Amoral Programming Language

Written by Paul Robson November 2020

Introduction

Amoral is a 16-bit programming language designed for the 6502, though it is not limited to it.

It is based around an extremely simple conceptual virtual machine that has one 16-bit register; operations are all on this 16-bit register. So, one may add to it, save or load it, and so on. This register is commonly referred to as R.

It is rather like using a programmable calculator, without the internal stack logic that makes 2+3*4 work, or as if you pressed equals after every operation: 2+3=5 *4 = 20.

You have “STO” and “RCL” so to speak ; a named register is “RCL” and “STO” is the pling (e.g. !count) [guess who likes FORTH], procedures/functions with parameters, loops, tests and so on.

There is the same basic idea of “the number in the display”.

This has two direct consequences. It maps very easily onto the 6502, which has very few registers, and it is reasonably straightforward to compile into machine code if required. In the 6502 implementation the 16-bit register is XA.

Amoral can generate either a p-code or 6502 code and switch between the two as you like. It should work the same; just one is much faster and takes up about 3-4 times the space.

This gets round the difficulties of the processor; p-code is compact but too slow; 16-bit 6502 code is fast but space inefficient.

Language

Syntactically the language is quite simple. A statement is either an operation on the accumulator, or ‘something else’ – a procedure call, a structure, whatever. Multiple statements are grouped in curly brackets.

Outside quoted strings and characters, case is ignored. Commenting is like Python, anything after a ‘#’ character is ignored in a line.

Terms

The following terms are supported.

Type	Example	Description
Constant	42	Integer constant
Hexadecimal	$F7	Hexadecimal constant
Character	'C'	ASCII value (32-127 only, not ')
String	"hello world"	ASCIIZ string (value is the address)
Variable	Count hello_world.42	A 16-bit static variable.
Reference	@count	Address of variable/procedure count

Binary Accumulator operations

Accumulator operations all involve a term, which is a variable or a constant, all 16 bits. It is not possible to work with 8-bit data directly. The data is unsigned.

The operators are as follows:

Operator	Examples	Function
None	42 count	Loads value into accumulator
!	!count	Stores value into variable. (not constant)
+	+5	Add
-	-count	Subtract
*	*$2A	Multiply
/	/4	Divide (zero is undefined)
%	%9	Modulus (zero is undefined)
&	& count	Binary and
|	| 'X'	Binary or
^	^ 72	Binary exclusive or.

There is no concept of precedence in the language. If you write a+2*3 it is not a+6 it is (a+2)*3, because there is no concept of an expression, merely terms and operators.

Operators cannot be used on strings, so you cannot write +"hello" which makes no sense anyway :) If you absolutely must you could write var temp;"hello" !temp …. then +temp

Unary Accumulator operations

Unary operations have no term, and simply operate on the 16-bit register

Operator	Description
>>	Logical Shift Right (bit 15 is cleared)
<<	Logical Shift Left
++	Increment
--	Decrement
true	Sets to $FFFF (e.g. -1)
false	Sets to $0000 (e.g. 0)

Identifiers

Identifiers must begin with a letter and after that can be alphanumeric, underscores or a full stop. It is advised, but not enforced, that the full stop be used for modular code.

Variables

Variables, which are all 16-bit single items, can be declared at any point. All variables are static, so if you want to do recursion that you will have to handle yourself (the procedure call will be fine).

Variables are either global, or local when defined inside a procedure. Local variables cannot have the same identifier as globals (because I say so!) or procedures.

Variable are initialised to zero at the start of the program.

A variable declaration looks like this.

```
var count,name,hello_world_42
```

Procedures and Functions

Procedures and functions are the same thing. This is achieved using the 16-bit register. When a procedure exits, the value currently in the 16-bit register stays there, so a function can be created by simply putting the return value in that register.

Entry is slightly different. If there are no parameters, then the procedure is called as a straight subroutine, but R is unchanged. So, you could write a unary function which doubles R as

```
proc double() { *2 }
```

When there are parameters, the current value of R is lost, as it is used to copy parameters into their respective arguments (which, again, are static and unique like variables). The last parameter is passed in R, however, and the first thing the implementation of the procedure does is to store this last parameter.

