3AA - Assembler Interpreter

A Programming Introduction for Newbys

Prof. Dr. W. Kowalk

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

Version 1.1 – August, 2002

This document describes programming of an interactive assembler interpreter 3AA – Three Address Assembler: 3AA. All programs are presented explicitly, and can be copied into the interpreter.

Introduction

An assembler is used to describe instructions of a processor. This assembler is a 3 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.

Lesson 1: The first example

The first program shows a simple computation and the display of results. It's no "Hello World!"-Program.

Open the Interpreter by clicking its link. The screen appears. Insert the following text in the Edit Window labeled "`Input Program Code`":

```
1           // Arithmetic expression Result := 6*(4+5)
2 Start     adr Aux1 := con 4 + con 5
3           adr Result := con 6 * val Aux1
4           stop
5 Result    con 0
6 Aux1
```

This program computes the arithmetic expression, displayed in the first line.

The first line starts with a **line number** (which helps to understand the program, but has definitely no meaning to the interpreter). Two slashes (`//`) start a **comment**, which is arbitrary text until the end of the line. The text (including the "`//`") has also no meaning to the interpreter. It only may help a human reader (particular your tutor) to understand the program. The spaces between the line number and the comment have also no meaning to the interpreter. One space is sometimes necessary to separate two words (which are usually called "***tokens***"), but otherwise spaces have no meaning to the interpreter.

Line 2 starts with a line number (again: with no meaning to the interpreter) and a label. This label "`Start`" gets automatically the number of the memory cell where the statement (or variable) in this line is stored. Its absolute value should be of no concern to the programmer! (Never, never use absolute addresses in your programs. Never!)

Line 2 holds a complete **statement**, which is here called an **arithmetic expression**, since a numerical computation is performed. The first token "`adr`" tells the interpreter to use the following **memory address number** (here "`Aux1`") to store the result of the computation, which stands on the right hand side of the

assignment symbol "`:=`". This memory address number "`Aux1`" is a constant which is defined in line 6 (look at the program!). Yes, you only write a **Name** on a line (besides an optional line number, which has definitely no meaning!), and the memory cell is reserved for an integer variable, where this **Name** is now a synonym for the address of that memory cell. This **Name** must be unique! But the compiler tells you, if its not.

Following the assignment symbol "`:=`" stands an arithmetic expression. The token "`con`" tells the interpreter that the following constant (it must be a constant!) is taken as a value. The expression "`con 4`" means the quantity four. That's simple, isn't? Congest what the next expression "`con 5`" means! If your answer is five, it's quite right. If it isn't, think again! Well, and the plus sing "`+`" means: add those two values. The first expression results to nine. And this result is stored at the memory cell labeled "`Aux1`". That's all you have to understand. That wasn't difficult, was it?

The next line with the line number 3 shows almost the same: An arithmetic expression, where the result is now stored at the memory cell labeled "`Result`". There is only one new concept. The second operand of the operator (which is now "`*`" or times) has the token "`val`". This token tells the interpreter to take the following constant (it must be a constant, this time a compiler constant) as a memory address; the quantity that is to be taken to compute with is now the value stored at the memory cell (here the memory cell labeled "`Aux1`"). You remember, we have just stored a value there, namely nine. Now we retrieve it to use it in another computation. That is practical, isn't it? You can compute something, store it by a well defined label, and get it back whenever you want. That is the concept of data mining (in a very simple environment, of course ✍). Well the interpreter computes now six times nine (the value stored at the memory cell "`Aux1`") and the result is stored at the memory cell labeled "`Result`".

The next line with the line number 4 has only one statement, which is called "`stop`". Well, it does what it tells you. It stops execution of the program. That's all. Nothing more to be said about.

You might have guessed, but I will tell you here: A program is executed one statement after the other. It starts (if you don't tell it something else) in the first executable line, which is here line 2, and proceeds to the next line (you can tell the program to do others, but you have to do it explicitly). That is all you have to understand in this first lesson. That's not too difficult!

But of course you are interested to see how the program is executed, won't you? Well, the edit field called "`Compiler Constant Memory Values`" is used for this task. Again, it is as simple as it can be. You can write into this area whatever you want. However, if you write the name of a memory label ("`Aux1`" or "`Result`" in our example program), the interpreter appends the contents of the corresponding memory cells to these names. Write therefore those two names into that area. Do it now, don't hesitate. The "`Compiler Constant Memory Values`" edit field should now look like

```
Result
Aux1
```

It is a wise decision to store your program now before going further. This is done by searching for a valid directory and give your program a name, that ends with ".asm", e.g. "`Arithmetic1.asm`". Then press the buttons "`save program`" and "`save`" in the Compiler Constant Memory Values area. By pressing the corresponding open buttons, you can retrieve these texts whenever you want.

