- BEAENGINE_VERSION: 5.3
- DOC_RELEASE: 1.3

1. Disasm function

The Disasm function disassembles one instruction from the Intel ISA. It makes a precise analysis of the focused instruction and sends back a complete structure that is usable to make data-flow and control-flow studies.

Syntax

```
int Disasm(
   PDISASM  &infos
);
```

Parameters

- **&infos**: Pointer to a structure PDISASM

Return

The function may sends you back 3 values. if the analyzed bytes sequence is an invalid opcode, it sends back UNKNOWN_OPCODE (-1). If it tried to read a byte located outside the Security Block, it sends back OUT_OF_BLOCK (-2). In others cases, it sends back the instruction length. Thus, you can use it as a LDE. To have a detailed status, use **infos.Error** field.

2. Disasm infos

This structure is used to store the mnemonic, source and destination operands. You just have to specify the address where the engine has to make the analysis.

```
struct PDISASM {
  UIntPtr EIP;
  UInt64 VirtualAddr;
  UInt32 SecurityBlock;
  char CompleteInstr[INSTRUCT_LENGTH];
  UInt32 Archi;
  UInt64 Options;
  INSTRTYPE Instruction;
  OPTYPE Operand1;
  OPTYPE Operand2;
  OPTYPE Operand3;
  OPTYPE Operand4;
  OPTYPE Operand5;
  OPTYPE Operand6;
  OPTYPE Operand7;
  OPTYPE Operand8;
  OPTYPE Operand9;
  PREFIXINFO Prefix;
  Int32 Error;
  UInt32 Reserved_[48];
};
```

Members

- **EIP**: *[in]* Offset of bytes sequence we want to disassemble
- **VirtualAddr**: *[in]* optional - (For instructions CALL - JMP - conditional JMP - LOOP) By default, this value is 0 (disable). The disassembler calculates the destination address of the branch instruction using VirtualAddr (not EIP). This address can be 64 bits long. This option allows us to decode instructions located anywhere in memory even if they are not at their original place
- **CompleteInstr**: *[out]* String used to store the instruction representation

- **SecurityBlock**: *[in]* By default, this value is 0. (disabled option). In other cases, this number is the number of bytes the engine is allowed to read since EIP. Thus, we can make a read block to avoid some Access violation. On INTEL processors, (in IA-32 or intel 64 modes) , instruction never exceeds 15 bytes. A SecurityBlock value outside this range is useless.
- **Archi**: *[in]* This field is used to specify the architecture used for the decoding. If it is set to 0 or 64 (0x20), the architecture used is 64 bits. If it is set to 32 (0x20), the architecture used is IA-32. If set to 16 (0x10), architecture is 16 bits.
- **Options**: *[in]* This field allows to define some display options. You can specify the syntax: masm, nasm ,goasm. You can specify the number format you want to use: prefixed numbers or suffixed ones. You can even add a tabulation between the mnemonic and the first operand or display the segment registers used by the memory addressing. Constants used are the following :
  - **Tabulation**: add a tabulation between mnemonic and first operand (default has no tabulation)
  - **GoAsmSyntax / NasmSyntax**: change the intel syntax (default is Masm syntax)
  - **PrefixedNumeral**: 200h is written 0x200 (default is suffixed numeral)
  - **ShowSegmentRegs**: show segment registers used (default is hidden)
  - **ShowEVEXMasking**: show opmask and merging/zeroing applyed on first operand for AVX512 instructions (default is hidden)
- **Instruction**: *[out]* Structure **INSTRTYPE**.
- **Operand1**: *[out]* Structure **OPTYPE** that concerns the first operand.
- **Operand2**: *[out]* Structure **OPTYPE** that concerns the second operand.
- **Operand3**: *[out]* Structure **OPTYPE** that concerns the third operand.
- **Operand4**: *[out]* Structure **OPTYPE** that concerns the fourth operand.
- **Operand5**: *[out]* Structure **OPTYPE** that concerns the fifth operand.
- **Operand6**: *[out]* Structure **OPTYPE** that concerns the sixth operand.
- **Operand7**: *[out]* Structure **OPTYPE** that concerns the seventh operand.
- **Operand8**: *[out]* Structure **OPTYPE** that concerns the eighth operand.
- **Operand9**: *[out]* Structure **OPTYPE** that concerns the ninth operand.
- **Prefix**: *[out]* Structure **PREFIXINFO** containing an exhaustive list of used prefixes.
- **Error**: *[out]* This field returns the status of the disassemble process :
  - **Success**: (0) instruction has been recognized by the engine
  - **Out of block**: (-2) instruction length is out of SecurityBlock
  - **Unknown opcode**: (-1) instruction is not recognized by the engine
  - **Exception #UD**: (2) instruction has been decoded properly but sends #UD exception if executed.
  - **Exception #DE**: (3) instruction has been decoded properly but sends #DE exception if executed

