UASM v2.36

Extended User Manual

1. Introduction

JWasm has been an incredible assembler, especially for software development under Windows and one which we have used for research, personal and commercial projects. We believe strongly that it has a place in a modern development environment and can be used to create high-performance functions, libraries (which can be used from higher level languages) as well as be a compelling alternative for the development of entire solutions.

With the loss of support and development on JWasm as well as the desire to expand the existing product we have taken it upon ourselves to continue the product's evolution under a new name Uasm. We have tried to remain true to its roots and as such are continuing from the existing version numbering and reflect its origin in the name.

As the core usage, command lines parameters and descriptions of existing functionality remains unchanged from JWasm we include all of the original JWasm documentation and examples and describe only the new or enhanced features in this document.

2. History

Version	Changes
2.36	Added automatic alignment check for all SIMD instructions which are not scalar. Improved simd type coercion by testing variables (global/local) on size rather than type. This simplifies code and reduces the need to use type ptr when working with DWORD, BYTE or other data types. Fixed a bug with literal string support handling \n escape. Improved handling of PTR and VARARG for SystemV calls.
2.35	Fixed some register combination warnings for VCMPSD, VCMPEQSD and VCMPSS. Added RP4/RP8 macros and case-sensitive conversion to macro library. Added support for floating point comparison expression. Add string literal support back into invoke.
2.34	Added –Sp command line switch to allow user specified segment alignment. Added further Intel MPX extension support, including BND prefix support. Added support to define return types on PROC/PROTO which enables the use of the new UINVOKE macro library function.
2.33	Fixed a number of regression issues, including several VEX encoded instructions which were not correct. Fixed ebx.STRUCT style addressing. Fixed SYSV support to allow ADDR in VARARGS. Fixed a potential crash when running multiple input files.
2.32	Added initial UTF8 BOM Check Rolled-back literal string support in favour of using existing CSTR/WSTR macros to ensure no legacy invoke calls are broken. Literal string support will be moved to a new invocation syntax.
2.31	Fixed 32bit MS FASTCALL issue. Fixed encoding bug with VMOVQ. Fixed VMOVSS/VMOVSD allowing 2 register form, when 3 registers are required for all-register mode and added error message for it. Updated VECTORCALL, FASTCALL x64 to support 3 register VMOVSS/VMOVSD form if archAVX on.

	Optimised and improved FASTCALL handlers and completely redeveloped SYSTEMV calling convention implementation. Fixed a potential memory corruption bug in the symbol table. Added support for Intel MPX extensions, bnd0-3 registers and new instructions. Added custom USES to OO Library method definitions as well as providing a replaceable set of memory allocation macros to allow for OO use on OSX/Linux.
2.30	Fixed a nested path bug present from JWASM which caused strange assembly errors when working a nested path location. Fixed MS Fastcall to support types other than DWORD and prevent incorrect warnings about incompatible types or redefinitions. Improved Linux 64bit package and compilation to use clang as per OSX so both builds are compatible.
2.29	Finalized Delphi (Borland Register calling convention) and updated language type specifier to BORLAND. Fixed some register overwrite issues with SystemV invoke. Added two new Macro Library functions, MEMALLOC <size> and MEMFREE <ptr> Added new built-in variable @Platform <0,1,2,3,4,5> which indicates the current output type of WIN32, WIN64, ELF32, ELF64, OSX to support cross-platform development. Fixed a potential general failure crash with some evex encodings when no base-register present. Fixed 32bit crash caused by Macro Library modifying read only string data.
2.28	Optimise SYSTEMV Invoke code generation. Fixed register overwrite warnings for SYSTEMV invoke. Fix long standing JWasm issue with source debugging information leaving out PROC source-lines when a procedure has no prologue. Add OPTION PROC:NONE, OPTION PROC:DEFAULT to simplify switching on/off procedure prologue/epilogue. OPTION PROC can also set prologue/epilogue simultaneously with: OPTION PROC:PrologueMacro,EpilogueMacro Change macro library OO functions names to use _ instead of $. Added DELPHI language type for Borland 32bit fast-calling convention. *(experimental, pending further updates)*