Note you can use ‘proc’ or ‘func’ in the definition interchangeably, purely for readability purposes.

So, a simple procedure that adds three numbers.

```
Proc addup(a,b,c) { a+b+c }
```

called with

```
addup(42,count,'A')
```

1. proceeds as follows (green code is the procedure addup being executed)
2. copy the constant 42 to the local parameter a
3. copy the variable count to the local parameter b
4. load the constant 'a' (97) into R (as it is the last parameter)
5. call the routine. 'addup'
6. the first thing addup does is to store R in c, completing the parameter load
7. load a into R
8. add b to it
9. add c to it
10. return with the total of a,b and c in R.

Procedure parameters can be sequences of operators, but not function calls. So, you can write :

```
addup(42,count*2,"Hello world")
```

though it would be more sensible to use count<< because a left shift is far quicker than a 16 bit multiply. This may be a later optimisation (for +1,-1,*2,/2) but do not rely on it.

Procedures parameters can be ~ which means 'use the current value of R'.

Code Structures

Structures again resemble C.

However, the use of { } is mandatory as the semicolon is merely syntactic sugar to delineate sequences which perform specific actions (e.g. the while loop below)

Conditional structures have a sequence which is surrounded in brackets, which can be empty, ended by a conditional comparison to a constant. e.g.

```
while (>=0)          if (a==5)       while (a+2<0)     if
                        (count<>9)
```

these can contain function calls unlike parameters. Signed comparisons are done by subtracting and looking at the sign of the result.

So, your loop may resemble

```
while (a >= 0) { print.constant(a) ; a--!a }
```

There is no direct equivalent of the for loop. Amoral has a structure 'times' which executes a code a given number of times, counting down until zero. A variable can be used as the index, or it can be omitted, in which case the compiler will handle it.

So, the following

```
times(6,x) { print.constant(x); }
```

would print 5,4,3,2,1,0. If you wanted to print 6 stars you could just write

```
times(6) { print.char('*'); }
```

The count parameter can be any sequence including ~ (the current state).

Note : all conditionals are destructive, so if you do if (a == 0) { <do something> } you cannot guarantee the value of R after the test has been evaluated.

Inline Code

Besides programming code, code can also be inlined using [] in quotes e.g. ["20F203"] inlines JSR $03F2.

This will obviously *not* behave the same way when used in p-code or 6502 code, as the two are numerically vastly different.

It is really designed to allow automatically generated code, not for the developer to use.

It can also be used for data, for example if you have code such as

```
code proc my.data() { ["02040608"] }
```

then @my.data will get the address of the procedure ; because it is a code procedure there is nothing in it other than the data, so it is the address of the data. Of course, if you try to call this procedure it will probably crash !

Functions

The following functions are in the run time.

Function	Platforms	Purpose
abs(n)	All	Returns the absolute value as if it were signed
alloc(n)	All	Allocate n bytes of memory, return base address
halt.program()	All	Stop program
len(str)	All	Return length of ASCIIZ string
max(n1,n2)	All	Return larger of n1,n2 (unsigned)
min(n1,n2)	All	Return smaller of n1,n2 (unsigned)
neg(n)	All	2's complement the register
peek.b(a)	All	Read byte
peek.v(a1,a2)	X16	Read VRAM (17 bit address, a1 is bank)
peek.w(a)	All	Read word
poke.b(a,d)	All	Write byte
poke.v(a1,a2,d)	X16	Write VRAM (17 bit address, a1 is bank)
poke.w(a,d)	All	Write word
print.character(c)	All	Print character code(c) (32...127)
print.crlf()	All	Move to start of next line
print.hex(n)	All	Print unsigned hex with leading space
print.int(n)	All	Print unsigned decimal with leading space
print.sint(n)	All	Print signed decimal with leading space
print.string(str)	All	Print ASCIIZ string
random()	All	Return random 16 bit integer
read.timer()	X16	Return number of 60Hz ticks since the start.
sgn(n)	All	Return -1,0,1 dependent on sign of n

Data Structures

Structures are defined as follows:

```
struct sprite { x,y,w,h, angle }
```

this generates a set of functions automatically to manage such a structure. Structures must be defined outside of procedure definitions, and they are all global.

