meaning of the buttons in the area labeled 'Compiler Constant Memory Values' is as follows:

Open	The control value data the file of which is specified in the edit window above this button is loaded into the edit window of this area. The file name is changed, where the extension '`.asm`' is replaced by '`.ctl`'. All previous input is overwritten. No security question is made, so all previous content that is to be preserved must be stored, using the 'Save' button. The content of the clipboard is preserved.
Save	Saves the content of the edit window of this area using the file name which is specified in the edit window above this button, where the extension '`.asm`' is replaced by '`.ctl`'. The content of this file is overwritten and cannot be restored. So the user should be careful not to destroy valuable data.
Change	If this button is set (which is the default value) the identifiers in the edit area which correspond to a compiler constant of the program are altered during execution by appending information about the value of this corresponding compiler constant. If this button is cleared, the edit area is not altered during execution of the program, however the step button removes earlier simple output.
Replace	If this button is set (which is the default value) the identifiers in the edit area which correspond to a compiler constant of the program are altered during execution by appending information about the value of this corresponding compiler constant. Existing values, which have been altered by previous statement executions, are overwritten. If this button is cleared, existing values, which have been altered by previous statement executions, are preserved, so the user can trace the change of variables.
Label	If this button is set (which is not the default value) control value output is appended by parentheses enclosing the label of the statement, which has changed this variable, and also the count of executed statement. Thus in case of several value changes the sequence of changes can easily be derived. If no label is given, the line number is displayed instead of the label.
Line	If this button is set (which is not the default value) control value output is append by parentheses enclosing the line number of the statement, which has changed this variable, and also the count of executed statement. Thus in case of several values the sequence of changes can easily be derived. If the button 'Label' is active as well, this button has no effect.
Show Memory Address	If this button is set (which is not the default value) the memory address of a compiler constant is appended to the name, enclosed in "[..]".

Program Size

This displays the number of program statements of the program in the program code window. This value is set by the compiler (and valid only after the '**Compile'** button has been pushed). A statement is either an instruction or a memory variable or a compiler instruction:

```
adr Name := val Name + con 1  // example of an instruction
Wert con 0                    // example of a memory variable
Name := con Wert + con 1      // example of a compiler instruction
```

Memory Size

This displays the memory required to execute the program. Its value is used by the '**Load'** button. Sufficient memory has to be supplied to implement stacks, heaps and other dynamic memory space. During execution the memory size is checked and a run time error occurs if not sufficient memory is supplied.

Start at line

The interpreter starts execution at this line in the program text. If its value points to a line that does not contain an executable statement a warning is displayed.

Program Counter

The current value of the program counter during execution is displayed.

Syntax of input language

This program interprets a Three-Address Assembler language. It is not intended to emulate any existing assembler.

1. A program consists of a sequence of statements.
2. Each statement is written on one line.
3. Each statement consists of an optional (line) number, an optional label, and an instruction.
4. A statement is either a compiler instruction, a memory instruction, or a processor instruction.
5. A **compiler instruction** consists of a label, an assignment symbol and an expression,
   `compiler instruction :: label ":=" expression`
   where the expression can be evaluated during compile time and the value of the label becomes the value of this expression.
6. A **memory instruction** consists of an optional label and an expression,
   `memory instruction :: [label] expression`
   where the expression can be evaluated during compile time. By the 'Load' command the value of the expression is stored into the memory at this location before execution. The value of the label becomes the program counter's value of this instruction. Since this is no processor statement, this instruction must never be executed.
   Memory instructions can be used to initialize memory with values and label it to access these values (and also change it; thus without 'Load' this values may differ at different runs.)
7. A **processor instruction** is either an assignment, a goto instruction, a conditional goto instruction, or a nop-, step-, or stop-instruction:
   ```
   processor instruction :: assignment
   processor instruction :: goto-instruction
   processor instruction :: conditional-goto-instruction
   processor instruction :: nop-instruction
   processor instruction :: step-instruction
   processor instruction :: stop-instruction
   ```
8. An **assignment** consists of an assignment variable, an assignment symbol and an expression.
   `assignment :: assignment-variable := expression`
   The assignment symbol is '`:=`'.The operator '=' is never used in this assembler. During program execution, the value of the expression is evaluated and stored into the memory identified by the assignment variable. After this, the next statement is executed.
9. An **assignment variable** consists of one or two address keywords `adr` and a variable name.
   `assignment-variable :: [adr] adr name`
   The meaning of `adr name` is: Store the result under the address `name`, where `name` is a compiler constant.
   The meaning of `adr adr name` is: Store the result under the address `adr name`, where `name` is a compiler constant. Thus the value at memory location `name` is used as the address where the result is to be stored. We call this **indirect addressing**.