3. Instruction infos

this structure gives informations on the analyzed instruction.

```
struct INSTRTYPE {
  Int32 Category;
  Int32 Opcode;
  char Mnemonic[24];
  Int32 BranchType;
  EFLStruct Flags;
  UInt64 AddrValue;
  Int64 Immediat;
  REGISTERTYPE ImplicitModifiedRegs;
  REGISTERTYPE ImplicitUsedRegs;
};
```

Members

- **Category**: *[out]* Specify the family instruction . More precisely, (infos.Instruction.Category & 0xFFFF0000) is used to know if the instruction is a standard one or comes from one of the following technologies: MMX, FPU, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, VMX or SYSTEM. LOWORD(infos.Instruction.Category) is used to know if the instruction is an arithmetic instruction, a logical one, a data transfer one ... To see the complete list of constants used by BeaEngine, go HERE .
- **Opcode**: *[out]* This field contains the opcode on 1, 2 or 3 bytes. If the instruction uses a mandatory prefix, this last one is not present here. For that, you have to use the structure infos.Prefix.
- **Mnemonic**: *[out]* This field sends back the instruction mnemonic with an ASCII format. You must know that all mnemonics are followed by a space (0x20). For example , the instruction “add” is written “add”.
- **BranchType**: *[out]* If the decoded instruction is a branch instruction, this field is set to indicate what kind of jump it is (call, ret, unconditional jump, conditional jump). To get a complete list of constants used by BeaEngine, go HERE

- **Flags**: *[out]* Structure EFLStruct that specifies the used flags.
- **AddrValue**: *[out]* If the decoded instruction is a branch instruction and if the destination address can be calculated, the result is stored in that field. A “jmp eax” or a “jmp [eax]” will set this field to 0 .
- **Immediat**: *[out]* If the instruction uses a constant, this immediat value is stored here.
- **ImplicitModifiedRegs**: *[out]* Some instructions modify registers implicitly. For example, “push 0” modifies the register RSP. In that case, infos.Instruction.ImplicitModifiedRegs.gpr == REG4. Find more useful informations on that field looking at the Structure REGISTERTYPE
- **ImplicitUsedRegs**: *[out]* Some instructions use registers implicitly. Find more useful informations on that field looking at the Structure REGISTERTYPE

4. Operand infos

This structure gives informations about the operand analyzed.

```
struct OPTYPE {
  char OpMnemonic[24];
  UInt64 OpType;
  Int32 OpSize;
  Int32 OpPosition;
  UInt32 AccessMode;
  MEMORYTYPE Memory;
  REGISTERTYPE Registers;
  UInt32 SegmentReg;
} ;
```

Members

- **OpMnemonic**: *[out]* This field sends back, when it is possible, the operand in ASCII format.
- **OpType**: *[out]* This field specifies the operand type. infos.Operandxx.OpType indicates if it is one of the following :
  - REGISTER_TYPE
  - MEMORY_TYPE
  - CONSTANT_TYPE+ABSOLUTE_
  - CONSTANT_TYPE+RELATIVE_
- **OpSize**: *[out]* This field sends back the size of the operand.
- **AccessMode**: *[out]* This field indicates if the operand is modified or not (READ=0x1) or (WRITE=0x2).
- **Memory**: *[out]* Structure MEMORYTYPE , filled only if infos.Operandxx.OpType == MEMORY_TYPE.
- **Registers**: *[out]* Structure REGISTERTYPE , filled only if infos.Operandxx.OpType == REGISTER_TYPE.
- **SegmentReg**: *[out]* This field indicates, in the case of memory addressing mode, the segment register used :
  - ESReg
  - DSReg
  - FSReg
  - GSReg
  - CSReg
  - SSReg