Each structure type has an allocated address in zero page ; these are allocated sequentially after the zero page area used by the runtime. This address is used to access an object, it is the rough equivalent of *this* in javascript.

This is done by automatically generated functions which provide the functionality, which is shown below.

Using the above as an example, and the member angle. Getters and Setters are defined for every field in the structure.

Function	Purpose
sprite.new()	Allocates a chunk of memory big enough to hold one sprite (10 bytes in the example) using alloc() and sets it as current in the zero page.
sprite.use(a)	Work on sprite at a ; this sets the value in the zero page.
sprite.swap()	To ease comparison between two sprites, this swaps the zero page value with a variable in memory.
sprite.r_angle()	Returns the angle value in sprite (e.g. offset 8,9)
sprite.w_angle(n)	Writes the angle value in sprite (e.g. offset 8,9)

System structure

Memory is used as follows by default. The locations can vary of course.

The Boot Section The boot section is 32 bytes long, and normally starts with a jmp <last defined routine>, it can also contain parameters passed in and anything else the developer wants. It is predefined as boot.address
The Amoral runtime. This is a linked list of routines which include the P-Code interpreter and helper functions such as multiply and divide which the 6502 cannot do. Routines are added to the definitions.
Additional user runtimes. These use the same format as the Amoral runtime.
Amoral generated code. A mixture of 6502 routines and pseudo code (and data, perhaps). These also are linked together to make a dictionary of all the routines
End of Dictionary marker ($0000, 2 bytes)
Available memory for user allocation, this is stored at boot.address+30

Variable memory is stored separately. These are for statics; arrays should be allocated from the user allocation memory at the end. This block is by default 1024 bytes in length, allowing for 512 global and local variables. In pseudo code, the "address" is an index into this table as a word, so the 4th word if the table is at $800 would be at $808 ($800+4*2)

The structure of the dictionary is as follows; it is like FORTH.

Offset	Description
+0,+1	Offset to next entry, or $0000 if this is the end of the dictionary.
+2	Name of routine as an ASCIIZ string in lower case. Can be $00 (private)
(+n)	Possible padding, as the code must be on an odd address, should be $00
+n	6502 code, possibly with Pseudocode intermixed.

A directly runnable amoral binary would be prefixed by the execution address, which would be the start of the boot area.

The following are reserved in the 32 byte boot area.

Offset from Boot	Contents
22	First zero page used
23	Byte after last zero page address used
24,25	Address of Boot
26,27	Address of the Variables
28,29	Address of the Amoral code, e.g. the end of the run time.
30,31	Address of Memory available to user.

Code Generation Control

Code generation control is controlled by the three keywords *fast slow* and *code.* The first two specify 6502 of p-code generation.

The third is used for the [] format, its means there are no store or header or exit or anything similar. No code is generated at all. If you include amoral code it is compiled in fast mode, as 6502 code.

The default is ‘slow’

Procedure structure

Every procedure is a 6502 machine code one, and they all pass their last parameter / current state in the 16 bit register XA.

Procedures that are written in pseudocode start with code to write the last parameter out, and a 6502 subroutine call to execute the pseudo code that immediately follows. Code that generates pure 6502 code just saves the last parameter.

So, suppose you have a subroutine defined as do.something(a,b,c) the code would look like the following depending on how it is compiled. (the last provided parameter, which should go in c, will be in R as described earlier). Only the green bits change depending on fast/slow compilation.