	Fix SHR for macro expression evaluation (Fix supplied by Nidud). Fix VARARG Register overwritten warning. Added progressive assembly reporting, console updates per source file. Added new command line option –**less** to reduce console output information.
2.27	Fixed the use of ADDR in invoke for SYSTEM V calls. Disable generation of listing while processing built-in macro library. Rename OO INTERFACE function in built-in macro library to OINTERFACE and ENDINTEFACE to ENDOINTERFACE to prevent conflicts with SDK headers.
2.26	Added support for USES ymm registers to stackbase:RBP 64bit mode. Reduced stack usage for stackbase:RSP with win64:15 mode Configured 64bit assembly to assume stackbase:rbp if nothing specified.
2.25	Continued improvement of align 16 in prologue, while reducing stack wastage. Implemented OPTION REDZONE for SystemV ABI Internal optimisation to character validation and hashing functions to yield 5% assembly time improvement. Extend macrolib functionality and limit to 64bit. Implement OO language extension framework.
2.24	Fixed literal strings including “ “. Fixed stackbase:rsp overwriting locals in some cases. Ensure stackbase:rbp works for modes with bit 1 set. Add support for SYSTEMV ABI.
2.23	Support for STACKBASE:ESP has been fixed (Big thanks to MASM32 forum member Nidud) Command line switch –nomlib added to disable internal Macro Library OSX Build supplied Fixed INCBIN related offset bug Fixed COFF 32bit name mangling bug Code paths for 64bit RSP and RBP stackbase have been completely separated, refactored and optimised.

	Simplification of code generation options
2.22	Added support for inline string literals in INVOKE include L"" wide character strings. Added support for allocating wide strings in data with dw. Automatic integer to float promotion for data declarations and struct members. Added new union initialisation syntax. Added built-in macro library that precompiles and adjusts to suit currently selected architecture setting (SSE/AVX). Fixed STACKBASE:ESP parameter positions. Fixed STACKBASE:RSP + WIN64:11 16byte alignment. Fixed STACKBASE:RSP parameter positions.
2.21	Fixed RBP prologue and epilogue for win64:6 and 7 modes. Added support to INVOKE allowing XMM register parameters for FLOAT type arguments (real4 and real8). Fixed stack and local align to 16 issues. Added OPTION ARCH:<SSE|AVX> and command line switches –archSSE –archAVX to change code-generation to favour SSE or AVX instructions (this is particularly relevant to invoke and prologue/epilogue generation).
2.20	Fixed opcode byte 83 to be 80 when assembler needs to assume BYTE type. Cleaned up linnum memory allocation from switch additions. Fixed option win64:6
2.19	Fixed corruption of structures and first proc with offset 0 bug when dumping symbols to file. More AVX, AVX512 and MMX instruction encoding fixes. Fixed line number debugging information corruption due to compiler check for memory address allocation. Fixed the use of label in data section which was causing an erroneous error of "USE OF REGISTER ASSUMED TO ERROR". Investigated custom epilogue macro expanding on a line with a label causes backwards jumps not to evaluate distance correctly. This is not being fixed but is noted, always put RET on its own line without a label when using a custom epilogue macro.

2.18	Fixed '&&' in hll (.while, .if, .for, .repeat), added additional label Fixed wrong locals alignment in 32 bit Fixed error reporting for overwritten registers Changed cmp reg,0 to test reg,reg for HLL generation
2.17	AVX 3 operand warnings for missing operands. Support for integer values to REAL4, REAL8, REAL10 and thus FP4/8/10 macros and structures. Ideal for vectors and structure initialisation like <0, 0, 0, 1> with REAL4 typing. Simplified symbolic debug data output via new –Fs command switch (with included file format guide) Option flat added which enables a lower-level assembly (like fasm) and removes the need for end directive - Adds support for USE16, USE32, USE64 to switch code-generation bit mode. - Maintains all UASM 64bit HLL features like PROC, FRAME, INVOKE, .FOR, .SWITCH, VECTORCALL etc. - See new example flat.asm Fixup ORG directive generation which incorrectly pads the code buffer with 0's instead of simply applying the address change Add support for .labelName style labels (like nasm/fasm support)
2.16	Complete re-working of AVX2 and AVX512 code generation to ensure that encodings are separated. Complete implementation of all missing AVX2 and AVX512 instructions. Added extensive checking to prevent misuse of register and memory operands with AVX, AVX2 and AVX512 instructions especially where mask and decorator flags are required. All AVX512 code now tested against Intel SDE emulation libraries.
2.15 r2	Fixed AVX2 encoding bugs. Fixed EVEX type coercion and added automatic conversion of vmovdqa to vmovdqa32/64. Fixed legacy jwasm issue of "ADD RSP,0" in epilogue.

