Spark Session: libasm

updated: 01/04/2021

Project description:

> Get familiar with assembly language

This tutorial was written with help from Thijs Bruineman (tbruinem) and this great tutorial series.

Topics

1. nasm
2. Registers
3. Instructions
4. Syscall
5. Sections
6. Stack Alignment

Setting Up nasm

For libasm, we'll be using the Netwide Assembler (`nasm`) to compile our assembly code.
If you haven't already, install `nasm` on your system using the following command:

- For Linux: `sudo apt update && sudo apt install nasm`
- For macOS: install Homebrew if you don't have it yet, then run `brew install nasm`

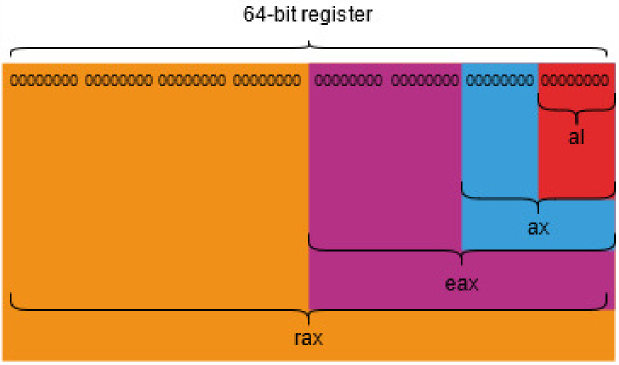
Registers

1. Registers are internal memory storage locations in the processor that temporarily hold memory.
   In x86_64 architecture, we have access to 64-bit registers. What does that "64-bit" mean? (5 mins)
   > that the registers can hold 64 bits of data

2. We won't go into details about all the registers. Broadly, some registers are used for specific purposes - such as segment registers and the Flags register - while some are for general use. These latter ones are called **General Purpose Registers** and there are **16** of them in 64-bit x86 architecture. What are the registers? (10 mins)
   > rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8-r15

3. You're not limited to working with these registers in their 64-bit entirety though. You can access smaller "sections" of these registers through identifiers. For example, the least significant 2 bytes (16

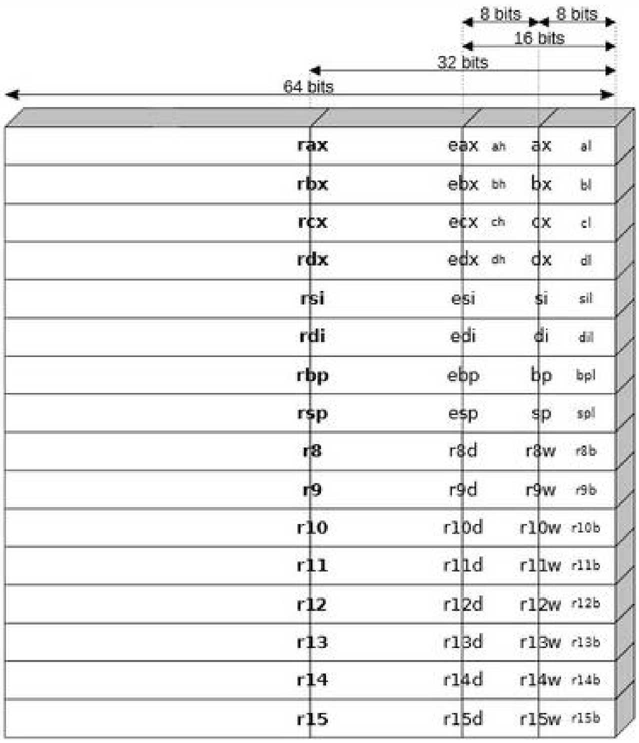
bits) of RAX can be treated as a 16-bit register called AX. (15 mins)



- **Question**: what is AL in this case?

  > the least significant byte of the accumulator register - bits 0 to 7 of RAX

- Here's a table showing the breakdown of every general purpose register:

  source, with many other helpful tips

- **Question**: how would you access the lowest 8 bits of R8?

  > r8b

- **Question**: are these "sub"registers independent? For example, will modifying `al` affect `ax`? > No, they're not independent registers. Modifying `al` will affect `ax`, but modifying `ah` won't affect `al`.