5. Prefixes infos

This structure gives informations on used prefixes. When can know if some prefixes are used properly or not.

```
struct PREFIXINFO {
  int Number;
  int NbUndefined;
  UInt8 LockPrefix;
  UInt8 OperandSize;
  UInt8 AddressSize;
  UInt8 RepnePrefix;
  UInt8 RepPrefix;
  UInt8 FSPrefix;
  UInt8 SSPrefix;
  UInt8 GSPrefix;
  UInt8 ESPrefix;
  UInt8 CSPrefix;
```

```
  UInt8 DSPrefix;
  UInt8 BranchTaken;
  UInt8 BranchNotTaken;
  REX_Struct REX;
  char alignment[2];
};
```

Membres

- **Number**: *[out]* Indicates the number of prefixes used.
- **NbUndefined**: *[out]* Indicates the number of prefixes used in a wrong way (illegal use).
- **LockPrefix**: *[out]* Concerns the LOCK prefix. It can takes one of the following values :
  - NotUsedPrefix = 0
  - InUsePrefix = 1
  - SuperfluousPrefix = 2
  - InvalidPrefix = 4
  - MandatoryPrefix = 8
- **OperandSize**: *[out]* Concerns the prefix used to define the size of operands.
- **AddressSize**: *[out]* Concerns the prefix used to define the AddressSize
- **RepnePrefix**: *[out]* Concerns the prefix used to define the REPNE.
- **RepPrefix**: *[out]* Concerns the prefix used to define the REP.
- **FSPrefix**: *[out]* Concerns the prefix used to define the FS segment .
- **SSPrefix**: *[out]* Concerns the prefix used to define the SS segment .
- **GSPrefix**: *[out]* Concerns the prefix used to define the GS segment .
- **ESPrefix**: *[out]* Concerns the prefix used to define the ES segment .
- **CSPrefix**: *[out]* Concerns the prefix used to define the CS segment .
- **DSPrefix**: *[out]* Concerns the prefix used to define the DS segment .
- **BranchTaken**: *[out]* Concerns branch hint prefix 0x3E (taken).
- **BranchNotTaken**: *[out]* Concerns branch hint prefix 0x2E (not taken).
- **REX**: *[out]* Concerns the prefix used to define the REX in 64 bits mode. The structure sended back is :

```
struct REX_Struct {
  BYTE W_;
  BYTE R_;
  BYTE X_;
  BYTE B_;
  BYTE state;
};
```

Fields W_, R_, X_, B_ are set to 1 if the field is used. The field state is set to *InUsePrefix* if a REX prefix is used.

6. EFLAGS infos

This structure gives informations on the register EFLAGS.

```
struct EFLStruct {
  BYTE OF_;
  BYTE SF_;
  BYTE ZF_;
  BYTE AF_;
  BYTE PF_;
  BYTE CF_;
  BYTE TF_;
  BYTE IF_;
  BYTE DF_;
  BYTE NT_;
  BYTE RF_;
  BYTE alignment;
};
```

Members

Except for the field “alignment” that is only present for alignment purpose, all fields can be filled with one of the following values :

- TE_ ; the flag is tested
- MO_ ; the flag is modified
- RE_ ; the flag is reset
- SE_ ; the flag is set
- UN_ ; undefined behavior
- PR_ ; restore prior state

7. Memory infos

This structure gives informations if `infos.Operandxx.OpType == MEMORY_TYPE`.

```
struct MEMORYTYPE {
  Int64 BaseRegister;
  Int64 IndexRegister;
  Int32 Scale;
  Int64 Displacement;
};
```

Members

- **BaseRegister**: *[out]* Indicate the base register in the formula: [BaseRegister + IndexRegister*Scale + Displacement].
- **IndexRegister**: *[out]* Indicate the index register in the formula: [BaseRegister + IndexRegister*Scale + Displacement].
- **Scale**: *[out]* Indicate the scale: 1, 2, 4 ou 8.
- **Displacement**: *[out]* Value of the displacement in the formula: [BaseRegister + IndexRegister*Scale + Displacement].

8. Registers infos

This structure gives informations on operands if:

- `infos.Operandxx.OpType` == REGISTER_TYPE
- or if `infos.Instruction.ImplicitModifiedRegs` is filled

```
struct REGISTERTYPE{
   Int64 type;
   Int64 gpr;
   Int64 mmx;
   Int64 xmm;
   Int64 ymm;
   Int64 zmm;
   Int64 special;
   Int64 cr;
   Int64 dr;
   Int64 mem_management;
   Int64 mpx;
   Int64 opmask;
   Int64 segment;
   Int64 fpu;
   Int64 tmm;
};
```

