Argante2
Assembly Tutorial

James Kehl
ecks@optusnet.com.au

May 26, 2002

Chapter 1

Assembly with NAGT

This is a little tutorial/HOWTO for writing Argante programs in NAGT assembly. It is assumed you are already familiar with general programming, though not completely familiar with Argante.
It's not completely essential to understand these low-level workings; there are higher level languages available which hide these details. But if you do put in the brain sweat, you will be able to break the limits of these languages, and even help develop them!

1.1 Assembler Basics

It's assumed you have your copy of Argante2 built and ready to run, other documents should be better able to explain how to do this - and we are assuming some basic skills here. We will also work out of the Argante root directory.

1.1.1 Registers and Arithmetics

Our first program will add the numbers from 1 to 5.

begin ex_reg1.agt

```
# This program adds the numbers from 1 to 5.
# This is a comment, by the way.
.code
        mov u:r0, 0
        add u:r0, 1
        add u:r0, 2
        add u:r0, 3
        add u:r0, 4
        add u:r0, 5
        syscall2 $IO_PUTINT, u:r0
        syscall2 $IO_PUTCHAR, 10
        halt
```

end

Save this code into a file called ex_reg1.agt, using a text-only editor (very important!) and *assemble* it with "compiler/nagt ex_reg1.agt ex_reg1.img". This tells NAGT to produce ex_reg1.img from ex_reg1.agt, and in future this will be what we mean when we say 'assemble x.agt'.
Now run it - it should say 15.
So, what does each line do in the above code?

`.code`	tells the assembler to expect code instructions.
`mov u:r0, 0`	sets the *unsigned* value of register 0 to 0.
`add u:r0,` *x*	adds x to the unsigned value of register 0.
`syscall2 $IO_PUTINT, r0`	prints the contents of register 0, assuming it to be an integer.
`syscall2 $IO_PUTCHAR, 10`	prints a newline (ASCII 10) character.
`halt`	terminates the program.

In an ordinary programming language, you construct variables, which have specific types.
In Argante2, there are 31 registers (32, but r31 is special) - basically variables - which can be accessed as any type. 31 might not seem all that many, but you will never end up using more than about 10 in even the most complicated programs, and there are ways to create more variables, which we will go into later.
So why aren't there specific types to registers? As they are shared between all functions in your program, it would otherwise be impossible to use 32 ints in one place and 32 floats in another. Another good reason is that it dramatically simplifies the syscall calling convention - more on that later.
As an aside, an *unsigned* value is one which can take integral (whole number) values from 0 to 4294967295 (no negatives!), a *signed* value can range from -2147483647 to $+2147483647$ (integral), and a floating point value can take positive or negative, fractional values from, at worst, $1 * 10^{-37}$ to $1 * 10^{37}$. A float is not as fast as a integer type, and may only provide 6 digits of accuracy.

Exercise. Write a program to find 3 factorial and 5 factorial.

Figure 1.1: Summary of arithmetic commands.

The following operations work with all register types: u: (unsigned), s: (signed) and f: (floating point), and any mixture.

`mov` x, y	Sets x to y
`add` x, y	Adds y to x
`sub` x, y	Subtracts y from x
`mul` x, y	Multiplies x by y
`div` x, y	Divides x by y
`mod` x, y	Sets x to the remainder of x/y

The following binary operations require two unsigned arguments

`and` x, y	Sets x to the binary intersect of x and y
`or` x, y	Sets x to the binary union of x and y
`xor` x, y	Sets x to the exclusive-or of x and y
`not` x	Sets x to its binary inverse.
`shl` x, y	Multiplies x by 2^y, discarding bits above 2^{32}.
`shr` x, y	Divides x by 2^y, discarding bits below 1.
`rol` x, y	Multiplies x by 2^y, moving bits above 2^{32} to the lower end.
`ror` x, y	Divides x by 2^y, moving bits below 1 to the upper end.

The following commands are useful for debugging.

`syscall2 $IO_PUTINT,` x	prints x as a signed int.
`syscall2 $IO_PUTCHAR, 10`	prints a newline character.

1.1.2 Labels and Control Flow

In the previous section, you can see we used the same code, repeated five times, with different arguments. When you are dealing with complicated operations (generally anything above five instructions) you don't want to do this: it's too much work, it's easy to make mistakes, and one change has to be repeated throughout the code.