4. In 64-bit architecture, most registers no longer serve the special purposes for which they're named - like "accumulator" (rax) and "counter" (rcx). But there's still some things to keep in mind about certain registers. (15 mins)

   - RAX is commonly used to store _ when functions are called within assembly code.

> function return values. Also used to pass system call numbers such as for calls to `write`

- RSP is called the Stack Pointer and RBP is the Base Pointer. What do they do?

  > RSP points to the topmost element in the stack, RBP points to base of stack

- Lastly, 6 specific registers are used to pass parameters to functions. What are they and which arguments do they correspond to?
  > 1st: RDI, 2nd: RSI, 3rd: RDX, 4th: RCX, 5th: R8, 6th: R9

5. There are certain registers whose values are preserved across function calls - **callee-saved registers** - and registers whose values are not preserved and must be saved if you want to make sure your values aren't changed - **caller-saved registers**. (10 mins)

   - Which are the callee-saved registers?

     > RBX, RSP, RBP, and R12–R15

   - Note that which registers are caller/callee-saved vary by system, thus "calling conventions".

   - The convention states that the original values within callee-saved registers should be saved by the called function (the callee). The expectation is that those registers hold the same value after the called function returns. What does this mean if we wish to use a callee-saved register?

     > must save and restore original value by pushing at start and popping at end (which we'll get into in a bit)

   - Here's a handy table with the registers and their usages: link

Break (5 mins)

Instructions

There are a lot of instructions you can use in x86 assembly, but we're going to focus on a few key ones.

1. Figure out what the following instructions do: (30 mins)

instruction	example
mov	mov rax, rbx
push	push rax
pop	pop rax
add	add rax, 42
sub	sub rax, 42
inc	inc rax
dec	dec rax
cmp	cmp rax, 42
jmp	jmp _main
je/jne/jl/jg	je _done

instruction	example
call	call _printf
ret	ret

- Make sure you know which operand is affecting which!
  > e.g. `add rax, 42` -> rax = rax + 42

2. What effect would the following `jmp` instruction have? (5 mins)
   `_main: jmp _main`
   > program would be stuck in an infinite loop

3. Registers can also serve as pointers. Putting square brackets - `[]` - around registers allows you to access the value the register is **pointing to**, rather than the value of the register itself. (5 mins)

   - What's the difference then between `mov rax, rbx` and `mov rax, [rbx]` ?
     > first instruction loads value in rbx register into rax, second loads value rbx is pointing to into rax

Syscall

Now let's look at system calls or **syscall**s, which allow a program to request a service from the kernel.

1. All syscalls have an associated ID. This ID is what you pass into RAX within your assembly code to call on a system function. These IDs vary by operating system. (5 mins)

   - For Linux: syscall IDs as found in unistd_64.h
   - For macOS: syscall IDs in syscalls.master. Note: You'll need to add `0x200000` before each ID.
   - What are the syscall IDs for `write()` and `exit()` for your system?
     > `write` is 1 (Linux) and 0x2000004 (mac), `exit` is 60 (Linux) and 0x2000001 (mac)

2. As with normal functions, syscalls can take arguments. Just like in C, `write` takes an fd, a buffer, and the number of bytes to write. We talked about parameter registers earlier. If you want to call `write` , which parameters would you pass to which register? (10 mins)

   Here's a convenient table for you:

syscall	rax	rdi	rsi	rdx	rcx (r10 for Linux)	r8	r9	
write								
> RAX: 1 or 0x2000004, RDI: fd, RSI: address of string, RDX: byte count								

