Programming Project 1
695.744

Audrey Long
Johns Hopkins University

Programming Assignment 1

695.744
T. McGuire
W. Elam
Johns Hopkins University

1. Briefly (~1 page) discuss the strengths and weaknesses of the recursive descent and linear sweep algorithms. What makes IDA a powerful disassembler?

2. Throughout the semester, you will be utilizing tools such as **IDA Pro**, **objdump**, **Windbg, gdb** and other debugging/disassembling utilities. One feature these tools share is the ability to turn *machine code* into human readable *assembly language*. It is important to have an understanding of how these tools work. In doing so, you will have a better idea of how to make the tool work better for you. The goal of this project is to create a program that can turn *machine code* into *assembly language.* You will be required to handle a small subset of the Intel Instruction Set (see below for the required mnemonics).

Recursive descent and Linear Sweep are the two main algorithms used in the disassembly process of a binary which constructs assembly instructions. There are advantages and disadvantages for both of these techniques, but also depending on the source code, complexity, size can really hurt or help depending on the technique you adopt.

Looking in depth at the linear sweep algorithm this process first looks through the binary's .txt section and disassembles the codes sequentially byte by byte. Some disassembly engines that use this technique include gdb, WinDbg, and objdump.

Some advantages of the linear sweep algorithm include the fact that this is a very simple algorithm to use and operate. Correct code execution along with the fact that all of the bytes will be decoded.Some disadvantages of the linear sweep algorithm include illegal instructions that get encountered throughout this process, and mistakes that are unintentionally left in the code to derail this algorithm. Another disadvantage is the fact that all of the bytes in this algorithm get decoded including data not involved in the executable.

Moving on to the recursive descent algorithm, we know that this approach is very similar to the linear sweep algorithm mentioned above but the branching instructions take more intelligent paths. This algorithm is recursive in regards to taking both branches of eligible branching statements and does a BFT of the execution tree providing the user all potential paths.

Some advantages include: This technique is far less susceptible to making mistakes unlike the linear sweep algorithm discussed above. Also If a jmp call or any other control flow instruction comes across the disassembler the target address of the operand also gets processed. Another advantage is the fact that this algorithm only disassembles bytes that the CPU encounters and not the extra bytes unlike the linear sweep algorithm. Disadvantages include the fact that this algorithm is relatively complex and takes more time to traverse all of the possible branches. Another downfall is the fact that this algorithm does not disassemble the whole executable if there is code not explicitly called based on address information.

IDA Pro is among one of the most popular disassemblers on the market along with its newest rival Ghidra. IDA Pro creates very detailed mappings of machine code to assembly which then can be created into something that may resemble the original source code. IDA uses recursive descent disassembly which analyzes each instruction for references in other locations and has a very high accuracy of disassembling these instructions for further analytical use.

Advanced techniques in IDA makes the disassembly easier to understand which also makes it one of the best debuggers in the business. Not only can IDA disassemble binaries but it also includes a huge package of debugging tools and helper functions to make the analyst's job easier by stepping through the function calls and plugin integration.

Requirements

R1) Your program must be written in any of the following: Python, Go, C, C++, Java. Please let me know if you have any issues with this requirement.
R2) Your program must not crash on any (in)valid inputs.
R3) You must use either the recursive descent algorithm or linear sweep algorithm.
R4) Your program must print to stdout.
R5) You must handle jumping/calling forwards and backwards and adding labels where appropriate with the following form (see Example 2 below).
    a) `offset_XXXXXXXXh:`
R6) If you hit an unknown opcode, your program must print the address, the byte and the assembly as: `db <byte>`. See the skeleton code for an example.
R7) This must work on the supplied samples, but it must work on other tests as well.
R8) Only the given opcodes shall be implemented (see the table below).
R9) Your program must have its input file specified using the "`-i`" command-line option.
R10) Only addresses, instruction machine code (i.e. the bytes that make up the instruction) and disassembled instructions/data and labels should be printed to the screen.

Assumptions

When writing your program, you can make the following assumptions about the input file.

A1) The input file is a binary that contains some x86 machine code.
A2) Code starts at offset 0 in the given file. This means you do not have to worry about the headers that are generally added by linkers.

Supported Mnemonics

For all instructions below, do not worry about the `ESP` register being the `r/m32` operand, except in addressing mode `0b11` (direct register). `ESP` is sometimes handled differently (SIB indicator) and you are **not** expected to handle that.