	Fixed a number of format compatibility options for WIN64 between UASM and JWASM for WIN64:1, no WIN64, WIN64:3. Unified the stack handling between WIN64:3 and RSP and have dropped any support for WIN64:2
2.15	Allowed xmm, ymm and zmm registers to be saved at the same time with USES. Fixed problem with the reserved stack size Implemented VECTORCALL (R-VECTORCALL) Implemented SSE compatibility with ML64 (automatic xmmword type promotion with switch –Zg) Fixed the bug with MOVSS Fixed EIP/RIP encoding bug Added @ProcLine built-in equate which matches MASM/ML's @Line value during user-defined prologue execution.
2.14	Added .switch support Fixed 0x c style tokenization bug with immediate values Fixed some bugs pertaining to stack prologue generation in 64bit.
2.13	First official release of Uasm, changes reflect enhancements to JWasm. New instructions added: RDRAND, RDSEED, F16C(VCVTPH2PS, VCVTPS2PH), MOVQ and VMOVQ to supplement MOVD and bring instructions in line with Intel and AMD reference guides. Removed warnings relating to alignment by 32 (as this is a requirement for AVX vectors), leads to clean builds. Additional unsigned flag comparison predicates added for HLL syntax (ABOVE, BELOW) Added support for AVX512F, ZMM registers and EVEX encoding. RIP register support. .FOR high level macro support and enhancements. VEX encoded general purpose instructions added. Support for more optimised stack and invoke usage as well as stack alignment and RSP stack base. Fixed bug requiring the use of xmmword ptr and ymmword ptr. Added OPTION EVEX to enable encoding of evex instructions and support for AVX512 registers. Added OPTION ZEROLOCALS to clear all PROC local values to 0 on entry.

	Fixed .WHILE loop issues present in JWASM. Added VGATHERDPD, VGATHERQPD, VGATHERDPS, VGATHERQPS, VPGATHERDD, VPGATHERQD, VPGATHERDQ, VPGATHERQQ ,VCMPxxPD, VCMPxxPS, VCMPxxSD, VCMPxxPD, VCMPxxSS.

3. Roadmap

Version	**Possible Features**
2.36	Allow PROC definitions to overload based on parameters. Add –macho64 output format support.

4. New or Enhanced Features

4.1) Flag-based comparison predicates

The flag-based comparison predicates allow you to generate higher level .IF syntax which will generate the corresponding Jcc instructions based on the condition type and any previous comparison. This allows the higher level syntax to be used with floating point and SIMD based comparison instructions.

Comparison Predicate	Signed/Unsigned	Equivalent Branch
LESS?	Signed	JGE <label>
GREATER?	Signed	JLE <label>
ABOVE?	Unsigned	JBE <label>
BELOW?	Unsigned	JAE <label>
CARRY?	N/A	JNC <label>
EQUAL?	Both	JNE <label>
ZERO?	N/A	JNZ <label>
OVERFLOW?	N/A	JNO <label>
SIGN?	N/A	JNS <label>

Comparison predicates can also be combined with a ! (NOT) in-front.