Hence, we use subroutines.

The following program uses 'call' and 'ret' to print the first ten terms of the Fibonacci Sequence:

`begin ex_lab1.agt`

```
# This program prints ten terms of the Fibonacci Sequence.
.code
        mov u:r0, 0
        mov u:r1, 1
        syscall2 $IO_PUTINT, r1
        call :fibo_func
        call :fibo_func
        call :fibo_func
        call :fibo_func
        call :fibo_func
```

```
        call :fibo_func
        call :fibo_func
        call :fibo_func
        call :fibo_func
        syscall2 $IO_PUTCHAR, 10
        halt
        fibo_func:
        mov u:r2, u:r0
        add u:r2, u:r1
        mov u:r0, u:r1
        mov u:r1, u:r2
        syscall2 $IO_PUTCHAR, 20 # ASCII 20 - a space character.
        syscall2 $IO_PUTINT, u:r1
        ret 1
```

end

Type in this program, assemble and run it. What order are the instructions executed in?

Exercise. Write a program to find and print the second, fourth, eighth and sixteenth powers of 1, 2, 3 and 4.

At this point we will mention the 'jmp' instruction. It resembles the 'call' instruction in syntax and function, but there is no 'ret' equivalent. The next 'ret' instruction will still return to the last unreturned 'call', which can be useful if the last instruction in a function would have been a call to another function:

begin ex_lab2.agt

```
# This program demonstrates tricky (ab)use of the jmp instruction.
.code
        mov u:r0, 0
        mov u:r1, 1
        mov u:r2, 1
        call :func
        halt
:func
        call :subfunc
        call :subfunc
        call :subfunc
        call :subfunc
        jmp :subfunc
:subfunc
        mov u:r3, u:r1
        mul u:r3, u:r3
        add u:r0, u:r3
        add u:r1, u:r2
        syscall2 $IO_PUTINT, u:r0
        syscall2 $IO_PUTCHAR, 10
        ret 1
```

```
end
```

This example is a little contrived. What do you think would happen without the 'jmp' - try it!

Exercise. Modify the program above to add the squares of the first five even numbers.

Figure 1.2: Summary of flow control commands.

`:label`	Defines a label
`jmp :label`	Makes :label the next instruction to execute.
`call :label`	Executes the instructions following :label, but resume here on a 'ret'.
`ret 1`	Resumes execution following the last active 'call'
`ret` x	Resumes execution following the xth last active 'call'. 'calls' which occurred after that will be forgotten.

1.1.3 Conditionals and Loops

So far you haven't done anything a pocket calculator wouldn't be able to do faster. We'll make up for that by introducing the conditional commands.

begin ex_cond1.agt

```
# This program prints the sum of the numbers between 1 and 100.
.code
        mov u:r0, 0
        mov u:r1, 1
:loop_top
        add u:r0, u:r1
        add u:r1, 1
        ifbel u:r1, 100
        jmp :loop_top
        syscall2 $IO_PUTINT, u:r0
        syscall2 $IO_PUTCHAR, 10
        halt
```

end

That made life easy! Only when a conditional comparison is TRUE does the next statement get executed. Note that execution normally continues on to the statement after that, so unless the first statement is JMP, the second statement is always executed.

Exercise. What happens when you put one conditional right after another? Can you see a use for this?

As loops like this are fairly common, there is a special 'loop' instruction, which decrements a counter and jumps to an address if the counter is above zero.

begin ex_cond2.agt

```
# This program prints the numbers between 1 and 100 ending in 4 or 6.
.code
        mov u:r0, 1
        mov s:r1, 100 # Signed values for decreasing loops are preferable.
:loop_top
        add u:r0, 1
        mov u:r2, u:r0
        mod u:r2, 10 # What's the last digit?
        ifneq u:r2, 4
        ifeq u:r2, 6
        call :print_num
        loop s:r1, :loop_top
        halt
:print_num
        syscall2 $IO_PUTINT, u:r0
        syscall2 $IO_PUTCHAR, 10
        ret 1
```

```
end
```

Exercise. Modify the above program to print an exclamation mark (ASCII 33) if the number is also divisible by 7.

Figure 1.3: Summary of conditionals.