Members

- **type**: *[out]* set of flags to define which type of registers are used. For instance, to test if operand1 is a general purpose register, use `infos.Operand1.Registers.type & GENERAL_REG`.
- **gpr**: *[out]* set of flags to define which general purpose register is used. For instance, to test if operand 1 uses RAX, test `infos.Operand1.Registers.gpr & REG0`
- **mmx**: *[out]* set of flags to define which MMX register is used. For instance, to test if operand 1 uses MM0, test `infos.Operand1.Registers.mmx & REG0`
- **xmm**: *[out]* set of flags to define which XMM register is used. For instance, to test if operand 1 uses XMM0, test `infos.Operand1.Registers.xmm & REG0`
- **ymm**: *[out]* set of flags to define which YMM register is used. For instance, to test if operand 1 uses YMM0, test `infos.Operand1.Registers.ymm & REG0`

- **zmm**: *[out]* set of flags to define which ZMM register is used. For instance, to test if operand 1 uses ZMM0, test `infos.Operand1.Registers.zmm & REG0`.
- **special**: *[out]* set of flags to define which special register is used. Special Registers are following :
  - RFLAGS/EFLAGS (REG0)
  - MXCSR (REG1)
  - SSP (REG2)
  - PKRU (REG3)
  - UIF (REG4) User Interrupt Flag (1 bit in the user interrupt state) is not a MSR but actually, no MSR is used to set/read this flag
  - MSR IA32_TIME_STAMP_COUNTER (REG5)
  - MSR IA32_TSC_AUX (REG6)
- **cr**: *[out]* set of flags to define which CR register is used. For instance, to test if operand 1 uses CR0, test `infos.Operand1.Registers.cr & REG0`.
- **dr**: *[out]* set of flags to define which DR register is used. For instance, to test if operand 1 uses DR0, test `infos.Operand1.Registers.dr & REG0`.
- **mem_management**: *[out]* set of flags to define which memory management register is used.
  - GDTR (REG0)
  - LDTR (REG1)
  - IDTR (REG2)
  - TR (REG3)
- **mpx**: *[out]* set of flags to define which bound register is used. For instance, to test if operand 1 uses *BND0*, test `infos.Operand1.Registers.mpx & REG0`.
- **opmask**: *[out]* set of flags to define which opmask register is used. For instance, to test if operand 1 uses *k0*, test `infos.Operand1.Registers.opmask & REG0`.
- **segment**: *[out]* set of flags to define which segment register is used.
  - ES (REG0)
  - CS (REG1)
  - SS (REG2)
  - DS (REG3)
  - FS (REG4)
  - GS (REG5)
- **fpu**: *[out]* set of flags to define which FPU register is used. For instance, to test if operand 1 uses *st(0)*, test `infos.Operand1.Registers.fpu & REG0`.
- **tmm**: *[out]* set of flags to define which TMM register is used (intel AMX extension). For instance, to test if operand 1 uses *tmm0*, test `infos.Operand1.Registers.tmm & REG0`.

9. Constants

Here is an exhaustive list of constants used by fields sends back by BeaEngine.

Values taken by (infos.Instruction.Category & 0xFFFF0000)

```
GENERAL_PURPOSE_INSTRUCTION   =          0x10000,
FPU_INSTRUCTION               =          0x20000,
MMX_INSTRUCTION               =          0x30000,
SSE_INSTRUCTION               =          0x40000,
SSE2_INSTRUCTION              =          0x50000,
SSE3_INSTRUCTION              =          0x60000,
SSSE3_INSTRUCTION             =          0x70000,
SSE41_INSTRUCTION             =          0x80000,
SSE42_INSTRUCTION             =          0x90000,
SYSTEM_INSTRUCTION            =          0xa0000,
VM_INSTRUCTION                =          0xb0000,
UNDOCUMENTED_INSTRUCTION      =          0xc0000,
AMD_INSTRUCTION               =          0xd0000,
ILLEGAL_INSTRUCTION           =          0xe0000,
AES_INSTRUCTION               =          0xf0000,
CLMUL_INSTRUCTION             =         0x100000,
AVX_INSTRUCTION               =         0x110000,
AVX2_INSTRUCTION              =         0x120000,
MPX_INSTRUCTION               =         0x130000,
AVX512_INSTRUCTION            =         0x140000,
```

```
SHA_INSTRUCTION                =          0x150000,
BMI2_INSTRUCTION               =          0x160000,
CET_INSTRUCTION                =          0x170000,
BMI1_INSTRUCTION               =          0x180000,
XSAVEOPT_INSTRUCTION           =          0x190000,
FSGSBASE_INSTRUCTION           =          0x1a0000,
CLWB_INSTRUCTION               =          0x1b0000,
CLFLUSHOPT_INSTRUCTION         =          0x1c0000,
FXSR_INSTRUCTION               =          0x1d0000,
XSAVE_INSTRUCTION              =          0x1e0000,
SGX_INSTRUCTION                =          0x1f0000,
PCONFIG_INSTRUCTION            =          0x200000,
UINTR_INSTRUCTION              =          0x210000,
KL_INSTRUCTION                 =          0x220000,
AMX_INSTRUCTION                =          0x230000,
```