Example(s):

```
cmp eax,ebx
.if(EQUAL?)      ; Execute this block if eax == ebx
.endif

vucomiss xmm0,xmm1
.if(!BELOW?)     ; Execute this block if xmm0 >= xmm1
.endif

test rax,rax
.if(ZERO?)       ; Execute this block if rax == 0
.endif
```

4.2) HLL .FOR / .ENDFOR loop

The syntax has been updated to use : instead of | as a separator as per previous implementation in JWasm. The code generation has been optimised and now supports a wide range of initializers, modifiers and conditional operators:

Initializers	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=,++,--,<constant>

Conditional operators	==,!=,> ,< ,>=,<=,&&,||,&,!,ZERO?,CARRY?,SIGN?,PARITY?,OVERFLOW?,LESS?,GREATER?,ABOVE?

```
.for (edx=88, ecx=4 : eax != 24 && hWnd > lParam || ebx <= 20 || ebx >= 3 : eax=23, edx=24, ebx++)
    nop
.endfor

.for (rcx=0 : rcx<rdx : rcx++, rbx<<=4)

.endfor

.for (:r8:r8++,[rcx].RECT.top=eax)
    .if (rax)
        .continue
    .endif
    mov[rcx],dl
.endfor

; infinite loop
.for (::)
    .break .if eax
.endfor
```

4.3) Shadow Space Optimisations

Although the first four parameters are passed via registers, there is still space allocated on the stack for these four parameters. This is called the parameter homing space or shadow space and is used to store parameter values if either the function accesses the parameters by address instead of by value or if the

The minimum size of this homing space is 0x20 bytes or four 64-bit slots, even if the function takes less than 4 parameters.

- When the homing space is not used to store parameter values, Uasm uses it to save non-volatile registers (as specified by the USES clause).
- If a procedure doesn't have invoke it will not unnecessarily allocate the homing space.
- If a procedure doesn't use locals a FRAME will not be created so you will not need to use:

OPTION PROLOGUE:NONE
OPTION EPILOGUE:NONE

Procedure declaration…

OPTION PROLOGUE:PrologueDef
OPTION EPILOGUE:EpilogueDef

4.4) INVOKE Optimisation

If any of the first 4 parameter values are FALSE, NULL or 0 they will now be generated via XOR instead of MOV reg,0. In addition of other parameters after the fourth are compatible they can be written using the same XOR'ed register(s).

invoke testproc5, NULL,FALSE,NULL, 0,0, rdx

```
000000013FB618F0 33 C9             xor       ecx,ecx
000000013FB618F2 33 D2             xor       edx,edx
000000013FB618F4 45 33 C0          xor       r8d,r8d
000000013FB618F7 45 33 C9          xor       r9d,r9d
000000013FB618FA 48 C7 44 24 20 00 00 00 00 mov       qword ptr [rsp+20h],0
000000013FB61903 48 89 54 24 28    mov       qword ptr [rsp+28h],rdx
000000013FB61908 E8 76 00 00 00    call      testproc5 (013FB61983h)
```

Note: RDX being reused for zero in the sixth parameter.

4.5) General Procedure Prologue/Epilogue Optimisation

Uasm will now automatically make the best use of the available shadow space entries, parameters, locals to not only maintain alignment but to avoid executing any unnecessary prologue code sequences. For example if parameters are unused they won't be saved to shadow space. If no invokes are used within the procedure, automatic stack reservation will not be applied and so on. This allows all optimal procedure forms to be generated using the standard PROC FRAME USES arrangement and syntax.

4.6) RIP Register Support

Examples:

```
displacement EQU 200h

mov ah,[rip]
mov rax,[rip+3]
mov rax,[rip+400h]
mov cx,[rip+128]
mov [rip+127],cx
mov [rip+displacement],rbx
mov rbx,[rip+displacement]
mov rax,[rip]
mov [rip+1],sil
cmp byte ptr [rip], 90h
lea rbx,[rip]
lea rax,[rip+2]
call qword ptr [rip+400]
push [rip]
push [rip+80h]
pop [rip]
```

4.7) .SWITCH

Uasm now supports generation of optimised switch statements using either .if .else for short cases, jump tables or non-recursive binary search.

Example:

```
mov eax,280
.switch eax
        .case 273
                mov  edx,273
                .break
        .case 280
                mov  edx,280
                .break
         .case 275
                mov  edx,275
                .break
         .default
                mov edx,0
                .break
.endswitch
```

A new option is provided which allows the tradition C style of switch requiring (.break) and ASMSTYLE which does not.

option SWITCHSTYLE : ASMSTYLE
option SWITCHSTYLE: CSTYLE

Multiple cases can be combined as with:

```
 mov eax, 184h
   .switch eax
   .case 179h,180h,1c5h,17bh,17dh,182h,184h,185h
     mov edx,1d5h
   .case 1d3h
     mov edx, 1d3h
   .case 1f4h
     mov edx, 1f4h
   .case 200h
      mov  edx, 200h
   .default
     mov edx, 0
.endswitch
```

.default case can be omitted if not required.