All register references will be 32-bit references. For example, you do not need to handle "`mov dl, byte [ ebx ]`", you only need to handle "`mov edx, dword [ ebx ]`". An immediate will be a 32-bit value while the displacement *may* be 8-bit or 32-bit in size. The only exceptions are the "`retn/retf imm32`" instructions.

add	**inc**	**nop**	**retn**
and	**jmp**	**not**	**sal**
call	**jz/jnz**	**or**	**sar**
clflush	**lea**	**out**	**sbb**
cmp	**mov**	**pop**	**shr**
dec	**movsd**	**push**	**sub**
idiv	**mul**	**repne cmpsd**	**test**
imul	**neg**	**retf**	**xor**

Instruction Details

For the "`repne cmpsd`" instruction, recall that the "`d`" in "`cmpsd`" refers to the data size. In this case, it is a DWORD or 32-bit value. Thus, in the Intel Manual we are looking for "`repne cmps m32, m32`".

For the "`movsd`" instruction, recall that the "`d`" in "`movsd`" refers to the data size. In this case, it is a DWORD or 32-bit value. Thus, in the Intel Manual we are looking for "`movs m32, m32`".
Note: This is a string operation and NOT the "move scalar double-precision" operation

For the "`sal/sar/shr`" instructions, you only need to support:
Note: "`sal`" and "`shl`" are the same opcode (Why is this?)

```
sal r/m32, 1
sar r/m32, 1
shr r/m32, 1
```

For the "`jz/jnz/jmp`" instructions you must implement:

```
jz  rel8
jz  rel32
jnz rel8
jnz rel32
jmp rel8
jmp rel32
jmp r/m32
jmp [ disp32 ]
jmp [ r/m32 + disp8 ]
jmp [ r/m32 + disp32 ]
```

For the "`retn/retf`" (listed as just "ret" in the Intel Instruction Manual) instruction family, you must implement the following:
Note: "`retn`" refers to "return near" and "`retf`" refers to "return far"

```
retn
retn imm16
retf
retf imm16
```

For the "`mov/add/and/not/or/pop/push/sub`" and similar instructions, you must implement (where applicable):

```
mov r/m32, imm32
mov r32, imm32
mov r32, [ disp32 ]
mov r32, r/m32
mov r32, [ r/m32 + disp8 ]
mov r32, [ r/m32 + disp32 ]
mov r/m32, r32
mov [ disp32 ], imm32
mov [ disp32 ], r32
mov [ r/m32 ], imm32
mov [ r/m32 + disp8 ], imm32
mov [ r/m32 + disp32 ], imm32
mov [ r/m32 + disp8 ], r32
mov [ r/m32 + disp32 ], r32
```

Sample Output

The following samples show how the input file is passed to your program and how output is to be formatted. The name of your program and the method by which it is invoked may be different depending on the language you use to implement it.

Example 1: With no jumps

```
$ disasm -i nojump.o
00000000:  31C0              xor eax,eax
00000002:  01C8              add eax,ecx
00000004:  01D0              add eax,edx
00000006:  55                push ebp
00000007:  89E5              mov ebp,esp
00000009:  52                push edx
0000000A:  51                push ecx
0000000B:  B844434241        mov eax,0x41424344
00000010:  8B9508000000      mov edx,[ebp+0x00000008]
00000016:  8B8D0C000000      mov ecx,[ebp+0x0000000c]
0000001C:  01D1              add ecx,edx
0000001E:  89C8              mov eax,ecx
00000020:  5A                pop edx
00000021:  59                pop ecx
00000022:  5D                pop ebp
00000023:  C20800            retn 0x0008
```

Example 2: With a conditional jump

```
$ disasm -i condjump.o
00000000:  55                push ebp
00000001:  89E5              mov ebp,esp
00000003:  52                push edx
00000004:  51                push ecx
00000005:  39D1              cmp ecx,edx
00000007:  740F              jz offset_00000018h
00000009:  B844434241        mov eax,0x41424344
0000000E:  8B5508            mov edx,[ebp+0x00000008]
00000011:  8B4D0C            mov ecx,[ebp+0x0000000c]
00000014:  01D1              add ecx,edx
00000016:  89C8              mov eax,ecx
offset_00000018h:
00000018:  5A                pop edx
00000019:  59                pop ecx
0000001A:  5D                pop ebp
0000001B:  C20800            retn 0x0008
```

Complete Opcode and Addressing Mode Requirements

Mnemonic/Syntax	Opcode	Addressing Modes
add eax, imm32	0x05 id	MODR/M Not Required
add r/m32, imm32	0x81 /0 id	00/01/10/11
add r/m32, r32	0x01 /r	00/01/10/11
add r32, r/m32	0x03 /r	00/01/10/11
and eax, imm32	0x25 id	MODR/M Not Required
and r/m32, imm32	0x81 /4 id	00/01/10/11
and r/m32, r32	0x21 /r	00/01/10/11
and r32, r/m32	0x23 /r	00/01/10/11
call rel32	0xE8 cd	Note: treat *cd* as *id*
call r/m32	0xFF /2	00/01/10/11
clflush m8	0x0F 0xAE /7	00/01/10 Note: *m8* can be a [disp32] only, a [reg], a [reg + disp8], or a [reg + disp32]. Addressing mode 11 is illegal.
cmp eax, imm32	0x3D id	MODR/M Not Required
cmp r/m32, imm32	0x81 /7 id	00/01/10/11
cmp r/m32, r32	0x39 /r	00/01/10/11
cmp r32, r/m32	0x3B /r	00/01/10/11
dec r/m32	0xFF /1	00/01/10/11
dec r32	0x48 + rd	MODR/M Not Required
idiv r/m32	0xF7 /7	00/01/10/11
imul r/m32	0xF7 /5	00/01/10/11
imul r32, r/m32	0x0F 0xAF /r	00/01/10/11
imul r32, r/m32, imm32	0x69 /r id	00/01/10/11
inc r/m32	0xFF /0	00/01/10/11
inc r32	0x40 + rd	MODR/M Not Required

