UASM v2.42

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. New or Enhanced Features

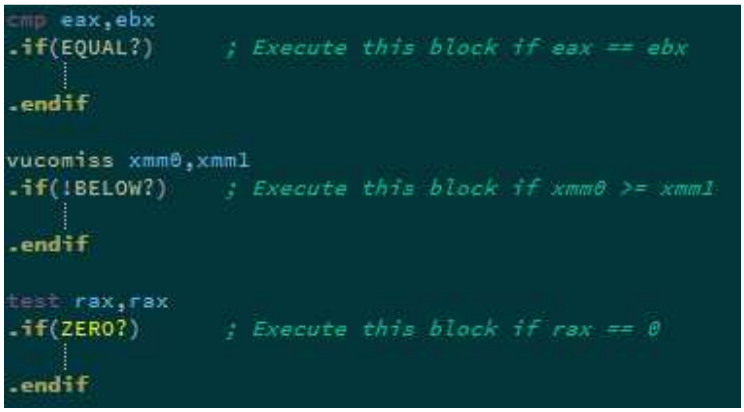
2.1) Flag-based comparison predicates

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):



2.2) 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	
Conditional operators	

```
.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
```

2.3) Shadow Space Optimisations

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…

2.4) 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.

2.5) 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.

2.6) RIP Register Support

```
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]
```

2.7) .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.

In addition there is also:

OPTION SWITCHSIZE:SIZE.

The default is 0x4000 and is limited to a maximum of 0x8000.
The purpose is to allow you to customise the performance/size trade of switch generated jump tables.

2.8) 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.

2.9) 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**

Declaring wide string data with dw will only happen with OPTION LITERALS:ON and using command line switches –Zm or –Zne will disable this.

2.10) 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.

2.11) 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 > }
```

2.12) 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:

Name	32bit	64bit	Description
CSTR	x	x	Literal text inline for use with invoke. Usage: Invoke SomeFunction, CSTR("This is literal text")
WSTR	x	x	Same as CSTR but for wide string data. Both of these are provided for backwards compatibility with MASM as UASM features direct support for string literal data in invoke.
FP4	x	x	Declare a real data element of the specified size inline. Usage: Movss xmm0,FP4(2.3)
FP8	x	x	As above for Double instead of float.
FP10	x	x	Full 80bit precision data item for use with FPU.
RV	x	x	Insert return value into EAX or RAX in a nested INVOKE call.
MEMALIGN	x	x	Round a number up to the next multiple of number. Usage: MEMALIGN <reg>,<number>
LOADSS	x	x	Optimised and architecture setting dependant load of an immediate float value to a simd register. Usage: LOADSS xmm0, 2.2 Uses a temporary register (EAX/RAX).
LOADSD		x	Same as above for double sized data.
LOADPS	x	x	Same as LOADSS but broadcasts to all elements.
MEMALLOC	x	x	Allocate memory using C runtime malloc. This primarily to provide a cross-platform way to allocate memory to be used by the OO library. Usage: MEMALLOC <size>
MEMFREE	x	x	Release memory via C runtime. Usage: MEMFREE <ptr>
UINVOKE		x	(refer to 2.18 for example) – Automatic inline form of invoke that supports determination of return type.
R4P	x	x	(refer to 2.19 for example) – To be used in place of real4 or real8 ptr with floating point HLL comparison. Alias name AsFloat.
R8P	x	x	Alias name AsDouble.
MOV64	x	x	Move a 64bit immediate value to memory. Usage: Mov64 <someVariable>, <immediate64>
MOV128	x	x	Move 128bit value to memory Mov128 dst, immLo64, immHi64

MOVXMMR128	x	x	Load 128bit value to an xmm register Movxmmr128 dstReg, immLo64, immHi64
SLXMMR		x	Shift left xmm register by bit count SLXMMR xmmreg, cnt
SHIFTLEFT128		x	Shift memory location 128bit left by bit count SHIFTLEFT128 memAddr, cnt
SRXMMR		x	Shift right xmm register by bit count SRXMMR xmreg, cnt
SHIFTRIGHT128		x	Shift memory location 128bit right by bit count SHIFTRIGHT128 memAddr, cnt

```
;=======================================================
; MOV64 - Moving 64 bit immediate value to a memory location
; usage:  MOV64  num2,102030405060708h
;=======================================================
; MOV128 - Moving 128 bit immediate value to a memory location
; usage:  MOV128  num2,0001020304050607h,08090a0b0c0d0e0fh
;=======================================================
; MOVXMM128 - Loads 128 bit immediate value to any xmm register
; the value has to be reverted EG:
; MOVXMM128 xmm0, 0f0e0d0c0b0a0908h,0706050403020100h
; MOVXMM128 xmm1, "redroeht","desreveR"
; XMM0 = 0F0E0D0C0B0A09080706050403020100
; XMM1 = 726564726F6568746465737265766552
; but when you store register in memory we will get it right
; 0x00007FF7431A52D8  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f  ................
; 0x00007FF7431A52E8  52 65 76 65 72 73 65 64 74 68 65 6f 72 64 65 72  Reversedtheorder
;=======================================================
; SLXMMR - Shift left 128 bit xmm register value
; the value has to be reverted EG: "redroeht","desreveR"
; cnt is limited to low 7 bits, 127 or 7Fh, to avoid overflow
; MOVXMMR128  xmm0, "redroeht","desreveR"
; MOVXMMR128  xmm1, "redroeht","desreveR"
; XMM0 = 726564726F6568746465737265766552
; XMM1 = 726564726F6568746465737265766552
; SLXMMR  xmm0,16
; SLXMMR  xmm1,72;
; XMM0 = 0000726564726F656874646573726576
; XMM1 = 00000000000000000726564726F6568
; 0x00007FF651025258  76 65 72 73 65 64 74 68 65 6f 72 64 65 72 00 00  versedtheorder..
; 0x00007FF651025268  68 65 6f 72 64 65 72 00 00 00 00 00 00 00 00 00  heorder.........
;=======================================================
; SHIFTLEFT128 - Shift left 128 bit memory value
; the value is in normal order EG; 12345678
; .data
; num4 db "The actual order"
; num2 db "The actual order"
;  lea rax,num4            ;pointer to memory in rax
;  SHIFTLEFT128  rax,16
;  lea rax,num2            ;pointer to memory in rax
;  SHIFTLEFT128  rax,72
;before:
;0x00007FF7B4A45000  54 68 65 20 61 63 74 75 61 6c 20 6f 72 64 65 72  The actual order
;0x00007FF7B4A45010  54 68 65 20 61 63 74 75 61 6c 20 6f 72 64 65 72  The actual order
;after:
```