`ifabo` x, y	If $x <= y$, skip next command.
`ifbel` x, y	If $x >= y$, skip next command.
`ifneq` x, y	If $x = y$, skip next command.
`ifeq` x, y	MIf $x \neq y$, skip next command.
`loop` x, `:addr`	Decrease x, and goto :label if above zero.

1.2 Advanced Assembler

Hey, well done! Now you can do all the maths as you could ever want. Ok, maybe not, but somebody is probably writing a complex/trig/log/linalg module somewhere. Give us enough time and someone will write a Differential Equation engine...

Not that you wanted to do maths anyway, right?

1.2.1 Data and References

In the previous section, we mentioned that there were other ways of storing data other than registers. All of them revolve around memory, so it's time to learn to use it.
Here's one of our old programs - recognize it?

begin ex_dat1.agt

```
# This program prints ten terms of the Fibonacci Sequence.
.data
:terma
        0
:termb
        1
:tempterm
        0
:count
        10
.code
:loop_top
        syscall2 $IO_PUTINT, *s::termb
        call :fibo_func
        loop *s::count, :loop_top
        syscall2 $IO_PUTCHAR, 10
        halt
        fibo_func:
        mov *u::tempterm, *u::terma
        add *u::tempterm, *u::termb
        mov *u::terma, *u::termb
        mov *u::termb, *u::tempterm
        syscall2 $IO_PUTCHAR, 20
        ret 1
```

end

The first thing you should notice is that the first thing in the file is not '.code' - it's '.data'. This must mean that all those labels are *data* labels.
And, scattered through the code where the registers used to be, are their references - looking sort of like code labels, only with a '*' in front of them, and a 'u:' part we recognize as specifying the data type.
When we used the ':label' syntax for code, the assembler replaced the reference with the address of the code. Now for data, we don't really want to know the

address of the variable - we want to change the contents. So, we must indicate that we want the contents of the address. That's the '*'.
You might think that using data labels instead of registers makes code a lot more readable, and you'd be right. But registers are accessed with less complexity than other memory, which makes them a lot faster.
You can also apply the dereference to registers:

begin ex_dat2.agt

```
# This program prints an array.
.data
:numbers
        -1000
        2000
        -4000
        8000
        -16000
.code
        mov u:r0, :numbers
        # % means "size (in integers) of this label's content"
        mov u:r1, %numbers :loop_top
        syscall2 $IO_PUTINT, *s:r0
        syscall2 $IO_PUTCHAR, 10
        add u:r0, 1
        loop u:r1, :loop_top
        halt
```

end

Note r0 is *not* used as more than one type. The dereference line means "get the signed contents of the memory at address r0", not "get the contents of the memory at signed address r0". There's no such thing as a signed address.

Exercise. What happens if you change %numbers in the above code to, say, 400? Why?

1.2.2 Strings and Buffers

So far all we've done is print numbers, numbers, numbers. Maybe last section we used an array - very important stuff, but still *dull*. Nobody wants code that runs like a collection of Windows error messages ("MMSYSTEM 451: Contact vendor for details").
So, onward! Text!

begin ex_str1.agt

```
# Ultracool stuff.
.rodata
:message_to_world
        "You smell DISGUSTING!!\n"
.code
        mov u:r0, :message_to_world
```

```
        # ^ means "size (in bytes) of this label's content"
        mov u:r1, ^message_to_world
        syscall $IO_PUTSTRING
        halt
```

end

Well, it's original.
Note we have to feed IO_PUTSTRING two arguments via registers - the address and the size (in bytes), and that it isn't called by 'syscall2' - it's called by 'syscall'. ('syscall2' is called what it is because it has two arguments.)
The second thing to notice is that we didn't use '.data' - we used '.rodata'. Because this string is not supposed to be modified, we can tell Argante so, and impede any evil hackers changing our manifesto.