10. An **expression** consists of one or two operands with one of the following operators.
    ```
    expression :: operand ( + | - | * | / | % | >> | >>> | << |
                            '|' | & | ^ | ~ ) operand
    ```
    with the well known meaning add, subtract, multiply and divide. % means: remainder of division. >>> means shift right number of Bits, same as << for shift left; >> means shift right and copy minus sign (highest order Bit), | means bit wise logical OR, & means bit wise logical AND, ^ means bit wise logical XOR, ~ means bit wise logical XOR of both operands and NOT of each Bit (i.e. 1's complement).
    Operands consist of either the key word `con` or one or two of the key words `val`, followed by a name.
    ```
    operand :: ( con  |  val  |  val val ) name
    ```
    The meaning of `con name` is: The result is `name`, where `name` is a compiler constant.
    The meaning of `val name` is: The result is the value found at memory location `name`, where `name` is a compiler constant; we call this **direct addressing**.
    The meaning of `val val name` is: The result is the value found at memory location `val name`, where `name` is a compiler constant. Thus the value at memory location `name` is used as the address from which the result is to be obtained. We call this **indirect addressing**.
11. A **goto-instruction** consists of the key word goto followed by an operand.
    ```
    goto-instruction :: goto operand
    ```
    Execution of this instruction sets the program counter to the value of the operand. The next instruction is executed at the corresponding memory location.
12. A **conditional goto-instruction** consists of the key word `if`, a Boolean expression, the key words `then goto` and an operand.
    ```
    conditional goto-instruction :: if booleanExpression then goto operand
    ```
    A Boolean expression consists of an operand, a compare symbol and another operand.
    ```
    booleanExpression :: operand ( < | <= | > | >= | == | >< ) operand
    ```
    The operands have the obvious meaning. The symbol == means identity (or same values). The operator '=' is never used in this assembler.
13. An instruction with **no function** consists of the key word nop.
    ```
    nop-instruction :: nop
    ```
14. An instruction that switches to the **step-by-step-execution mode** consists of the key word step.
    ```
    step-instruction :: step
    ```
15. An instruction that **stops execution of the program** consists of the key word stop.
    ```
    stop-instruction :: stop
    ```

Data types during compile time

Integers are stored into the memory by compile time expressions as

```
Labelname con 1 // stores value 1 in a memory cell the address of which …
                // … can be accessed via the compiler constant "Labelname"
con Labelname   // stores the value of the compiler constant "Labelname" …
                // … at the current memory cell
```

In Windows and Java integers have a size of 64 Bits. Two's complement is used to store negative values. Overflow during arithmetic operation is not recognized.

Characters are stored in the memory by compile time expressions as

```
con 'x'  // stores the ascii value of 'x' (=120) at the current memory cell
con 'ä'  // stores the ascii value of 'ä' (=228) at the current memory cell
```

In Windows characters have a size of 8 Bits and the ascii alphabet is used. Characters are interpreted as unsigned integers. In Java characters use Unicode.

To store a sequence of characters, texts can be used.

```
con "Symbol"  // stores the ascii values ‘S’, ‘y’, ‘m’, ‘b’, ‘o’, ‘l’ …
              // … into six consecutive memory cells
```

Text must not be empty!

```
con ""  // not allowed!!!
```

Data types during program execution

Program execution interprets all values as signed integers. Only integer operations like ‘+’, ‘*’ etc., bit wise logical operations like shift, AND etc., as well as integer comparisons are allowed.

Interpretation of results

To monitor the values of the memory, the user can insert compiler constants into the area labeled ‘**Control of memory values**’. It is also allowed and recommended to insert any other text into this area, to help understanding the meaning of the compiler constant names, but only compiler constants are extracted and their appending values are changed.

