Writing disassembler - part 1 - v1.0

0.0 Introdution

0.1 What?

Disassembler engine it's some procedure that take some pointer to assembled code (for example it takes it from some exe file from .text (.code) section. Then it disassembles it to some user-friendly structures. Normally assembled instructions have different length and it's hard (or impossible) to manipulate them without disassembling.

0.2 Why?

Disassembling is used for poli/meta-morphic viruses. For example metamorphic virus will disassemble his own body (even disassembler procedure) then shrink/change/expand instructions using disassembled structures and then it will reassemble it again and ofcourse put it to some exe it wants to infect :).

0.3 And how?

I will write the engine in assembler - because we probably want to use it in some viri stuff :). I use masm - many people say that masm is crap and they use nasm. I realy don't know which assembler is better. I started to learn assembler with masm and I realy like it as it has very powerful macro engine (which we will use writing our engine). Nasm have macro support too - I even tried nasm few times but, well, I prefer masm.

1.0 Overview of disassembler engine

I don't know how to write a good disassembler (dasm) engine yet :). But I have few sources and few ideas (more sources than ideas :) and I will try. The most important part of making such engine is planing - without planing we won't finish anything! So first few chapters of this "tutorial" will focus just on theoretical aspects of our dasm structure.

First of all we have to plan how we will store instructions disassembled by our engine - we need some structure which will be capable to hold any instruction we want. It must be also very comfortable to manipulate it. When we will make it, it will define our own pseudo-assembler language. So lets begin and jump to next chapter.

2.0 Structure of instruction

Okay, we need instruction structure - lets create it in "_instr.inc". Here is body of our instruction (empty now):

```
; _instr.inc

_instr struct
   ...
_instr ends
```

Most important field in our structure will be opcode field. It will define the instruction that our structure holds. Let it be one byte - it will allow us to hold there 2^8 = 256 different values (instructions). It should be enough as we don't need every instruction in processor to be recognizable by our dasm engine (for example we don't need FPU or MMX instructions, probably only the basic ones). Now our structure will look like this:

```
; _instr.inc

_instr struct
   opcode byte ?
   ...
_instr ends
```

2.1 Opcodes

Now we have to define some opcodes that our engine will use - lets create file "opcodes.inc" (writing code in separate files will allow us to manage project easier). We can decide for each instruction what number (from 0 to 255) it will have. But at first lets construct some list of x86 instructions that we want to have in engine:

name	**operand1**	**operand2**
arithmetic instructions		

```
add   ;  add  | mem/reg     | mem/reg/imm
sub   ;  sub  | mem/reg     | mem/reg/imm
inc   ;  inc  | mem/reg      |
dec   ;  dec  | mem/reg      |
neg   ;  neg  | mem/reg      |
mul   ;  mul  | mem/reg     | eax/edx
div   ;  div  | mem/reg     | eax/edx
----------------------------------------
logic instructions
----------------------------------------
or    ; or    | mem/reg      | mem/reg/imm
and   ; and   | mem/reg      | mem/reg/imm
xor   ; and   | mem/reg      | mem/reg/imm
not   ; not   | mem/reg      |
----------------------------------------
shift instructions
----------------------------------------
shl   ; shl   | mem/reg     | imm/cl
shr   ; shr   | mem/reg     | imm/cl
sal   ; sal   | mem/reg     | imm/cl
sar   ; sar   | mem/reg     | imm/cl
----------------------------------------
rotation instructions
----------------------------------------
rol   ; rol   | mem/reg     | imm/cl
ror   ; ror   | mem/reg     | imm/cl
rcl   ; rcl   | mem/reg     | imm/cl
rcr   ; rcr   | mem/reg     | imm/cl
----------------------------------------
data transfer instructions
----------------------------------------
mov   ; mov   | mem/reg     | mem/reg/imm
xchg  ; xchg  | mem/reg      | mem/reg
push  ; push  | mem/reg/imm |
pop   ; pop   | mem/reg      |
pusha; pusha |               |
popa  ; popa  |              |
pushad;pushad|               |
popad; popad |               |
pushf; pushf |               |
pushfd;pushfd|               |
popf  ; popf  |              |
popfd; popfd |               |
stc   ; stc   |             |
clc   ; stc   |             |
cmc   ; stc   |             |
std   ; stc   |             |
cld   ; stc   |             |
sti   ; stc   |              |
cli   ; stc   |             |
cbw   ; stc   |             |
cwd   ; stc   |              |
cwde  ; stc   |              |
----------------------------------------
other instructions
----------------------------------------
```

lea ; lea	reg	mem
nop ; nop		
program control instructions		
jxx ; jxx	mem/reg/imm	
jmp ; jmp	mem/reg/imm	
enter; enter	imm	imm
leave;leave		
call; call	mem/reg/imm	
ret ; ret	imm	
loopxx;loopxx	imm	
string instructions		
cmps; cmps	**esi**	**edi**
lods; lods	**esi**	**eax**
movs; movs	**esi**	**edi**
scas; scas	**edi**	**eax**
stos; stos	**edi**	**eax**
compare in structions		
cmp ; cmp	mem/reg	mem/reg/imm
test; test	mem/reg	mem/reg/imm
virtual instructions		
movm; movm	mem	mem
apistart;	imm	
apiend;	imm	imm