jmp rel8	0xEB cb	Note: treat *cb* as *ib*
jmp rel32	0xE9 cd	Note: treat *cd* as *id*
jmp r/m32	0xFF /4	00/01/10/11
jz rel8	0x74 cb	Note: treat *cb* as *ib*
jz rel32	0x0f 0x84 cd	Note: treat *cd* as *id*
jnz rel8	0x75 cb	Note: treat *cb* as *ib*
jnz rel32	0x0f 0x85 cd	Note: treat *cd* as *id*
lea r32, m	0x8D /r	00/01/10 Note: *m* can be a [disp32] only, a [reg], a [reg + disp8], or a [reg + disp32]. Addressing mode 11 is illegal.
mov r32, imm32	0xB8+rd id	MODR/M Not Required
mov r/m32, imm32	0xC7 /0 id	00/01/10/11
mov r/m32, r32	0x89 /r	00/01/10/11
mov r32, r/m32	0x8B /r	00/01/10/11
movsd	0xA5	MODR/M Not Required
mul r/m32	0xF7 /4	00/01/10/11
neg r/m32	0xF7 /3	00/01/10/11
nop	0x90	MODR/M Not Required Note: this is really xchg eax, eax
not r/m32	0xF7 /2	00/01/10/11
or eax, imm32	0x0D id	MODR/M Not Required
or r/m32, imm32	0x81 /1 id	00/01/10/11
or r/m32, r32	0x09 /r	00/01/10/11
or r32, r/m32	0x0B /r	00/01/10/11
out imm8, eax	0xE7 ib	MODR/M Not Required

pop r/m32	0x8F /0	00/01/10/11
pop r32	0x58 + rd	MODR/M Not Required
push r/m32	0xFF /6	00/01/10/11
push r32	0x50 + rd	MODR/M Not Required
push imm32	0x68 id	MODR/M Not Required
repne cmpsd	0xF2 0xA7	MODR/M Not Required Note: 0xF2 is the repne prefix
retf	0xCB	MODR/M Not Required
retf imm16	0xCA iw	MODR/M Not Required Note: *iw* is a 16-bit immediate
retn	0xC3	MODR/M Not Required
retn imm16	0xC2 iw	MODR/M Not Required Note: *iw* is a 16-bit immediate
sal r/m32, 1	0xD1 /4	00/01/10/11
sar r/m32, 1	0xD1 /7	00/01/10/11
shr r/m32, 1	0xD1 /5	00/01/10/11
sbb eax, imm32	0x1D id	MODR/M Not Required
sbb r/m32, imm32	0x81 /3 id	00/01/10/11
sbb r/m32, r32	0x19 /r	00/01/10/11
sbb r32, r/m32	0x1B /r	00/01/10/11
sub eax, imm32	0x2D id	MODR/M Not Required
sub r/m32, imm32	0x81 /5 id	00/01/10/11
sub r/m32, r32	0x29 /r	00/01/10/11
sub r32, r/m32	0x2B /r	00/01/10/11
test eax, imm32	0xA9 id	MODR/M Not Required
test r/m32, imm32	0xF7 /0 id	00/01/10/11
test r/m32, r32	0x85 /r	00/01/10/11
xor eax, imm32	0x35 id	MODR/M Not Required
xor r/m32, imm32	0x81 /6 id	00/01/10/11
xor r/m32, r32	0x31 /r	00/01/10/11
xor r32, r/m32	0x33 /r	00/01/10/11

References

https://resources.infosecinstitute.com/linear-sweep-vs-recursive-disassembling-algorithm/#gref [1]

http://neilscomputerblog.blogspot.com/2011/10/disassembly-algorithms.html [2]

https://www.hex-rays.com/products/ida/#:~:text=A%20disassembler%20like%20IDA%20Pro,symbolic%20representation%20called%20assembly%20language.&text=That%20is%20why%20advanced%20techniques,that%20complex%20code%20more%20readable. [3]

https://resources.infosecinstitute.com/basics-of-ida-pro-2/#:~:text=IDA%20Pro%20is%20the%20best,we%20won't%20need%20here. [4]