Now, the next step is to compile the program. "**Compile**" is used as a technical term, which analyses a program and generates an intermediate code. Do it now, press the button "`Compile`". If you've done what I told you, compiling should end with a congratulation. This means, that no syntactical errors are found (don't worry about the notion "`syntactical`". It is a terminus technicus, you will understand better later). There can still be logical errors in your program, for example multiple definitions of constants (i.e. the same name with different values). This type of logical errors can your compiler find, and it will do in the next phase. That is simple for it. However, if you make logical errors in your program, your compiler cannot find them. Thus, if no syntactical errors are found, this does not mean that your program is

“correct”. It only means, the compiler has done its very best and looked for any possible type of error it can detect. If it didn’t detect any, the rest is up to you.

If you did not mistype anything, the compiler will end up with no further comment. Then you can load your program. This means to load the statements as well as other memory content, like predefined values, into the memory. Do “Load” before each execution. Don’t forget it, wont you! 3AA asks you now where to start, and it also makes a good suggestion, which you should accept (i.e. click “Yes”). You always must start on an executable memory cell.

Now you can run the program. Don’t be shy. Press the “Run” button. It won’t hurt you. What do you see? Well, if you have observed that the display of the Memory constants shows something, you have good eyes. It should now look like:

```
Result=54
Aux1=9
```

If you didn’t know before what six times nine yields, here you got it. It seems to work, doesn’t it? All’s well, that end’s well!

This ends your first lesson. You have learned how to input a program, save and open the programtext, compile, load and run the program, and display the results.

Lesson 2: Compiler Constants

Compiler Constants aren’t that difficult. You have learned almost everything you must know about them. They are names. A name is a sequence of characters, starting with a letter (like ‘A’, ‘D’, ‘X’) and followed by another letter or cipher. Thus “Start” is a correct name, as well as “Aux1” or “Label1234”. But “1Ende” is no correct name, since it doesn’t start with a letter. Of course, there must not be any other character within this sequence, particular no space. A name is what you might have expected, so don’t worry about it.

A compiler constant is a name that has a value, which is a number like 0, 1, 122 or 8888. Instead of an integer number you always can write a compiler constant. That is more useful, since a well named compiler constant tells the reader what it means. The value of a compiler constant defined like this

```
Label … // something to be stored into a memory cell
```

is always equal to the memory cell, which is defined on the same line. (remember, the text from “//” to the end of the line has no meaning to the interpreter. This text is called a comment). This can be a cell to remember a value (like a number) or it can be a statement. To define a memory cell for a number you can write something like this:

```
Label0          // defines a memory cell with initial value 0
Label1 con 256  // defines a memory cell with initial value 256
```

Thus you might guess how to define a memory cell with initial value thirty-three and a compiler constant named “Price1” the value of which is the number of that memory cell. It should look like

```
Price1 con 33   // defines a memory cell with initial value 33
Tax    con 13   // defines a memory cell with initial value ??
```

What will be the initial value of the memory cell at “Tax”. If you guessed thirteen you are quite right, if you didn’t, restart this lesson from the very first line. Sometimes it is confusing, but what we are defining here are two things. In the last example “Tax” is the address of a memory cell. The initial value at this address is thirteen. These are two completely different and independent things. Please, don’t muddle up. Type the following program, compile, load, and run it,

```
1           // Compiler Constants
2           stop
3
4 Label0
```

```
5 Label1    con 0
6 Price     con 33
7 Tax       con 13
```

and you will get the result:

```
Label0=0
Label1=0
Price=33
Tax=13
```

It is a custom to call such a memory address a **variable**. Thus a variable is something that can have variable values, where its name comes from. Using such a variable name, we can store and retrieve values that are required for some computations. Thus, when we call something a variable we mean a memory address. The value of a variable is the value at he memory place (and not the address). Try to make this quite clear. It will become your daily bread.

Now, we come to the simpler part. Sometimes a constant is required, the value of which is not a memory address. You can define them in 3AA as well. Write

```
1           // Compiler Constants 2
2
3           stop
4 Value1    := con 243
5 MagicNumber := con 42
```

compile, load, and run this program and you have all you want.

```
Value1=243
MagicNumber=42
```

That is much simpler, isn't it? Now you have a compiler constant the value of which is explicitly defined. It is no address; so don't use it as an address. It will cause problems while executing the program.