Okay, this table needs some explanation. It includes all instructions that we need in our engine - ofcourse we can put here any instruction but we probably won't use most of them. Virtual instructions - what is it? In assembler for example there is no instruction mov mem,mem - it's forbidden. But we have to do this very often in our program. We do this by for example push mem/pop mem or mov reg,mem/mov mem,reg and so on. But in our pseudo assembler we can hold thos instructions like one instruction movm (move mem to mem). It will help us to manipulate the code later. We will just have to expand such instruction in 2 instructions during assembly process. Instructions apistart/apiend are just prologue of the procedure (push ebp/mov ebp,esp/sub esp,imm) and epilogue (add esp,imm/pop ebp).

I you don't know any instructions from the table just type "intel instruction set" in google and check first few links to get instructions and descriptions of them.

There is something important about operands - even if operand1 can be mem and operand can be mem too - don't forget that mem,mem is forbidden!

Okay what about operands size? Ofcourse the register operand can be 8 or 16 or 32 bit. And immidiate value can also be 8/16/32bit. But our pseudo assembler must be as comfortable for us as possible so we will hold information about operands sizes later in the structure.

Now when we have list of instruction lets construct "opcodes.inc", where we will declare some opcodes constants.

```
; opcodes.inc

.const

       ; arithmetic instructions
       OPCODE_ADD equ 000h
       OPCODE_SUB equ 001h
       OPCODE_INC equ 002h
       OPCODE_DEC equ 003h
       OPCODE_NEG equ 004h
       OPCODE_MUL equ 005h
       OPCODE_DIV equ 006h

       ; logic instructions
       OPCODE_OR  equ 007h
       OPCODE_AND equ 008h
       OPCODE_XOR equ 009h
       OPCODE_NOT equ 00ah

       ; shift instructions
       OPCODE_SHL equ 00bh
       OPCODE_SHR equ 00ch
       OPCODE_SAL equ 00dh
       OPCODE_SAR equ 00eh

       ; rotation instructions
       OPCODE_ROL equ 00fh
       OPCODE_ROR equ 010h
       OPCODE_RCL equ 011h
       OPCODE_RCR equ 012h

       ; data transfer instructions
       OPCODE_MOV    equ 013h
       OPCODE_XCHG   equ 014h
       OPCODE_PUSH   equ 015h
       OPCODE_POP    equ 016h
       OPCODE_PUSHA  equ 017h
       OPCODE_POPA   equ 018h
       OPCODE_PUSHAD equ 019h
       OPCODE_POPAD  equ 01ah
       OPCODE_PUSHF  equ 01bh
       OPCODE_PUSHFD equ 01ch
       OPCODE_POPF   equ 01dh
       OPCODE_POPFD  equ 01eh
       OPCODE_STC    equ 01fh
       OPCODE_CLC    equ 020h
       OPCODE_CMC    equ 021h
       OPCODE_STD    equ 022h
```

```
OPCODE_CLD    equ 023h
OPCODE_STI    equ 024h
OPCODE_CLI    equ 025h
OPCODE_CBW    equ 026h
OPCODE_CWD    equ 027h
OPCODE_CWDE   equ 028h

; other instructions
OPCODE_LEA equ 02ah
OPCODE_NOP equ 090h

; program control instructions
OPCODE_JXX    equ 02ch
OPCODE_JMP    equ 02dh
OPCODE_ENTER  equ 02eh
OPCODE_LEAVE  equ 02fh
OPCODE_CALL   equ 030h
OPCODE_RET    equ 031h
OPCODE_LOOPXX equ 032h

; string instructions
OPCODE_CMPS equ 033h
OPCODE_LODS equ 034h
OPCODE_MOVS equ 035h
OPCODE_SCAS equ 036h
OPCODE_STOS equ 037h

; compare in structions
OPCODE_CMP  equ 03eh
OPCODE_TEST equ 03fh

; virtual instructions
OPCODE_MOVM     equ 040h
OPCODE_APISTART equ 041h
OPCODE_APIEND   equ 042h
```