Switch supports immediate numeric values as well as character immediates such as 'A', 'AB','ABCD','ABCD1234' etc.

4.8) VECTORCALL

Vectorcall calling convention support has been added for 64bit Windows targets.

To specify that a procedure should use this calling convention instead of the usual FASTCALL a new language type specifier has been added which must be included on both the PROTO and PROC as follows:

TestVectorcallProc PROTO **VECTORCALL** a:__m128

TestVectorcallProc PROC **VECTORCALL** FRAME a:__m128

The vector call convention is slightly more complex than fastcall so attention should be paid to how arguments are passed. If any of the first 4 arguments are integers, they are passed in RCX-R9 as per fastcall. If any of the first 6 arguments are floating point they're passed in XMM0-XMM5. The home space is increased to store 6 entries. Vectorcall supports passing of SIMD types by value in register rather than by reference and these are passed in registers XMM n or YMM n depending on the argument position. Vectorcall also introduces the concept of HFA (homogenous float array) and HVA (homogenous vector array). These types are populated into any empty registers after integer/floating point and vector types have been dealt with. If there aren't sufficient free registers then the entire HFA or HVA is passed by reference.
An HFA is any structure containing 1-4 floats or doubles (REAL4 or REAL8).
An HVA is any structure containing 1-4 valid SIMD type vectors (so you can think of it as a matrix).

HFA and HVA types are populated into registers component wise, so a 3 element float HFA passed as a single argument to a procedure would put the first float element into XMM0, the second into XMM1 and the final one into XMM2.

Due to the fact that a 4 element HFA is technically also a SIMD type, UASM has to make provision to determine the intended application and as such we now provide an include file of standard SIMD data types. We rely on the naming convention specifically to ensure that a vector is handled appropriately and not treated as an HFA. The types have been named as per the C/C++ convetion: __m128 (the union) with specifically typed variants __m128f, __m128i etc.

UASM will allow you to declare a PROC that a type, for example the union __m128 but will determine how to use on a per-invoke basis. So for example it would be completely valid to have a PROC declared that expects a vector type __m128f and then pass it a variable declared as a 4 element float HFA. In this instance the HFA will be treated as a vector by that procedure, so you can consider it automatic type coercion where compatible.

As with FASTCALL UASM will automatically optimise the prologue to remove any unused argument. Ideally to obtain the performance benefit intended with VECTORCALL one should use the arguments directly from registers where possible. In the C/C++ version the homespace for vectorcall arguments is unused, however in UASM we will populate the stack with HFA/HVA/Vector elements if the argument is referenced and the substitute the memory address of the structure into the homespace provided.

Uasm implements the Vectorcall convention in a slightly different way to C/C++ in that all parameters passed by reference point to their original variables, we call this Referential Vectorcall

or R-VECTORCALL. This is completely compatible with C/C++ however special attention must be made for the following:

1) When calling an Assembly language RVECTORCALL function from C:
   Any modification of a parameter will only modify the local copy and not the original.
2) When calling a C VECTORCALL function from Assembly language, the function may modify variables which are assumed to be local copies thus modifying your original, so in these cases a copy should be made prior to invoking the function.

This choice was made to ensure optimal efficiency when using RVECTORCALL functions written in UASM from UASM as referential passing is far more efficient that making repeated local copies of entities on the stack.

For further information on Vectorcall please refer to:

https://msdn.microsoft.com/en-us/library/dn375768.aspx

https://blogs.msdn.microsoft.com/vcblog/2013/07/11/introducing-vector-calling-convention/

A new example source has been included to demonstrate the use of these structures, types and calls.