If you are really interested in the values of a memory address you can display them as well. A program that does so might look like:

```
1           // Compiler Constants
2           adr Output := con Price
3           stop
4 Label0
5 Label1    con 0
6 Price     con 33
7 Tax       con 13
8 Output
```

with the results:

```
Label0=0
Label1=0
Price=33
Tax=13

Output=4
```

But it is completely unnecessary to know the absolute value of any memory address. So forget it!

There are some things you should know to simplify editing. Your editor can indent and set line numbers automatically. Test the button "`Format`", and everything becomes clear.

To copy a text, select it by putting the cursor on the first symbol and draw it to the last one, by keeping the left mouse button pressed. Then press <strg><C> to copy, <strg><X> to cut and <strg><V> to insert. If you are used to usual editors, this should make no problems for you. Using the cursor keys, you can move the cursor around, insert text or delete by use of backspace or delete keys.

Lesson 3: Going Further by Gotos

There is another type of statement that is important for any program, besides the simplest ones. Instead of executing each statement one after the other you can select any other statement to be executed by using a "goto"-statement. To make this clearer we give the following example:

```
 1           // Sum number from 1 to To
 2 Start     adr Run := con 1
 3           adr Sum := con 0
 4 Loop      if val Run > val To then goto con End
 5           adr Sum := val Sum + val Run
 6           adr Run := val Run + con 1
 7           goto con Loop
 8 End       stop
 9 To        con 5
10 Run       con 0
11 Sum       con 0
```

This program sums the numbers from one to To (which is here five, i.e. the variable To has the value five.) In the first statement at line 2 the variable Run gets the value 1, in the next line the variable Sum gets the value 0. We say, we **initialize** those variables by one and zero. Now comes a statement that evaluates a condition. It says, if something is true then go to somewhere else and execute the statements you find there. That is the concept of a goto: Change the regular execution sequence, if necessary. We use a **conditional goto** and an **unconditioned goto**.

Let's analyze our example. In line 4 you find a Label "Loop" and a conditional goto. First you have to evaluate the condition, which depends on the values of the variables of Run and To. If the first variable Run is larger than the second variable To, you end the program by going to the statement at line 8 (stop). Otherwise you perform the next statement at line 5, which adds to Sum the value of Run. This is done by getting the values of the variables Sum and Run, add them, and store the result to the variable Sum. It is important to see that the value of Sum is taken before it is changed. Anything else would be curious. In a similar way Run is incremented by 1 in the next statement at line 6. Now we come to the statement at line 7 which is an unconditional goto. It says to continue execution at line 4, where you find the statement labeled Loop. That is all you have to understand. Label a statement with a compiler constant like "Loop" and continue somewhere else execution of your program at that statement by use of the statement "goto con Loop".

Before you test this program, please save it. Do this always, since one of the worst problems is to end a program that executes infinitely, since there are loops that never ends. A loop consists of some statements that are repeated over and over again. And if you have forgotten to insert a conditional goto into that loop, it will never stop. It depends on your operating system how to stop this process, but in most cases almost everything will be lost, you have programmed – unless you have saved it before. Thus, use the save buttons as often as possible. It is so simple to do this, and it will help you to save time, money, and avoid frustration.

Now test the program. It should display the result

```
1+...+To=5 is Sum=15!
```

We told you already, that you can write anything into the Compiler Constant Memory Values area. So we wrote

```
1+...+To is Sum!
```

to get the result from above. Try it, and you will find it very helpful to document your results in the output area as easily as shown here.

Sometimes it is a good idea to follow program execution step by step. That helps to understand what your program is doing, so don't be shy to use the "Step" button. This will execute the first statement, refresh

the output and wait until you enter "Yes" for the next step, "No" to execute the program until its natural end, or "Cancel" to cancel program execution. Study the display. It tells you accurately what has happened in the last step. This is really a useful feature!

How can you run the program until some point and then start stepwise execution? Insert the statement "step" and the program will start Step-Modus whenever it executes this statement.

The statistics button opens a display that shows you the counts of the last execution, which are also cumulated. Use it if you need to know the number of statements your program has executed.

This ends lesson 3. You have learned how to use conditional and unconditional gotos and that the worst thing in computer programming is an infinite loops.

Lesson 4: Compute a quantity's place

You can do very much with the programming system you have learned so far. But there is some concept still missing that is important. This will be presented now.