After each execution of an instruction the new values of the memory (if any) are appended to the corresponding compiler constant name by the expression:

```
Name=NewValue
```

It depends on the switch ‘Replace’, whether previous values are overwritten. Also, the statement that has changed this value can be displayed by pushing the button ‘Label’ or ‘Line’. Its syntax is:

```
Name=NewValue(LabelOfInstruction,SequenceNumber)
```

SequenceNumber counts the number of changes of memory values. In case of several compiler constants to be traced this helps to pursue the sequence of execution.

This area can easily be saved by the ‘Save’ button and retrieved by the ‘Open’ button, so pushing the ‘Open’ button can rescind any change by execution of the program.

Plotter

The Variable PLOTTER (which is a keyword in 3AA) has a special functionality. It plots a graphical panel, on which lines and text can be drawn as known by plotters. There are the following instructions:

```
adr PLOTTER := con -1 // clears the plotter (a new sheet)
adr PLOTTER := con 40 // sets the line size to 4 points.
adr PLOTTER := con 106  (see list below).
adr PLOTTER := con 0 // no drawing till next point
adr PLOTTER := val x , val y // sets the pencil to (x,y);
PLOTTER   con 0 // Plotter variable must be defined!
```

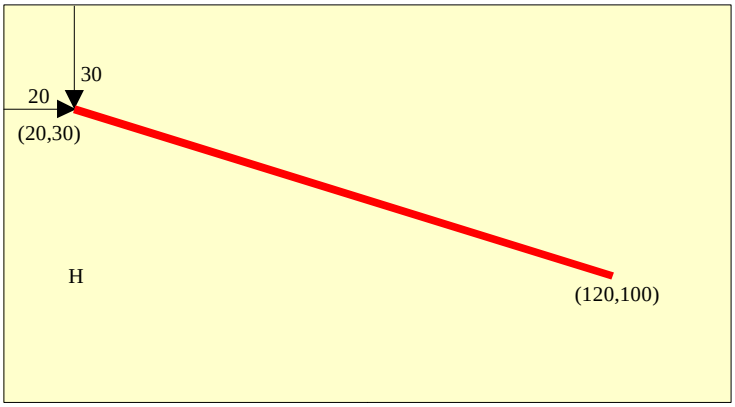
The Plotter panel can be cleared by a negative assignment. A positive number from 0 to 100 determines the size of the next line, where 0 means no drawing until next point. A number greater than 100 is interpreted as a color, where

Index	***Color***	***Index***	***Color***	***Index***	***Color***
101	black	106	green	111	red
102	blue	107	lightGray	112	yellow
103	cyan	108	magenta	113	white
104	darkGray	109	orange	>103	black

Index	Color	Index	Color	Index	Color
105	gray	110	pink		

The plotter area is an x-y-coordinate system, where the point (0,0) is the left upper corner. Assignment of a comma-expression sets the pencil to the corresponding coordinate. To print the following figure you write

```
PLOTTER con 0          // A Plotter variable must be defined!
adr PLOTTER := con -1  // clears the plotter (a new sheet)
adr PLOTTER := con 80  // sets the line size to 4 points.
adr PLOTTER := con 111 // sets the line color to red
adr PLOTTER := con 20 , con 30 // sets start of plotter to (20,30)
adr PLOTTER := con 120 , con 100 // draws a line to (120,100)
stop
```



Also, text can be written by using the operator dollar (`'$'`). To set a character use a negative first parameter; plot the character by use of two positive coordinates, and the dollar operator:

```
adr PLOTTER := con -1  $ con 'H' // push character 'H' to output channel
adr PLOTTER := con 120 $ con 210 // plot content of output channel to (120,210)
```

prints the character 'H' at the position: (120, 110). Color and size of the text is defined in the same way as color and size of a line, where size is the point size of “Monospaced”.

Future extensions

planned:

1. The compiler constant PC (programming counter) can be used to display the execution of the program in the control window. However, line numbers are to be displayed.
2. Further types: floats.
3. Macros, with or without parameters

Known bugs

Not yet known any.