Assembly Asylum

ASM 1.0.0 Release Build 3

System Manual

Introduction.

Assembly Asylum's goal is to teach some of the lost art of assembly language programming using our own custom compiler, linker and runtime.

The system, simply named 'ASM' is a sandboxed environment written specifically so that it can't harm your computer in any fashion. It has no access to write files or modify anything within your kit. Apart keyboard polling and reading from plain text files it is completely locked for down learning purposes.

Version 1.0.0 of ASM is entirely terminal based. Should their be the interest then a more advance GUI version with optimised GPU call may be added.

ASM relies on the use of VT100 escape codes to communicate with the terminal. A VT100 compatible terminal and 64Bit OS are the only system requirements. On Linux and Mac no additional terminal software is requires and both systems are fully compatible.

ASM is a standalone compiler and runtime. It does not have an internal editor so you will require the preferred text editor of your choice. A BBEdit code-less language file is available to help with syntax and code highlighting if you use BBEdit.

Instructions / Reference.

Binary.

Instruction	Usage
NOT	Performs a NOT operation on the bits of the value in register A, bits are flipped between 0 and 1. Default is an 8 bit operation, though the number of bits can be specified as a parameter.
OR	Performs an OR operation on the value in Register A with the passed value.
ROL	Performs a rotate left operation on the bits of the value in register A. Unlike SHL the left most bit will be wrapped to the right side. Default is an 8 bit operation, though the number of bits can be specified as a parameter.
ROR	Performs a rotate right operation on the bits of the value in register A. Unlike SHR the right most bit will be wrapped to the left side. Default is an 8 bit operation, though the number of bits can be specified as a parameter.
SHL	Shift the binary bits, of the value in register A, left by 1 (default) or by the specified parameter. Left most bit will be discarded. Shift left by one essentially doubles the value.
SHR	Shift the binary bits, of the value in register A, right by 1 (default) or by the specified parameter. Right most bit will be discarded.
XOR	Perform an XOR operation on the value in register A with the passed value.

Variable Operators.

Instruction	Usage
CLR	Clears and zeros out the specified variable's memory block. After clearing the variable's memory block will be resized to a single '0' entry. The variables type will remain the same.
CP	Copy one variable to another. Destination variable will be changed to the same type as the origin. Source variable address is loaded into the A register and the destination a parameter on the CP instruction.
FILL	Fills a specified variable's data block with a value loaded into the A register.
LEN	Returns the length of the specified variable's memory block. Register A is loaded with the result. Result will be size -1 as data blocks are zero based.
RES	Resets the size of the specified variable's data block. Size must be an integer value 1 or greater. Variable type will remain unchanged. Any data in elements < size will be retained. Additional values will be set to zero. New size must be loaded into the A register. Reported new size will be size -1 as data blocks are zero based.

Logic / Control.

Instruction	Usage
CMP	Compares the passed value with the value of the A register and populates the A register with the difference. CMP is used exclusively with the conditional jump instruction in this table.
JG	Jump to the specified label if the value of the A register is greater than 0.

Instruction	Usage
JGZ	Jump to the specified label if the value of the A register is greater than or equal to 0.
JL	Jump to the specified label if the value of the A register is less than 0.
JLZ	Jump to the specified label if the value of the A register is less or equal to 0.
JNZ	Jump to the specified label if the value of the A registers is not 0.
JP	Perform an unconditional jump to the specified label.
JSR	Perform an unconditional jump to a subroutine and return when the RTS instruction is called. A return point will be placed on the execution stack.
JSRNZ	Jump to a subroutine when the value of the A register is not 0 and return when the RTS instruction is called. A return point will be placed on the execution stack.
JSRZ	Jump to a subroutine if the value of the A register is 0 and return when the RTS instruction is called. A return point will be placed on the execution stack.
JZ	Jump to the specified label if the value of the A register is 0.
LP	Jump to the specified label if the value of the C register is greater than or equal to zero. Value in the C register will be decremented by one on each loop. LP's specified label should occur earlier in the code than the LP
RTI	Return from an Interrupt called subroutine to the last program counter position pushed onto the execution stack. IRQ's will only occur if the _irq: label is present, the _irqTimeOut value is greater than 0 and the main program thread is not in a halted state waiting for a key press.
RTS	Return from a subroutine to the last program counter position pushed onto the execution when any of the JSRx instructions are used.