2.2 Operands

Now we need some variable in our _instr structure that will represent operands used by the instruction which is definied by the opcode field. First lets see what we will add in file "_instr.inc":

```
; _instr.inc

include opcodes.inc
include operands.inc

_instr struct
   opcode   byte ?
   operands byte ?
   ...
_instr ends
```

All we need is file operands.inc. But what types do we have in assembler? There are 3 types of operands: reg (register), mem (memory address), imm (immediate value - const number). Each isntruction can have 0,1 or 2 operands (well in fact there are instructions that take 3 arguments but we don't need them). So:

```
; _operands.inc

.const

   OPERANDS_NONE    equ 000h
   OPERANDS_REG     equ 001h
   OPERANDS_MEM     equ 002h
   OPERANDS_IMM     equ 003h
   OPERANDS_REG_REG equ 004h
   OPERANDS_REG_MEM equ 005h
   OPERANDS_REG_IMM equ 006h
   OPERANDS_MEM_MEM equ 007h
   OPERANDS_MEM_REG equ 008h
   OPERANDS_MEM_IMM equ 009h
```

2.3 Prefixes

Prefixes are some bytes that we can put in front of instruction. Instruction may have 0,1 or more prefixes. Not evry instruction can have specific prefix. There are few groups of prefixes:

Lock and repeat prefixes (3 values):

LOCK - 0f0h
REPNE/REPNZ - 0f2h
REP - 0f3h
REPE/REPZ - 0f3h (same as REP)
SIMD - 0f3h (same as REP)

Segment override prefixes (6 values):

CS - 02eh
SS - 036h
DS - 03eh
ES - 026h
FS - 064h
GS - 065h

Operand-size override prefix (1 value):

OP_SIZE - 066h

Address-size override prefix (1 value):

ADDR_SIZE - 067h

Each instruction can have only 1 prefix from each group - so one instruction can have up to 4 prefixes (we have 4 groups). There are few prefixes we wont use - SIMD and LOCK - we just don't need them. We will store all prefix data in one byte called prefixes:

```
; _instr.inc

include opcodes.inc
include operands.inc
include prefixes.inc

_instr struct
   opcode   byte ?
   operands byte ?
   prefixes byte ?
   ...
_instr ends
```

We have one byte so 8 bits to store those values. Lets do like that:

```
[7] [6] [5] [4] [3] [2] [1] [0]
      \  |  /     |   |   |   |_ operand size (bit 0)
       \ | /      |   |   |_ address size (bit 1)
        \|/       |   |_ REPNE/REPNZ (bit 2)
         |        |_ REP/REPE/REPZ (bit 3)
         |_ segment override (bit 4,5,6)
```

7th bit stays unused for now maybe we will use it later for some extra data. Okay lets look into "prefixes.inc":

```
; prefixes.inc

.const

; bit patterns
PREFIX_OP_SIZE   equ 01h ; bit 0
PREFIX_ADDR_SIZE equ 02h ; bit 1
PREFIX_REPNE     equ 04h ; bit 2
PREFIX_REPNZ     equ PREFIX_REPNE
PREFIX_REPE      equ 08h ; bit 3
PREFIX_REP       equ PREFIX_REPE
PREFIX_REPZ      equ PREFIX_REPE
PREFIX_SEG_NONE  equ 0 000 0000b
PREFIX_CS        equ 0 001 0000b
PREFIX_SS        equ 0 010 0000b
PREFIX_DS        equ 0 011 0000b
PREFIX_ES        equ 0 100 0000b
PREFIX_FS        equ 0 101 0000b
PREFIX_GS        equ 0 110 0000b

; bit indexes (only for 1 bit prefixes)
BI_OP_SIZE   equ 00h
BI_ADDR_SIZE equ 01h
BI_REPNE     equ 02h
BI_REPNZ     equ BI_REPNE
BI_REPE      equ 03h
BI_REP       equ BI_REPE
BI_REPZ      equ BI_REPE

; prefix real opcodes
```

```
OPCODE_OP_SIZE   equ 066h
OPCODE_ADDR_SIZE equ 067h
OPCODE_REPNE     equ 0f2h
OPCODE_REPNZ     equ OPCODE_REPNE
OPCODE_REPE      equ 0f3h
OPCODE_REP       equ OPCODE_REPE
OPCODE_REPZ      equ OPCODE_REPE
OPCODE_CS        equ 02eh
OPCODE_SS        equ 036h
OPCODE_DS        equ 03eh
OPCODE_ES        equ 026h
OPCODE_FS        equ 064h
OPCODE_GS        equ 065h
```