```
;0x00007FF7F6E95000  65 20 61 63 74 75 61 6c 20 6f 72 64 65 72 00 00  e actual order..
;0x00007FF7F6E95010  6c 20 6f 72 64 65 72 00 00 00 00 00 00 00 00 00  l order.........
;=======================================================
; SRXMMR - Shift right 128 bit xmm register value
; the value has to be reversed EG: 87654321
;  MOVXMMR128  xmm0, "redroeht","desreveR"
;  MOVXMMR128  xmm1, "redroeht","desreveR"
;  SRXMMR  xmm0, 16
;  SRXMMR  xmm1, 0x48
; before shift:
; XMM0 = 726564726F6568746465737265766552
; XMM1 = 726564726F6568746465737265766552
; after shift:
; XMM0 = 64726F6568746465737265766552000 0
; XMM1 = 65737265766552000000000000000000
; stored in memory:
; 0x00007FF764DA5268  00 00 52 65 76 65 72 73 65 64 74 68 65 6f 72 64  ..Reversedtheord
; 0x00007FF764DA5278  00 00 00 00 00 00 00 00 00 52 65 76 65 72 73 65  .........Reverse
;=======================================================
; SHIFTRIGHT128 - Shift right 128 bit memory value
; the value is in normal order EG; 12345678
;  lea rax,num4          ;pointer to memory in rax
;  SHIFTRIGHT128  rax,16
;  lea rax,num2          ;pointer to memory in rax
;  SHIFTRIGHT128  rax,72
;=======================================================
```

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

2.13) 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 SSE instruction set.

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

2.14) 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.

2.15) 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.

2.16) 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

2.17) 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
};
```

2.18) 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.

2.19) 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
```

2.20) New DEFINE and UNDEF directives

These two new directives are built-in to UASM and allow something which wasn't previously easy before to check for the existence of a defined item :

```
DEFINE USELITERALS

IF  USELITERALS
  OPTION LITERALS:ON ; Allow string literals use in INVOKE
  printf PROTO :PTR, :PTR
ELSE
  printf PROTO :PTR, :vararg
ENDIF

UNDEF USELITERALS
```

2.21) Improved RECORD directive and operators.

COLOR RECORD blink:1, back:3, intense:1, fore:3

Inline code can use up to 64bit record types directly.

```
mov   cl, COLOR<1, 7, 0, 1>
mov   cobalt.rc, COLOR<1, 7, 0, 1>
```

Additionally, 64bit records can be declared in the data section and manually referenced.

```
COLOR RECORD blink:8, back:8, intense:8, fore:8,blink1:8, back1:8, intense1:8, fore1:8
safir COLOR <224, 225, 226, 14, 224, 225, 226, 14>     ; E0E1E20EE0E1E20E
```

128bit records are also supported:

```
.data

terra MYREC128 <0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,00,1,1,0,0,1,1,0,\
0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,\
0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,\
0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0>
```

2.22) New FRAMEOFS Operator

A new operator, frameofs has been added which will return the relative displacement to the current stack frame for any local or procedure argument and can be used in 32bit and 64bit as follows:

```
main proc c aParam:dword
local   dwWritten:dword
local   hConsole:dword

    mov eax,FRAMEOFS(dwWritten)
    mov eax,[ebp+FRAMEOFS(hConsole)]

    ;for comparison
    mov eax,hConsole
    mov ebx,dwWritten
    mov eax,FRAMEOFS(aParam)
    mov eax,aParam
```

For esp/rsp based procedures the frameofs value can be used directly, however in the case of an ebp/rbp based procedure one has to take into account the setup of the stack frame. In a normal proc this would involve a push {ebp|rbp}. If you have a procedure with no automatic prologue, 4 or 8 will needed to be manually added to account for the lack of stack-frame or ideally one should be manually created as locals or parameters are being referenced.

3. 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.

4. 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).

5. 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.

6. OPTION FLAT

For some types of low-level programming, especially OS kernel and boot-loader code the model adopted by some other non-MASM style assemblers lends itself very well to mixing code of different bit types, for example switching between real and protected or long mode.

To make this type of programming more straight-forward UASM implements a flat mode. A single code section is create with the normal .code simplified directive, however the new directives USE16/USE32/USE64 can be used on their own to switch the current code-generation mode.

For example:

```
option flat:1
.code

    org 0h
    USE16

    xor ax,ax
    xor eax,eax

    org 100h
    var1 dd 10

    USE32

    xor ax,ax
    xor eax,eax

    org 1000h
    var2 dd 20

    USE64
    xor ax,ax
    xor eax,eax
    xor rax,rax
```

7. 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).

8. 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

9. 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