4.9) String Literal Support

Wide character literal data can now be declared with:

```
awideStr dw "wide caption ",0
```

String literals can be used directly in INVOKE using “” or L””. For a string literal to be accepted as such, the corresponding procedure parameter must be defined as **PTR**. Any other type will expect a character constant or numerical value. String literal support is switched off by default but can be enabled with : **OPTION LITERALS:ON**

4.10) Integer to Real type promotion

Certain data declarations required a real number literal to be supplied, this included REAL4, 8 and STRUCT members. Integer values can now be used in place:

```
; Automatic type promotion from integer to float
aReal REAL4 2

; This is example of initializing a union with floats (first sub-type)
; using normal syntax as well as hjwasm 2.17 update to promote integer literal to float
myVec1 __m128 { < 1.0, 2.0, 3.0, 4.0 > }
myVec2 __m128 { < 1, 2, 3, 4 > }
```

This can be quite convenient when using zeroes or identity rows in vectors and matrices.

4.11) New UNION syntax to specify sub-type initialization values

Previously a union could only be initialized using elements compatible with the first type. To overcome this limitation especially when using SIMD type structures as included in the supplied xmmtypes.inc a specific sub field can be specified.

Given the structures and union as follows:

```
__m128f struct
        f0 real4 ?
        f1 real4 ?
        f2 real4 ?
        f3 real4 ?
__m128f ends

__m128q struct
        q0 QWORD ?
        q1 QWORD ?
__m128q ends

__m128 union
        f32 __m128f <>
        q64 __m128q <>
__m128 ends
```

The following is now possible:

```
; Hjwasm 2.22 enhanced union type (now allows direct specification of sub-type to use in initialisation):
myVec4 __m128.f32 { < 1.0, 2.0, 3.0, 4.0 > }   ; you can try .f33 and hjwasm will emit an error when testing for valid sub-type.
myVec3 __m128.q64 { < 0x1234, 0x5678 > }
```

4.12) Integrated MACRO library

Uasm now provides a library of built-in macros which will automatically re-target to the current architecture setting (either SSE or AVX). This library can and will grow over time but for now includes:

CSTR – ASCII inline string literal for invoke

WSTR – Wide string literal for invoke

FP4, FP8, FP10 - declare a real data element of specified size and load it directly to a register
IE: movss xmm0,FP4(2.2)

RV – Insert rax return value into invoke.

MEMALIGN reg,number – Align value in reg to multiple of number

LOADSS, LOADSD, LOADPS – optimised and architecture specific load immediate float value to register, IE:
LOADSS xmm0, 2.3
LOADPS xmm2, 3.141

MEMALLOC <size>
MEMFREE <ptr>

UINVOKE (refer to 4.18 for example) – automatic inline form of invoke that supports determination of return type.

R4P, R8P (refer to 4.19 for example) – in place real4 and real8 type cast.

The integrated Macro Library can be disable with command line switch **–nomlib** if required.

4.13) OPTION ARCH:{SSE|AVX}

This setting determines which instruction set should be used for any automatically generated code. This includes prologue, epilogue, invoke as well as the built-in macro library.

For example the LOADSS built-in macro would be coded under SSE as :

mov eax,floatLiteral
movd xmmReg,eax

But under AVX it would use vmovd instead.

This setting is also available through command line switches –archSSE and –archAVX.

The default setting is to use the AVX instruction set.

The currently selected architected is also available through the built-in variable @Arch.

4.14) SIMPLIFIED CODE GENERATION

When using STACKBASE:RSP, frame:auto, win64:11 are now implied. Procedures with default settings will automatically be generated as FRAME procedures.

Using STACKBASE:RBP (or not specifying at all as this is the default) will imply frame:auto and default procedures will be generated as FRAME procedures. Win64 options 1-7 are valid.

In both cases the first local is aligned 16 and both options now implement optimisations to store only used parameters to home space.

STACKBASE:RBP will automatically optimise away the frame-pointer if no parameters or locals are referenced.

If the procedure is a leaf procedure and makes no use of locals or further invokes the add/sub rsp instructions will also be optimised out automatically.

4.15) NEW LANGUAGE TYPES

Procedures can now be decorated with attribute SYSTEMV to generate a 64bit SystemV ABI call.