Values taken by LOWORD(infos.Instruction.Category)

```
DATA_TRANSFER = 0x1
ARITHMETIC_INSTRUCTION = 2
LOGICAL_INSTRUCTION = 3
SHIFT_ROTATE = 4
BIT_BYTE = 5
CONTROL_TRANSFER = 6
STRING_INSTRUCTION = 7
InOutINSTRUCTION = 8
ENTER_LEAVE_INSTRUCTION = 9
FLAG_CONTROL_INSTRUCTION = 10
SEGMENT_REGISTER = 11
MISCELLANEOUS_INSTRUCTION = 12
COMPARISON_INSTRUCTION = 13
LOGARITHMIC_INSTRUCTION = 14
TRIGONOMETRIC_INSTRUCTION = 15
UNSUPPORTED_INSTRUCTION = 16
LOAD_CONSTANTS = 17
FPUCONTROL = 18
STATE_MANAGEMENT = 19
CONVERSION_INSTRUCTION = 20
SHUFFLE_UNPACK = 21
PACKED_SINGLE_PRECISION = 22
SIMD128bits = 23
SIMD64bits = 24
CACHEABILITY_CONTROL = 25
FP_INTEGER_CONVERSION = 26
SPECIALIZED_128bits = 27
SIMD_FP_PACKED = 28
SIMD_FP_HORIZONTAL = 29
AGENT_SYNCHRONISATION = 30
PACKED_ALIGN_RIGHT = 31
PACKED_SIGN = 32
PACKED_BLENDING_INSTRUCTION = 33
PACKED_TEST = 34
PACKED_MINMAX = 35
HORIZONTAL_SEARCH = 36
PACKED_EQUALITY = 37
STREAMING_LOAD = 38
INSERTION_EXTRACTION = 39
DOT_PRODUCT = 40
SAD_INSTRUCTION = 41
ACCELERATOR_INSTRUCTION = 42
ROUND_INSTRUCTION = 43
```

Values taken by infos.Instruction.BranchType

```
  JO = 1,
  JC = 2,
  JE = 3,
  JA = 4,
  JS = 5,
  JP = 6,
  JL = 7,
  JG = 8,
  JB = 9,
  JECXZ = 10,
  JmpType = 11,
  CallType = 12,
  RetType = 13,
  JNO = -1,
  JNC = -2,
  JNE = -3,
  JNA = -4,
  JNS = -5,
  JNP = -6,
  JNL = -7,
  JNG = -8,
  JNB = -9
```

Values taken by infos.Operandxx.OpType

```
NO_ARGUMENT =                0x10000,
REGISTER_TYPE =              0x20000,
MEMORY_TYPE =                0x30000,
CONSTANT_TYPE + RELATIVE_ = 0x4040000,
CONSTANT_TYPE + ABSOLUTE_ = 0x8040000
```

Values taken by infos.Options

```
  NoTabulation = 0x0,
  Tabulation = 0x1,

  MasmSyntax = 0x000,
  GoAsmSyntax = 0x100,
  NasmSyntax = 0x200,

  PrefixedNumeral = 0x10000,
  SuffixedNumeral = 0x00000,

  ShowSegmentRegs = 0x01000000,
  ShowEVEXMasking = 0x02000000,
```

Values taken by infos.Operandxx.SegmentReg

```
    ESReg 0x1
    DSReg 0x2
    FSReg 0x4
    GSReg 0x8
    CSReg 0x10
    SSReg 0x20
```

Values taken by infos.Instruction.Flags.OF_ , .SF_ ...

```
TE_      = 1   ; test
MO_      = 2   ; modify
RE_      = 4   ; reset
SE_      = 8   ; set
UN_      = 10h   ; undefined
PR_      = 20h   ; restore prior value
```