Exercise. Write a program that prints an array of strings.

begin ex_str2.agt

```
 # Ultracooler stuff.
 .ropack
 :question_of_user
        " What's your name?\n"
 :message_to_user
        " You smell disgusting, "
 .data
 :buffer
        0x0 repeat 8
 .code
        mov u:r1, :question_of_user
        mov u:r2, ^question_of_user
        syscall $CFD_WRITE
        mov u:r1, :buffer
        mov u:r2, ^buffer
        syscall $CFD_READ
        mov u:r3, ^buffer
        sub u:r3, u:r2
        mov u:r1, :message_to_user
        mov u:r2, ^message_to_user
        syscall $CFD_WRITE
        mov u:r1, :buffer
        mov u:r2, u:r3
        syscall $CFD_WRITE
        mov u:r1, 10
        syscall $CFD_WRITE_CHAR
        halt
```

end

This program makes use of input, which is a relatively unpolished feature as of the time of writing. (Windows is currently OK, unices depend on the configure options. No Multithreading = Console Input works.) So it might fail with a ERR_BAD_FD code.

Exercise. Get out your Syscall Reference and figure out how this thing works, when it works.

As we have more than one string, and they are readonly, it's foolish to keep them in separate sections in the file. So we specify '.ropack', which makes the assembler pack them into the same section.

Figure 1.4: Summary of data manipulators.

`*`	References the contents of an address.
`:`	The address of a label
`%`	The size of a label's content in dwords (four-byte blocks)
`^`	The size of a label's content in bytes.
`.data`	Creates data segments.
`.rodata`	Creates readonly data segments.
`.ropack`	Place multiple items in one readonly data segment.
`.packed`	If you use this without knowing what you are doing, a black hole will swallow the earth.
`repeat` *n*	Repeat all data on this line *n* times.

1.2.3 Dynamic Allocation

Now you are able to read input from a console, or, with a bit of side reading, from a file, you're going to be able to make things happen. But what if you don't know much memory your buffer is going to need? You could just put a really big number after repeat, but this will bloat your .img file. What you need is to be able to create data sections at runtime!

The trouble is that the assembler has no way of knowing where this segment is going to end up. So you have to keep this address in a register, or, if you store it in a data segment, you have to do two separate dereferences to get at the contents.

begin ex_alc1.agt

```
# What DOES this program DO?
.data
:numbers
        -1000
        2000
        -4000
        8000
        -16000
.code
        # We'll store the address in r16.
        mov u:r16, %numbers
        alloc u:r16, 3 # 3 = 1 + 2 = READABLE + WRITABLE
        mov u:r1, u:r16
        mov u:r0, :numbers
        mov u:r2, %numbers :loop_top
        mov *u:r1, *u:r0
        add u:r0, 1
        add u:r1, 1
        loop u:r2, :loop_top
        free u:r16
        halt
```

end

Exercise. Create a small, useful example of alloc, and send it to the author.

Figure 1.5: Summary of data allocation commands.

`alloc` x, y	Allocates a block of size x and permissions y, and stores its address in x.
`realloc` x, `u:`y	Resizes block x to size y and stores its new address in x.
`realloc` x, `s:`y	Changes the permissions of block x to y.
`free` x	Deallocates a block.

1.2.4 The Metastack

Let's say you have a subroutine which uses some temporary variables - in registers, or data areas, it doesn't matter. And this subroutine has to call itself, multiple times, without messing up its temporaries. Tricky? You should already know one solution using an array...

Save the temporaries in the array before you call, and increment the address into it. Then the next call will save its own temporaries in a different part of the array. And when it returns, you decrement the address and restore your temporaries.

It's a lot of work.

Hence Argante provides built-in support for this with the metastack functions. Once you have set up the metastack, you just 'push' (add) and 'pop' (retrieve and remove from stack) things you want to save or retrieve,

begin ex_stk1.agt

```
# Yes, they are getting boring for me, too.
.data
:stack
        0x0 repeat 40
.code
        stack :stack, %stack
        mov u:r0, 1
        call :funny_func
        syscall $IO_PUTINT
        halt
:funny_func
        add u:r0, 1
        ifabo u:r0, 5
        ret 1
        push u:r0
        call :funny_func
        pop u:r0
        ret 1
```

end

If everything works, the program will print 2.

Figure 1.6: Summary of metastack.

`stack` x, `u:`y	Set the metastack address to x and size to y. Does not zero the metastack pointer - but this may change.
`push` x	Add x on the stack and increment the metastack pointer.
`pop` x	Decrement the metastack pointer, and retrieve a value into x.