For example:

```
OPTION ARCH:SSE
nixproc PROC SYSTEMV FRAME USES rbx xmm0 arg1:qword, arg2:DWORD, arg3:REAL4
    mov rbx,arg1
    mov ecx,arg2

IF @Arch EQ 0

            movss xmm10,arg3

ELSE

             vmovss xmm10,arg3

ENDIF

     ret
nixproc ENDP
OPTION ARCH:AVX
```

This produces the correct invoke, prologue and epilogue for use on 64bit Linux and OSX.

In addition a new OPTION REDZONE:{YES|NO} is provided to enable or disable the use of SystemV ABI Red-Zone optimisation.

The language type BORLAND has also been added to support Delphi / Free Pascal style register based fastcall.

4.16) OPTION PROC Extensions

OPTION PROC now allows for the specification of prologue and epilogue macro in a single statement, as a form of short-hand for OPTION PROLOGUE, OPTION EPILOGUE combined. It also allows you to specify the types NONE and DEFAULT.

OPTION PROC:NONE
is equivalent to the pair:
OPTION PROLOGUE:NONE
OPTION EPILOGUE:NONE

and

OPTION PROC:DEFAULT
is equivalent to the pair:
OPTION PROLOGUE:PROLOGUEDEF
OPTION EPILOGUE:EPILOGUEDEF

For custom combinations use:

OPTION PROC:MyPrologueMacro,MyEpilogueMacro

4.17) Additional Built-In Variables

@ProcLine, indicates the current source code line number relative to the start of the current procedure.

@Platform <0|1|2|3|4|5> indicating the currently platform target as WIN32, WIN64, ELF32, ELF64, OSX to support improved cross platform assembly.

```
   IF @Platform EQ 0
       xor eax,eax
        echo 'im 32bit windows'
    ELSEIF @Platform EQ 1
       xor rax,rax
       echo 'im 64bit windows'
    ELSEIF @Platform EQ 2
       xor ebx,ebx
       echo 'im 32bit linux'
    ELSEIF @Platform EQ 3
       xor rbx,rbx
       echo 'im 64bit linux'
    ENDIF
```

@LastReturnType, indicates the last function return type following an invoke call. Values are:

```
enum returntype {
       RT_SIGNED = 0x40,
       RT_FLOAT = 0x20,
       RT_BYTE = 0,
       RT_SBYTE = RT_BYTE | RT_SIGNED,
       RT_WORD = 1,
       RT_SWORD = RT_WORD | RT_SIGNED,
       RT_DWORD = 2,
       RT_SDWORD = RT_DWORD | RT_SIGNED,
       RT_QWORD = 3,
       RT_SQWORD = RT_QWORD | RT_SIGNED,
       RT_REAL4 = RT_DWORD | RT_FLOAT,
       RT_REAL8 = RT_QWORD | RT_FLOAT,
       RT_XMM = 6,
       RT_YMM = 7,
       RT_ZMM = 8,
       RT_PTR = 0xc3,
       RT_NONE = 0x100
};
```

4.18) Procedure Return types and UINVOKE

It is now possible to specify the return data type on procedures and prototypes as follows:

```
myfuncR4 PROTO REAL4 :REAL4

myfuncD  PROTO DWORD :REAL4

myfuncQ  PROTO SQWORD :REAL4

myfuncX  PROTO XMMWORD :REAL4
```

The return type must immediate follow PROTO and unlike arguments takes no name and doesn't use a colon.

The matching procedure definitions are then:

```
myfuncR4 PROC REAL4 FRAME a:REAL4

myfuncD PROC DWORD FRAME USES rbx a:REAL4

myfuncQ PROC SQWORD USES rbx a:REAL4

myfuncX PROC XMMWORD a:REAL4
```

The return type must immediate follow PROC and preceed any other attributes such as FRAME, USES. Just like PROTO it requires no colon or name.

With the return-type specified, the following is now possible:

```
        vcmpss xmm0, uinvoke(myfuncR4, xmm2), xmm1, 0

        mov eax, uinvoke(myfuncD, xmm2)

        cmp rbx, uinvoke(myfuncQ, xmm3)

        vmovaps xmm0, uinvoke(myfuncX, xmm2)
```

The UINVOKE Macro Library function makes use of the @LastReturnType variable and can correctly return the relevant register to match the return type of the procedure and calling convention, IE: RAX, XMM0, etc.

4.19) HLL Floating Point Comparisons.

