mcc, a machine code compiler

October 9, 2019

github.com/esote/mcc

Overview

ELF

ELF headers consist of three parts:

1. file header (also called ELF header)
   - execution environment (architecture, OS ABI)
   - header sizes and offsets (program entry point "`_start`", count of program headers)
2. program header
   - layout during execution (memory allocated, permission flags)
   - where you can specify self-modifying executables
3. section header
   - relocation data
   - things useful when disassembling
   - section header string table (shstrtab) gives names to sections of the executable ("`.bss`", "`.text`", etc)

What

mcc parses files of binary or hexadecimal text to produce i386 and x86-64 ELF executables.

Why?

Is assembly a high-level programming language?

```
include \masm32\include\masm32rt.inc    ; use the Masm32 library

.code
demomain:
  REPEAT 20
    switch rv(nrandom, 9)   ; generate a number between 0 and 8
    mov ecx, 7
    case 0
        print "case 0"
    case ecx                ; in contrast to most other programming languages,
        print "case 7"      ; the Masm32 switch allows "variable cases"
    case 1 .. 3
        .if eax==1
            print "case 1"
        .elseif eax==2
            print "case 2"
        .else
            print "cases 1 to 3: other"
        .endif
    case 4, 6, 8
        print "cases 4, 6 or 8"
    default
        mov ebx, 19          ; print 20 stars
        .Repeat
            print "*"
            dec ebx
        .Until Sign?         ; loop until the sign flag is set
    endsw
    print chr$(13, 10)
  ENDM
  exit
end demomain
```

Figure: MASM code, Wikipedia “Assembly Language”

Why?

- Assemblers (and linkers) optimize by default
- Familiar constructs are really macros or "assembler directives"
  - `global _start` is the user-level directive macro of the primitive directive `[global _start]`
  - `section .text` is a convoluted macro for `[section .text]`
  - Labels are really just memory addresses
  - "`.bss`" just defines memory addresses which are writable, and "`.text`" executable[1]

No, when compared with languages like C, Lisp, or Scratch, assembly is nowhere near high-level.

[1] see also: self-modifying code

Why?

Assembly cannot be mapped one-to-one with machine code

OPCODE MAP

Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number *

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
80-83	1	mem, 11B		ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
8F	1A	mem, 11B		POP							
C0,C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL	2	mem, 11B		ROL	ROR	RCL	RCR	SHL/SAL	SHR		SAR
F6, F7	3	mem, 11B		TEST Ib/Iz		NOT	NEG	MUL AL/rAX	IMUL AL/rAX	DIV AL/rAX	IDIV AL/rAX
FE	4	mem, 11B		INC Eb	DEC Eb						
FF	5	mem, 11B		INC Ev	DEC Ev	near CALL[f64] Ev	far CALL Ep	near JMP[f64] Ev	far JMP Mp	PUSH[d64] Ev	
0F 00	6	mem, 11B		SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
0F 01	7	mem		SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb
		11B		VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100)	MONITOR (000) MWAIT (001) CLAC (010) STAC (011) ENCLS (111)	XGETBV (000) XSETBV (001) VMFUNC (100) XEND (101) XTEST (110) ENCLU(111)					SWAPGS[o64] (000) RDTSCP (001)
0F BA	8	mem, 11B						BT	BTS	BTR	BTC
0F C7	9	mem			CMPXCH8B Mq CMPXCHG16B Mdq					VMPTRLD Mq	VMPTRST Mq
			66							VMCLEAR Mq	
			F3							VMXON Mq	
		11B								RDRAND Rv	RDSEED Rv
			F3								RDPID Rd/q

Figure: Intel SDM Vol 2D page A-18

Why?

github.com/xoreaxeaxeax/sandsifter

Tool for finding undocumented x86 instructions. (Also used to find bugs in CPU hardware or hypervisors.)

On an Intel i5-8250U running Xen 4.8.5-7 through a PVH DomU VM: **89,919 undocumented instructions**.

Very few tools allow you to easily execute machine code directly from their binary representations.

To-Do

What's there left to do?

1. Fix inconsistent endianness and support for writing big-endian executables.
2. Support `.data` segments
    1. Support for specification of arbitrary segments
    2. Work around kernel randomization (position-independent executables)

Items 2.1 and 2.2 are required for execution in OpenBSD.

Tools

- `readelf(1)`
  1. `-eW (--headers --wide)`
  2. `-a (--all)`
- `objdump(1)`
  1. `-d (--disassemble)`
  2. `-D (--disassemble-all)`
  3. `-DFsxwz -M intel (--disassemble-all --file-offsets --full-contents --all-headers --wide --disassemble-zeroes -M intel)`
- `xxd(1)`
  1. `-b (-bits)`
- `strings(1)`

Resources

Intel 64 and IA-32 Architectures Software Developer's Manual (volume 2 chapter 3, appendices A and B) [2]

AMD64 Architecture Programmer's Manual (volume 3 chapter 3)

System V Application Binary Interface

System V ABI AMD64 Architecture Processor Supplement

`elf(5)` (online manuals: 1, 2)

A Whirlwind Tutorial on Creating Teensy ELF Executables for Linux

Black Hat 2017 "Breaking the x86 Instruction Set" (author of sandsifter)

[2] specifically the opcode reference, ModR/M, and SIB byte tables