Maths.

Instruction	Usage
ABS	Change the value in the A register to the absolute value of itself.
ADD	Add the specified value, in the parameter or parameters location to the value in the A register.
CEIL	Round the value in the A register up to the nearest whole number.
COS / ACOS	Populate the A register with the cosine or arc cosine of it's current value.
DEC	Subtract one from the value in the A register
DIV	Divide the value in the A register by the value, in the parameter or parameters location.
INC	Add one to the value in the A register.
MOD	Populate the A register with the Modulus, remainder, of the division with the parameter or value within the referenced location.
MUL	Multiply the value in the A register with the value, in the parameter or parameters location.
NEG	Change the value in the A register to the positive or negative of itself.

Instruction	Usage
POW	Populate the A register with the result of a power operation on itself, with the passed parameter or value held in the parameters location.
RND	Populate the A register with a random number between 0 and the current value in the A register.
ROUND	Round the value in the A register up or down to the nearest whole number.
SIN / ASIN	Populate the A register with the sine or arc sine of it's current value.
SQRT	Populate the A register with the square root of the current value in the A register.
SUB	Subtract the specified value, in the parameter or parameters location from the value in the A register.
TAN / ATAN	Populate the A register with the tangent or arc tangent of it's current value
TRUNC	Remove the floating points from the value in the A register.

Registers.

Instruction	Usage
LDA	Load Accumulator. Loads the A registers with the value of or from the address specified by the parameter.
POP	POP the last value of the stack into the specified register. If no register is specified then A is assumed.
POPA	Restore the value of all registers from the stack
PUSH	PUSH the specified register onto the stack as a backup if the values is required later. If no register is specified as a parameter then A is assumed.
PUSHA	Push / Backup all the registers onto the stack.
STA	Store Accumulator. Stores the value in the A register in the location or address specified by the parameter and optional offset. The variable's memory block is zero based. When using another variable as an offset then the default '0' memory block address of the offset variable will be used. When storing values in variables the value may be modified to match the destinations variables type. IE. Byte, Integer or Double (Real).
STAA	Store Accumulator with Append. Appends the value held in the A register to the end of the specified variable's data block.
STAI	Store Accumulator with Insert. Inserts the value held in the A register at the start of the specified variable's data block.
STC	Copy the value in the A register into the C register.
STR	Converts the numeric value stored in the A register to a string, storing the result in the specified variable. Destination variable type will be changed to a '.DB' byte array.
XCH	Swaps the values in the two specified registers.

System.

Instruction	Usage
IN	Reads the oldest waiting key code from the keyboard buffer and places it in the A register. If the buffer is empty then 0 is assumed.
OUT	Prints the character defined by the value in the A register to the screen.
SYS	Makes a system call to the underlying operating system, such as setting terminal parameters.

Thread.

Instruction	Usage
BRK	Terminates execution of the current program.
HLT	Halts the current program and wait for a keypress to resume. If no value is specified then 30 milliseconds is assumed between keyboard buffer checks.
IRQ	Trigger a software generated interrupt. Will only occur if an _irq: label and timeout has been set and if the main thread is not in a halted state while waiting for a keypress or an IRQ is already active.
NOP	No Operation. Performs no operation on the current CPU cycle.
SLEEP	Sleeps the current program thread for a specified number of milliseconds. If no value is specified then 30 is assumed.
TICKS	Returns the number of TICKS (60th) of a second that have elapsed since the system was started. Value is placed in the A register.
YLD	Yield any remaining CPU time from the program thread back to the underlying operating system.

Compiler Directives.

Instruction	Usage
_IRQ:	The _irq: label is a special label used to indicate where IRQ code is located. The IRQ system will only activate when the label is present, the _irgTimeOutValue is greater than 0 and the main program thread is not in a halted state waiting for a key press.
_START:	The _start: is used to inform the compiler where to set the program counter when the program first executes. An error will be generated if no _start: label is found. The _start: label can be positioned anywhere in your program, it do not need to be at the start though most demos show it as such.
.DB	Creates a byte array variable consisting of a single number, comma separated values or a string literal enclosed in quotes. String variables will be automatically 0 terminated.
.DD	Creates a defined double (Real Number) array consisting of a single number or comma separated values. Real numbers are 64 bits.
.DW	Creates a defined words (Integer) array consisting of a single number or comma separated values. Integer values are 64 bits.