.IF , .ELSEIF , .WHILE have been extended to support floating point type comparisons. .FOR however supports only the classic integer arguments as before.

Examples:

```
.if(xmm0 < FP4(1.5))
.endif

.if(xmm0 > FP4(2))
.endif

.if(xmm0 < floatvar1)
.endif

.if(xmm0 == floatvar2)
.endif

.if(xmm0 < [rdx])
.endif

.if(xmm0 == [rdx+rbx])
.endif

.if(xmm0 < xmm1)
.endif

.if(xmm0 > xmm1)
.endif

.if(xmm0 <= xmm1)
.endif

.if(xmm0 == xmm3)
.endif

.if(real4 ptr xmm0 < xmm1)
.endif

.if(xmm0 < real4 ptr xmm1)
.endif

.if(R4P(xmm0) < xmm1)
.endif

LOADSD xmm0, 1.0
LOADSD xmm1, 2.0
LOADSD xmm3, 1.0

.if(xmm0 < doublevar1)
.endif

.if(xmm0 == doublevar2)
.endif

.if(real8 ptr xmm0 < FP8(1.5))
.endif

.if(xmm0 < real8 ptr FP8(1.5))
.endif

.if(real8 ptr xmm0 < xmm1)
.endif

.if(real8 ptr xmm0 > real8 ptr xmm1)
```

```
.endif

.if(real8 ptr xmm0 <= xmm1)
.endif

.if(xmm0 == real8 ptr xmm3)
.endif

.if(xmm0 == r8p(xmm3))
.endif
```

5. OPTION WIN64 Extensions

When using OPTION WIN64:***n***, the bits of ***n*** define optional configuration parameters for code generation in 64bit mode. The bit fields are as follows:

```
W64F_SAVEREGPARAMS = 0x01, // 1=save register parameters in shadow space on proc entry
W64F_AUTOSTACKSP   = 0x02, // 1=calculate required stack space for arguments of INVOKE
W64F_STACKALIGN16  = 0x04, // 1=stack variables are 16-byte aligned; added in v2.12
W64F_SMART         = 0x08, // 1=takes care of everything (Uasm)
```

Japheth introduced W64F_STACKALIGN16 in v2.12 because he was storing locals in reverse order: the first local was stored the last, so it wasn't possible to have the first local aligned 16.

This has been changed it so that the first local is first after the stack homing area and is guaranteed to always be aligned to a 16 byte boundary.

With this feature it is possible to keep the LOCALS you need to have aligned to 16 bytes as the first ones and avoid using this bit flag.

So in general one would use a value of 11 for ***n*** *(or 15 if additional LOCAL alignment is required).*

- AVX (ymmword) or AVX512 (zmmword) LOCALS will be automatically aligned on the stack too.

6. OPTION EVEX

A new OPTION EVEX:{0|1} has been added to enable the assembly of AVX512 code and extended registers (ZMM0 – ZMM31, XMM/YMM 16-31).

7. OPTION ZEROLOCALS

A new OPTION ZEROLOCALS:{0|1} has been added to clear LOCAL values declared inside a PROC to zero. A threshold is set so that the generated code will use immediate moves if only a few locals are present, for larger local allocation on the stack string reps are used.

8. Building with Visual Studio

Copy the Uasm targets archive contents to:

C:\Program Files (x86)\MSBuild\Microsoft.Cpp\v4.0\V120\BuildCustomizations

(**Note**: the version of the folder and exact location will depend on your version of Visual Studio).

You should put Uasm.exe in this folder:

C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin\x86_amd64

(Note: Rename Uasm32.exe or Uasm64.exe to Uasm.exe when copying into this location).

9. Support and Contact

For any information, queries and general assembly language guidance please visit the MASM32 forum and Uasm related boards at : http://www.masm32.com/board/index.php

10. Thank You

We'd like to thank everyone in the assembler community for continuing to support the language and related products and hope you enjoy using Uasm!

A few people deserve special mention:

Agner Fog for providing ObjConv and assisting with bug fixes during the testing of EVEX encodings.
Japheth for providing JWasm and all his years of effort on the Assembler.

Happy Coding!

Branislav Habus and John Hankinson