Phase	6502 code	P-Code
Start Point	do.something:	do.something:
Write out c parameter	sta c stx c+1	sta c stx c+1
Start P-Code		jsr interpretpseudocode
Main body	adc #<stuff> inx (etc.)	pseudo code instructions (see later) ret (exits p-code)
Exit	rts (6502 opcode)	rts (6502 opcode)

If there is no parameter, the 'c write out' does not happen. This does not affect the passed in value in the 16 bit register XA. The InterpretPseudoCode command takes the responsibility of saving and loading XA.

A corollary of this is that an Amoral program can be used as a library for any other language (it will use some zero page locations so may need rebuilding for that) PseudoCode Definition

When running pseudo code, the 16 bit register will be kept in page zero, not in XA. However, it will be in XA on entry, on exit, and when calling a subroutine.

Pseudo Code

6502 Subroutine Call.

00-7F are subroutine calls. The high byte is in the following byte, and the address is built from that byte, and the lower 7 bits of the command shifted left once.

This physically calls a 6502 routine at an *even* address. On entry XA are loaded with the current value of R, and on exit it should have the (possibly) new value of R.

So, 35 8E executes code at 8E6A ($8E00+2*$35) for example.

The code is 6502 code not pseudo code. Remember all procedures are 6502 callable whether they are in pseudocode or not.

Branch/Memory Access Instructions

80-EF are branches and memory access instructions (branches are to pseudocode not 6502 code). These have the form.

7	6	5	4	3	2	1	0
1	Addressing Mode			Command			

Address modes are as follows

Binary	Example	Description
000	ldr #4	Short constant 0-255 in following byte
001	ldr #4137	Long constant in following byte, low high word order.
010	ldr [20]	The following byte is an index into the variable ram area
011	ldr [2091]	The following word (low/high) is an index into the variables
100	ldr $4173	The following word (low/high) is an absolute physical address

As stated early the [x] value is an index into a word array in the variable RAM area, not a physical address in its own right. This means for the first 256 declared local/global statics, access and operations only require 2 bytes of memory in p-code generation mode.

Commands are as follows.

Binary	Mnemonic	Action
0000	ldr	R := value
0001	and	R := R & value
0010	orr	R := R | value
0011	xor	R := R ^ value
0100	add	R := R + value
0101	sub	R := R - value
0110	mlt	R := R * value (upper 16 bits lost)
0111	div	R := R / value (division by zero unknown result)
1000	mod	R := R mod value (zero unknown result)
1001	dcv	[addr]-- (not immediate)
1010	str	[addr] := R (not immediate)
1011	bra	PC := addr (absolute only)
1100	beq	If (R == 0) PC := addr (absolute only)
1101	bne	If (R <> 0) PC := addr (absolute only)
1110	bmi	If (R < 0) PC := addr (absolute only)
1111	bpl	If (R >= 0) PC := addr (absolute only)

Note the tests are *not* like the 6502 ; they are on the current value of R not the result of the last operation.

Unary Instructions

Unary instructions take the format below, and do not have any parameters.

7	6	5	4	3	2	1	0
1	1	1	1	Command			

The following instructions are supported.

Binary	Mnemonic	Action
0000	inc	R++
0001	dec	R--
0010	shl	R shift left once.
0011	shr	R shift right once (bit 15 is zero)
0100	clr	R := 0 (also used to pad procedures to even addresses)
0101	ret	Return to caller – loads R into XA and executes 6502 rts

Why is it called AMORAL ?

Well the claim is that it stands for Accumulator based Mostek Register Associated Language.

The truth is I put Accumulator Mostek Language into a webpage that generates acronyms and "amoral" came out, and I thought, well I must call it that.

Revisions

Date	Notes
13 Nov 2020	First completed version of AMORAL specification/design
14 Nov 2020	Fixed SHL name. Changed times. Procedures always exit with RTS.
15 Nov 2020	Removed @ operator as it does not work.
16 Nov 2020	Added @ back in, conditions allow arbitrary constants. Added true and false, fast, slow and code. Added inbuilt functions table.
17 Nov 2020	Added DCV. Tweaks for 'Code'. @ works on procedures. Added structure design.