Proposal

Hanzhou Tang
hanzhout@smu.edu

Jiutian Yu
jiutiany@smu.com

February 2019

Abstract

We want to build an assembly simulator which should support basic x86 assembly instructions.

1 Motivation

To implement a basic architecture is a direct way to reflect what we are going to learn in this class. We could also say we are going to implement a simple virtual machine. Beyond simply implementing the original proposal, we want to add some new functions based on this class to complete the architecture.

2 Why Java

It's tempted to implement an assembly simulator in a low level language like C or C++. However, after some considerations, we decided to choose Java. There are several reasons which will be show below.

2.1 Java is easy for memory management

With C++11, it has smarter pointers to make life easier [1]. However, it's implemented in library level instead of language supporting. Sometimes, when we mistakenly mixed raw pointers and smart pointers , smart pointers may become useless.

2.2 Java is easy to test

There are lots of powerful java testing framework to do unit test. For example, JUnit or Groovy Spock. Meanwhile, because Java support proxy object, it's much easy to mock objects and record function invoking.

2.3 Java is easy for package manage

With the help of gradle [3], it's very easy for us to import different libraries and do deployment. We believe it could save us lots of time and efforts.

2.4 Java is easy to integrate with REST API

If we could finish our project well, we may want to implement an online version for all users. We could easily provide REST API with the help of Spring [4].

3 Our registers

We do want to implement some basic functionality of x86 platform. So we decide to imitate ten 32 bits registers. A table, which is Table1, of our registers is shown below. The table originally comes from [2]. As you can see, we ignore all segment registers. The reason is that according to our design, only one process can run at one time and no context switch. We think, in this case, segment information is unnecessary. Maybe we're wrong, we may change our decision later.

4 Our instruction set

We want to support a subset of assembly instructions on x86 platform. By doing some researches [2], we provide a table which contains all instructions we want to support.

5 Design

For now, we decided to split the project into 3 components. They are assembler, instructions and virtual machine.

5.1 assembler

We based on Microsoft XASM assembler and followed its' convention and directives. However, because our time and ability is limited, we cannot implement all features in XASM. But we at least want to implement the basic functionality, for example, the ability of define different segments, the ability of some basic directives, like $ and proc. We divided our assembler into 2 parts. One part is a lexer, which is responsible for tokenizing input asm file. Another part is a parser, which is responsible for convert tokens into machine codes. After we finishing our assembler, it shall be able to read asm file and convert it into a binary file which contains data and machine code. (The format of machine code is defined by ourselves for convenience). From now on, we do finish the lexer part and currently working on parser part. If we have enough luck, we can finish parser in the spring off.

Here is a screenshot of outputs from our lexer. The whole asm file and

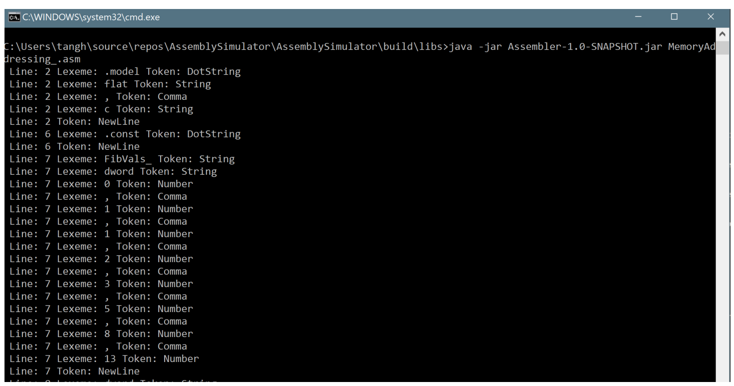

Figure 1: Outputs from our lexer

output are attached in our report.

5.2 Instructions

As we said before, we want to convert asm file into binary file, which means we need design format of machine codes. We followed the convention, all machine codes are converted into 4 byte machine codes. For assembler, we write the instructions into binary file. For virtual machine, we load instructions from binary file. This should be the common part of assembler and virtual machine.

5.3 Virtual Machine

The basic functionality of our virtual machine is loading the binary file and executing it. We don't care about performance and optimization for our virtual machine, which means the effectiveness of our virtual machine may be pretty low. However, we want to implement 5 stage pipeline in our virtual machine to show what we are learning from the Architecture class. Later, we may add more staffs into our virtual machine if we have time, like branch prediction. We think the virtual machine is really not easy to implement, thus we plan to focus more in this part.

6 To do

1. We need to transfer assembly files to binary files.

2. We need our simulator is able to read these files.

3. If we have time, we can add one LRU cache to our simulator to optimize the storages. We can also implement something having a strong connection with what we learn in this class

References

[1] Josuttis, N. M. *The C++ standard library: a tutorial and reference.* Addison-Wesley, 2012.

[2] Kusswurm, D. *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX.* Apress, 2014.

[3] Muschko, B. *Gradle in action.* Manning, 2014.

[4] Walls, C., and Breidenbach, R. *Spring in action.* Dreamtech Press, 2005.

Register	Descriptions
EAX	Accumulator. 0 to 7 can be refered as AL. 8 to 15 can be refered as AH. 0 to 15 can be refered as AX.
EBX	Memory pointer, base Register. 0 to 7 can be refered as BL. 8 to 15 can be refered as BH. 0 to 15 can be refered as BX.
ECX	Loop control. 0 to 7 can be refered as CL. 8 to 15 can be refered as CH. 0 to 15 can be refered as CX.
EDX	Integer multiplication, integer division. 0 to 7 can be refered as DL. 8 to 15 can be refered as DH. 0 to 15 can be refered as DX.
ESI	String instruction source pointer.
EDI	String instruction destination pointer.
ESP	Stack Pointer.
EBP	Stack frame base Pointer.
EIP	Instruction pointer register.
EFLAGS	Flag register.

Table 1: The registers we want to imitate

mov	Copy data from one place to another place.
push	Push register, memory location or immediate value onto stack.
pop	Pop the first item from stack.
add	Add two number
sub	Subtraction
cbw	Sign-extends register AL.
cwd	Sign-extends register AX.
bswap	Reverse the bytes of a 32-bit register.
and	Logic and.
or	Logic or.
xor	Logic xor.
not	Logic not.
sal/shl	Left shift.
sar	Arithmetric right shift.
shr	Logic right shift.
cmpsb/cmpsw/cmpsd	Compare the values at location.
lodsb/lodsw/lodsd	Loads the values at location.
stosb/stosw/stosd	Save the values at register to memory.
rep/repne	Repeat a specified instruction by condition
jmp/jcc/jecxz	Unconditonal/conditional jump
call	Push content to stack then do unconditonal jump.
ret	Pop stack then do unconditonal jump.
enter	Create a stack frame for function.
leave	Remove a stack frame of function.
loop/loope/loopz/loopne/loopnz	Loop.
nop	Advance the instruction pointer.

Table 2: The instructions we want to imitate