syscall	rax	rdi	rsi	rdx	rcx (r10 for Linux)	r8	r9	
3. Let's write a simple assembly program that calls `exit()` with `0` as an argument. This should make the program exit with a status of 0, indicating no errors. (15 mins)								
4. First, create a `.s` file. Here's what the first part of your code should look like:								
`section .text ; this is the section for code global _main ; this tells the kernel where the program begins _main:`								
5. Note: for Linux users, you'll need to remove the `_` before `_main`.								
6. To make a system call, you need to:								
7. pass the syscall ID into RAX								
8. pass any arguments for the syscall								
9. use the `syscall` instruction								
Go ahead and turn those steps into assembly code.								
`mov rax, 60` ; or 0x2000001 for mac								
`mov rdi, 0`								
`syscall`								
0. Although you'll be creating a library for libasm, today we're just going to make a standalone program. So the compilation steps will be different. To compile your `.s` file, run these commands:								
1. **Note**: remember to change `myfile.s` & `myfile.o` to your actual file name								
2. For Linux: `nasm -felf64 myfile.s && gcc myfile.o`								
3. For mac: `nasm -fmacho64 myfile.s && gcc myfile.o`								
4. Run `./a.out` and then `echo $?`. Does it output `0`?								

syscall	rax	rdi	rsi	rdx	rcx (r10 for Linux)	r8	r9	
Break (5 mins)								

Sections

1. Assembly files can be divided into 3 **sections** (there are more but we won't get into them now): `.data`, `.bss`, and `.text` (as seen earlier). What are each of these sections meant for? (5 mins)
   > data: where data is defined (initialized data) before compilation, bss: where data is allocated for future use (uninitialized data), text: executable instructions

2. Let's make use of a new section: the `.data` section. We're going to write a program that outputs "Hello, world!" followed by a newline. (20 mins)

   - Let's start by declaring a new section (section order doesn't matter in an assembly file):

     ```
     section .data
         text db "Hello, world!", 10
     ```

     - What is "text" here? And what does "db" mean? What is the "10" at the end?

       > `text` is a label we can use in our instructions, it's a pointer to the memory address of the string. `start` is also a label. they are essentially variable names.
       > `db` stands for define bytes, it means we are going to define some raw bytes (each character in our string is a byte).
       > `10` is ascii for the newline character.

   - Next declare your `.text` section as you did in the previous exercise.

   - For this exercise, we're going to call `write()` first to output our `text` onto **stdout**. Remember the instruction order for calling `exit()` earlier. What values do you need to move into which registers?

     > ; code example for Linux
     > main:
     > mov rax, 1 ; write for Linux
     > mov rdi, 1 ; stdout
     > mov rsi, text
     > mov rdx, 14 ; len of text
     > syscall

   - Finish with a call to `exit()` again like you did earlier.

   - Compile with the same commands you used earlier. Do you get "Hello, world!"?

- Note: your compiler may throw up some warnings, but if your program was compiled successfully, just ignore it!

Stack Alignment

Stack alignment can be a tricky thing to understand. What you need to know is that x86_64 requires that your stack be aligned on a 16-byte boundary.

- When your program starts at `main` (or `start`), `rsp` (the stack pointer) is 16-byte aligned.
- If you make an external function call, however, such as `call printf` or `call myownfunction`, an **8-byte return address** is pushed onto the stack. Because we're temporarily leaving this function and we need to know where to come back to, right?
- But this means the stack is now misaligned by **8 bytes**. So how do we re-align our stack? (15 mins)
  > By either pushing something that is 8 bytes, like a register (e.g. rbx), and popping it after function call or `sub rsp, 8` at beginning and `add rsp, 8` at end.
  > good clear explanation about alignment: link

You can find more detailed explanations of the stack and alignment online.\
Here's a helpful link for later: what does it mean to align the stack

Bonus

1. Write a `compare` function in assembly that compares 2 integers and returns:

   - `1` if `a > b`;
   - `-1` if `a < b`;
   - `0` if `a == b`.

   The function prototype is `int compare(int64_t a, int64_t b)`.

2. You'll also make a test main C file. It should:

   - include for the `int64_t` types;
   - have your `compare` function prototype;
   - call your `compare` function with a variety of inputs (positive, negative, 0s) and print the return.

3. If you're on macOS, you'll probably need to prefix your function with `_` in your assembly code, so that it's `global _compare` and `_compare:`. Because reasons. Linux does not require this.

4. Compile and run it to see if your function is working correctly.

   - For Linux: `nasm -felf64 compare.s && gcc compare.o main.c`
   - For macOS: `nasm -fmacho64 compare.s && gcc compare.o main.c`