Instruction	Usage
.FILE	Links a text file into the compiler application. Text file will be converted to byte array variable containing the file contents. Variable buffer will be automatically 0 terminated.
.INCLUDE	Includes the contents of another ASM file or library into the one being compiled. When including a library file from the Libs directory the directory name can be ignored as the compiler will automatically check that path.
.MB	Creates a memory block of a specified length containing 64 bit real numbers.

Constants.

Instruction	Usage
TRUE	Equates to 1 when loaded into Register A
FALSE	Equates to 0 when loaded into Register A
PI	Equates to 3.14159 when loaded into Register A

Compiler.

Allowable Patterns.

The list below shows the allowable patterns the compiler will accept without error. All instructions must be entered on a single line, it is not possible to combine instructions on the same line.

Comments:
Comments can be placed on a line by themselves or on the same line after an instruction. All instructions regardless of placement are started with the '**;**' character. There is no scope for start and end block comments.

Compile Time Considerations:
Comments, compiler directives, label names and any other non-runtime instructions will be stripped from the final executable during compile. Unquoted spaces, or '**+**' plus signs will be replaced with commas during the compile process for ease of compilation. Any white space at the start or end of the instruction will also be stripped.

Parameters:

- $address is the address of a variable loaded with the LDA $VARNAME syntax
- A constant is a numeric value
- A variable is the actual variable name specified after one of the compiler '.' directives
- A register is one of the system registers "a, b, c, d, x, y, r0, r1'
- A pointer is a register containing the address of a variable with the register name enclosed in square brackets
- A label is a : colon appended name within your code used for logic operations. "_start:" and "_irq:" are examples of system specific labels.
- Any parameter specified after a '+' sign is an offset from the variable starting address. When no offset is specified then the default '0 / first' address of the variable's memory block will be used or modified. When using a variable as an offset then the default '0 / first' memory block value of the offset variable will be used.

Patterns:
The list below is ordered by the instructions bit code ID. For Example the bit code of NOP is '0' zero. The position in the list or bit code value has no affect on the speed of the specific operation.

```
nop
brk
inc
dec
in
out
lda $address
lda constant
lda register
lda variable
lda variable constant
lda variable register
lda variable variable
lda [pointer]
```

lda [pointer] + constant
lda [pointer] + register
lda [pointer] + variable
sta register
sta variable
sta variable + constant
sta variable + register
sta variable + variable
sta [pointer]
sta [pointer] + constant
sta [pointer] + register
sta [pointer] + variable
lp label
jp label
jz label
jnz label
jl label
jlz label
jg label
jgz label
jsr label
jsrz label
jsrnz label
rts
rti
cmp $address
cmp constant
cmp register
cmp variable
cmp variable + constant
cmp variable + register
cmp variable variable
cmp [pointer]
cmp [pointer] + constant
cmp [pointer] + register
cmp [pointer] + variable
len variable
len [pointer]
push register
pop register
xch register, register
sub constant
sub register
sub variable
sub variable + constant
sub variable + register
sub variable + variable
sub [pointer]
sub [pointer] + constant
sub [pointer] + register
sub [pointer] + variable

add constant
add register
add variable
add variable + constant
add variable + register
add variable + variable
add [pointer]
add [pointer] + constant
add [pointer] + register
add [pointer] + variable
mul constant
mul register
mul variable
mul,variable + constant
mul,variable + register
mul,variable +variable
mul [pointer]
mul [pointer] + constant
mul [pointer] + register
mul [pointer] + variable
div constant
div register
div variable
div variable + constant
div variable + register
div variable + variable
div [pointer]
div [pointer] + constant
div [pointer] + register
div [pointer] + variable
shl constant
shr constant
neg
abs
trunc
rnd
ticks
irq
sys constant
yld
sleep constant
ceil
round
sqrt
mod
pow
cos
acos
sin
asin
tan

atan
xor
or
and
not constant
pusha
popa
str variable
str [pointer]
cp variable
cp [pointer]
ror constant
rol constant
hlt constant
stc
clr variable
clr [pointer]
res variable
res [pointer]
fill variable
fill [pointer]
staa variable
staa [pointer]
stai variable
stai [pointer]