We start by stating a problem. Instead of adding computed results, like those in the last lesson, we might be interested in adding a list of variables. How can we do that, if we know only the address of the first element in the list? Let us consider a program that solves this problem.

```
 1           // Sum numbers from First to Last
 2
 3           adr Run := con First
 4           adr Sum := con 0
 5 Loop      if val Run > con Last then goto con End
 6           adr Sum := val Sum + val val Run
 7           adr Run := val Run + con 1
 8           goto con Loop
 9 End       stop
10 Run       con 0
11 Sum       con 0
12 First     con 123
13           con 234
14           con 345
15 Last      con 456
```

It's almost the same program as the one before. The first two statements initialize Run and Sum. Run is now initialized by the address of the first value to be added, which stands at line 12. Thus the value of the variable Run is now an address! This shouldn't confuse you, since addresses are integer numbers, nothing else, and we can calculate with them as we can with any other integer. In line 5 the conditional statement tests, whether the address Run points to, is larger than the address of the last quantity to be summed, which stands at line 15. You should now understand while Run is prefixed by val, while Last is prefixed by con. Both values results to addresses (otherwise, this comparisons would be of no sense), but Run is an address variable, Last is an address constant. To get the value of a variable you need the prefix val, and to get the value of a constant, you need the prefix con.

The loop encloses the add-statement, which is now something quite new. The second operand of the add-expression at line 6 is prefixed by val twice. What's that? Well, Run holds a value that is an address. Writing val Run would result into the address, which however is not the value to be added, but the quantity found at this address is to be summed. Thus we have to inform the interpreter that we need the value found at an address, which is found at memory location Run. Run holds the address of a value, not the value itself. Thus, Run is to be "dereferenced" twice to get the value at the corresponding address location. It is complicated, I agree, but it is a fundamental concept you need in many places: Compute the address where you find an object. In most cases it is not so easy as we have assumed until now to find the location of a quantity. The concept of "**indirect addressing**", as this technique is called, is fundamental in programming.

In the next statement at line 7 we make some computation with addresses! We compute the next address by adding 1 to Run. That's okay. You can do so. Please, consider that you do not need to know the absolute value of the addresses. You only have to compute a relative value, namely the next, and it is not important to know whether your list of quantities starts at line 11, 100 or 333.

This method solves the problem. But it's only the first half of the story. Now comes the second one, which is not much more difficult (to be honest, it's as difficult as the first half). Sometimes we want to store a result at an address. Let us store some numbers (always the same) into the memory. To give a simple example, look at the following program:

```
 1           // Store squares from First to Last
 2
 3           adr Run := con First
 5 Loop      if val Run > con Last then goto con End
 6           adr adr Run := con 2
 7           adr Run := val Run + con 1
 8           goto con Loop
 9 End       stop
10 Run       con 0
12 First     con 123
13 Valu1     con 234
14 Valu2     con 345
15 Last      con 456
```

It is similar to the previous one, besides that now at line 6 the constant 2 is stored into a memory cell. The address of this memory cell is given by the Variable Run. And that is the whole story. To store something at an address that is given by the variable Run we have to write "`adr adr Run := …`". The meaning of this expression is: Write the value at the address found at Run. Again, this variable's value is an address. (By the way, you can also write "`adr val Run := …`", which is completely equivalent.)

Since this is not quite easy to understand, we give another example, although every thing has been told already. The next example copies all values from list `First1` to list `First2`. We need two memory variables, we call them `Run1` and `Run2`, the rest is obvious.

```
 1           // Copy from First to Last
 2
 3           adr Run1 := con First1
 4           adr Run2 := con First2
 5 Loop      if val Run1 > con Last1 then goto con End
 6           adr adr Run2 := val val Run1
 7           adr Run1 := val Run1 + con 1
 8           adr Run2 := val Run2 + con 1
 9           goto con Loop
10 End       stop
11 Run1
12 Run2
13 First1    con 123
14 Valu11    con 234
15 Valu12    con 345
16 Last1     con 456
17 First2    con 1
18 Valu21    con 2
19 Valu22    con 3
20 Last2     con 4
```

The result is

```
11 Run1=14
12 Run2=18
13 First1=123
14 Valu11=234
15 Valu12=345
16 Last1=456
```

```
17 First2=123
18 Valu21=234
19 Valu22=345
20 Last2=456
```

as you might have expected. It is important to understand this program, so study it carefully.

This ends lesson 4. You have learned the important concept of **indirect addressing**. Now you can compute the location of a quantity and access the value you find at the computed address.

Lesson 5: Plot a figure

This lesson is the most funniest one. It shows you how to plot a figure. Yes, you can draw a picture with 3AA. Did you expect this?