1.2.5 Some Things We Skipped

There is another sort of assembler directive we haven't mentioned. These are placed before any '.data' or '.code' directives and set various elements of the image header. And most of these elements are obsolete.
There is one command which still may have a future: '!signature'. Currently it allows you to embed a short (32char) description of your program, which should probably include its author.
If you have a register dedicated to a specific function all the way through your code, you don't actually have to write the number all the way through your code. Use '.define' macros, and your code will be easier to read, and you will make less mistakes.
It's also a good idea to use '.define' macros for numeric options and nearly anything that's shared between different bits of your program.

begin ex_msc1.agt

```
# UselessProgram version 2.
!signature "UselessProgram <noflames@please>"
# We'll still store the address in r16, but it's easy to change.
.define addy% u:r16
.data
:numbers
        -1000
        2000
        -4000
        8000
        -16000
.code
        mov addy%, %numbers
        alloc addy%, 3 # 3 = 1 + 2 = READABLE + WRITABLE
        mov u:r1, addy%
        mov u:r0, :numbers
        mov u:r2, %numbers :loop_top
        mov *u:r1, *u:r0
        add u:r0, 1
        add u:r1, 1
        loop u:r2, :loop_top
        free addy%
```

```
        halt
```
end

Exercise. Why use a funny symbol on the end of addy? What happens if you change 'addy%' to 'rs'? Why?

1.3 Argante Concepts

This ends the NAGT-specific part of this guide. From here on, this applies to everything in Argante, on every level - even HLLs.

1.3.1 Exceptions

When a 'fatal' error happens in Argante, we use a concept usually found in object oriented languages: that of *raising* (or throwing) an exception. If there is no active *exception handler* in a subroutine in which an exception is raised, the subroutine is aborted and the exception propagates up to the parent routine. (If there is no parent routine, the program dies.)
Examples speak louder than words, anyway:

begin ex_exc1.agt

```
# Exception demonstrator.
.ropack
:ex1
        "Exception handler 1 activated.\n"
:ex2
        "Exception handler 2 activated.\n"
.code
        handler :hand2
        call :subrtn1
# The code should never get here, so let's crash if it does.
# Handler 0 deactivates the exception handler.
        handler 0
        mov *u::ex1, 0 # write to readonly data
:subrtn1
# This handler is for the current function only.
        handler :hand1
# To throw an exception yourself, use 'raise'.
        raise 0xfeedface
        ret 400 # This should die, too, despite a handler...
:hand1
        mov u:r0, :ex1
        mov u:r1, ^ex1
        syscall $IO_PUTSTRING
# The exception handler has been activated,
# and so won't trigger again (that would loop forever!)
        raise 0xf007f00d
```

```
        ret 400
:hand2
# The exception number (0xf007f00d) will be placed in r31.
        syscall2 $IO_PUTHEX, u:r31
        syscall2 $IO_PUTCHAR, 10
        mov u:r0, :ex2
        mov u:r1, ^ex2
        syscall $IO_PUTSTRING
        halt
```

end

Exception numbers for system-defined errors are listed in include/exception.h, and their symbolic names ($ERR_xxxx) can be used in assembler code.

Figure 1.7: Summary of exception commands.

`handler 0`	Any exceptions in this routine will be passed upward. (No handler.)
`handler :label`	Any exceptions in this routine will set r31 to the exception number, and jump to :label.
`raise` x	Raises exception x.

1.3.2 Syscalls

Each syscall is part of an Argante kernel module (i.e. privileged code) which are written to allow interactions with the real system or to do things difficult to achieve from Argante code. (For example, the StrFD module could be written in Argante, but it would be slower and would require an OO language to be as useful.)
In general, all syscalls (used with the 'syscall' op) will take their first argument in r0, their second argument in r1, their third argument in r2, and so on.
For a call marked by "SYS2", you can use the second argument of syscall2 instead of r0.
Return values are not so consistent. When you see 'ignore' in a list of outputs from a syscall, it means that register is returned unchanged.
Take CFD_READ for example:
it returns
[ignore, unsigned @null_after_data, unsigned space_left_in_buffer]
which means r0 is unchanged.

All the syscalls can be found in the Argante2 Syscall Reference - likely to be found wherever you found this document.

1.4 Conclusion

Hey, you now know every opcode in Argante2 assembler. Go forth, and code!