2.4 Instruction data

Now as we have defined our instruction by its opcode, operands and prefixes, we have to create some structure which will hold data for instruction - for example which register it uses, memory address, any immediate data and so on. The use of this structure will depend on which instruction it is and what operands it uses and what prefixes it has (operand and address prefixes especially). Lets look on this structure ("_idata.inc"):

```
; _idata.inc

include registers.inc
include _mem.inc

_idata struct
   union
      reg1 byte ?
       union
         imm1_8  byte  ?
         imm1_16 word  ?
         imm1_32 dword ?
      ends
      mem1 _mem <>
   ends
   union
      reg2 byte ?
      union
         imm2_8  byte  ?
         imm2_16 word  ?
         imm2_32 dword ?
      ends
      mem2 _mem <>
   ends
ends
```

Okay, we have 2 unions inside - we can use each union as register/immediate/memory. It allow us to construct any option from our defined OPERANDS_XXX constants. The reg1 and reg2 fields will be used ofcourse to encode registers. We need some constants ("registers.inc"):

```
; registers.inc

.const

   REG_EAX equ 00h
   REG_EBX equ 03h
   REG_ECX equ 01h
   REG_EDX equ 02h
   REG_ESI equ 06h
   REG_EDI equ 07h
   REG_EBP equ 05h
   REG_ESP equ 04h
```

2.4.1 Memory address structure

We have few addressing modes on x86 processors. For example we have direct address [0x00112233] or by register [eax] and so on. The most complex will be addressing mode like this [reg1+reg2 *multiply+displacement] (we will focus on specific addressing modes in later chapters). Multiply can be 1/2/4/8. It's 4 values so we need only 2 bits to encode it. Then we have displacement - it can be 1/2/4 byte long so we need 4 bytes to handle this. So our _mem struct will be like this ("_mem.inc"):

```
; _mem.inc

include registers.inc

.const
   ; multiplication values
   MULTI_1 equ 00 000000b
   MULTI_2 equ 01 000000b
   MULTI_4 equ 10 000000b
   MULTI_8 equ 11 000000b

   MULTI_BITMASK equ 11 000000b

   ; addressing modes
   MODE_DISP         equ 100 00000b
   MODE_REG          equ 011 00000b
   MODE_REG_REG      equ 010 00000b
   MODE_REG_DISP     equ 001 00000b
   MODE_REG_REG_DISP equ 000 00000b

   MODE_BITMASK equ 111 00000b


_mem struct
   memreg1 byte  ? ; bits 5/6/7 = mode
   memreg2 byte  ? ; bits 6/7 = multiplicator
   union
      disp8  byte  ?
```

```
            disp16 word  ?
            disp32 dword ?
         ends
      _mem ends
```

So in memreg1 byte, bits number 0/1/2 contain info about register and bits 5/6/7 about the addressing mode.
In memreg2 byte bits 0/1/2 encode the second register and bits 6/7 define multiplicator. Disp is just displacement which can be 1/2/4 byte long.

2.5 Pointer to code

The next member of _instr structure will be the pointer to the code - it will just point to the real code that we are disassembling - it is very important but I will explain it later.
So this pointer will be just a dword and we will call it "pointer" - look into **2.6** for the whole _instr structure definition.

2.6 "Label mark"

Label mark is very handy thing which was "invented" by Mental Driller (I think so) in his metamorphic virus called "metamorpho" :). It's not really neccessery during disassembling but it will be important during morphing of the code. Label mark it's just a byte that can be 0 or 1. It's 1 if any jump/call point to this instruction (instruction has a label on it) or 0 if not. The whole _instr structure now looks:

```
; _instr.inc

 include opcodes.inc
 include operands.inc
 include prefixes.inc
 include _idata.inc

 _instr struct
    opcode    byte ?
    operands  byte ?
    prefixes  byte ?
    _idata    idata <>
    pointer   dword ?
    labelmark byte  ?
 _instr ends
```

3.0 Files organization

Okay we created some files - now lets make a clear view of how we set up directories for our engine. First of all we need some root directory - lets call it "dasm_engine". Inside we should have 2 folders "source" and "include" or "src" and "inc" as you wish (I choose src/inc). We will put all *.inc files into the "inc" directory and all *.asm files into "src" directory. So till now we have:

> dasm_engine
> src
> inc
> _instr.inc
> opcodes.inc
> operands.inc
> prefixes.inc
> registers.inc
> _mem.inc
> _idata.inc

U can download those files from the link: http://rapidshare.com/files/292167997/dasm_engine.rar.html .

4.0 What in next part?

In next part we will discuss about the whole engine routine - how it will work, what parameters it will take and so on. Then we will write few (or many :) useful macros. And we will start to write the disassembling procedure :). If you have any questions or remarks or anything else just email me: alek.barszczewski@gmail.com.