Well, the concept is simple, although cumbersome. At first, you have to define a plotter by defining a variable called PLOTTER. This is a **keyword**. Don't use it for anything else. Do it in the following way:

```
1           // Plotter example
2 PLOTTER   con 1
```

The next step is to assign something to this variable. Start simply by assigning two comma-expressions:

```
1           // Plotter example
2 PLOTTER   con 1
3           adr PLOTTER := con 100, con 100
4           adr PLOTTER := con 200, con 200
5           stop
```

The output will be a line, drawn on an extra panel that has popped up, and connecting the pixel points (100,100) with (200,200). A simple line, nothing else. In black, one point in size. Guess, you know how to plot further lines, e.g. to points (300,100)! Do it. Your program should now look like:

```
1           // Plotter example
2 PLOTTER   con 1
3           adr PLOTTER := con 100, con 100
4           adr PLOTTER := con 200, con 200
5           adr PLOTTER := con 300, con 100
6           stop
```

Now, assign a simple number to the PLOTTER. Use for example line 6 through 9 in the next example:

```
 1           // Plotter example
 2 PLOTTER   con 1
 3           adr PLOTTER := con 100, con 100
 4           adr PLOTTER := con 200, con 200
 5           adr PLOTTER := con 300, con 100
 6           adr PLOTTER := con 50
 7           adr PLOTTER := con 400, con 200
 8           adr PLOTTER := con 111
 9           adr PLOTTER := con 500, con 100
10           stop
```

Did you see, what happend? Well, the output has changed. Using a number between 1 and 100, you can select the thickness of your line. Its about one tenth of the number you assigned to PLOTTER. Of course, 0 means no output (i.e. the plotter's pencil is raised). But, if you have a number greater than 100, you select a color. The following list gives you the correspondence between code number and color:

Code	*Color*	*Code*	*Color*	*Code*	*Color*
101	black	106	green	111	red
102	blue	107	lightGray	112	yellow
103	cyan	108	magenta	113	white

Code	Color	Code	Color	Code	Color
104	darkGray	109	orange	>103	black
105	gray	110	pink		

Everything clear? It is up to you to plot any picture that you like. Give me a nice one!

However, one thing should be mentioned before. Before you rerun your program, you should clear the output. That is done by assigning the number -1 to the plotter. Also, if you rerun the program the plotter uses the last size and color values, which are now 50 and red. So, don't forget to reinit your PLOTTER. Use the following initial phase (line 3 to 5) for this!

```
 1           // Plotter example
 2 PLOTTER   con 1
 3           adr PLOTTER := con -1
 4           adr PLOTTER := con 1
 5           adr PLOTTER := con 101
 6           adr PLOTTER := con 100, con 100
 7           adr PLOTTER := con 200, con 200
 8           adr PLOTTER := con 300, con 100
 9           adr PLOTTER := con 50
10           adr PLOTTER := con 400, con 200
11           adr PLOTTER := con 111
12           adr PLOTTER := con 500, con 100
13           stop
```

Well, that's the first part of the plotter story. Now comes the second one. You can plot **text** as well, one symbol by the other. That's almost as simple as plotting a number. You need only to tell the PLOTTER which letter to write, and where. We introduce a new operator, the $, and append to the last program:

```
13           adr PLOTTER := con -1 $ con 'H'
14           adr PLOTTER := con 100$ con 300
15           stop
```

Do it. There will appear a letter 'H' at an appropriate place. Now, its up to you to write the “Hello World!” program! What's the color of this letter? It's red, isn't it. Well, you can plot a blue one by assigning the code number 102 to the PLOTTER. And what's about the size. Assign another one, for example 24, to get a smaller letter. Do it now!

```
14           ...
15           adr PLOTTER := con 24  // size is now 24 points
16           adr PLOTTER := con 102 // color is now blue
17           adr PLOTTER := con -1 $ con 'a'
18           adr PLOTTER := con 130$ con 300
19           stop
```

To send a character to the PLOTTER, you have to assign an dollar-expression, the first parameter of which is negative. That looks cumbersome, however, it seems to be a simple method to assign the parameters to the PLOTTER.

That' all about the plotter. You can test it, but you should always consider, that plotting is interesting particularly if you compute the coordinates of the points. And then you can produce very nice pictures, e.g.:

```
20           adr PLOTTER := con -1
21           adr PLOTTER := con 1
22           adr PLOTTER := con 101
23 Loop      if val Zahler > con 50 then goto con End
24           adr PLOTTER := con 0
25           adr PLOTTER := val x1, val y1
26           adr PLOTTER := con 1
27           adr PLOTTER := val x2, val y2
28           adr y1 := val y1 - con 1
29           adr y2 := val y2 + con 10
```

```
30            adr Zahler  := val Zahler + con 1
31            goto con Loop
32 End        stop
33 x1         con 100
34 y1         con 550
35 x2         con 700
36 y2         con 50
37 Zahler     con 0
```