intranet.atlasschool.com

Concept: [EYNTK] x86-64 Assembly | Atlas Intranet

15–18 minutes

Everything you need to know

Introduction

In this concept, we'll go through the basics of assembly programming. We won't be able to cover everything about assembly, but enough for you to complete your project.

Throughout this concept, we'll talk about the way to manipulate data in assembly using registers, instructions, and the stack. We'll also see how to control the flow of your program using loops and conditions.

It is strongly advised to search the internet for help if you have any question. But be careful, there are different syntaxes for assembly and we're only gonna use the Intel syntax.

Environment

Here's a list of the different tools used during this concept:

- Linux 3.13.0-92-generic
- Ubuntu 14.04 LTS
- NASM version 2.10.09
- gcc version 4.8.4

Assembly syntax

- x86 Assembly
- 64-bit architecture
- Intel

Assembly to ELF object file

Do you remember the different steps to transform a C source file onto an executable file? Here it goes:

- Preprocessing (Remove comments, replace macros, include headers, ...)
- Compilation (Compile C into assembly)
- Assembly (Assemble to object file)
- Linking (Link object files to form the executable)

It is pretty easy to guess that we will only need the two last steps in order to transform our

assembly code onto an executable. One thing though: we're only gonna use `gcc` for linking.

In order to assemble our code to form an object file, we'll use the tool `nasm` (apt-get install nasm).

Here's the command line we'll use to assemble our `asm` files:

```
$ nasm -f elf64 <file.asm>
```

This command forms a `.o` file that we can then link with other `.o` files (even if they were compiled from C source code).

Registers

In assembly, you won't be able to declare and use variables. For arithmetic, you are given a bunch of `registers` to work with. And if you need to save a value in memory before calling a function that could erase it, you'll have to push it into the stack (That's exactly how variables work in C).

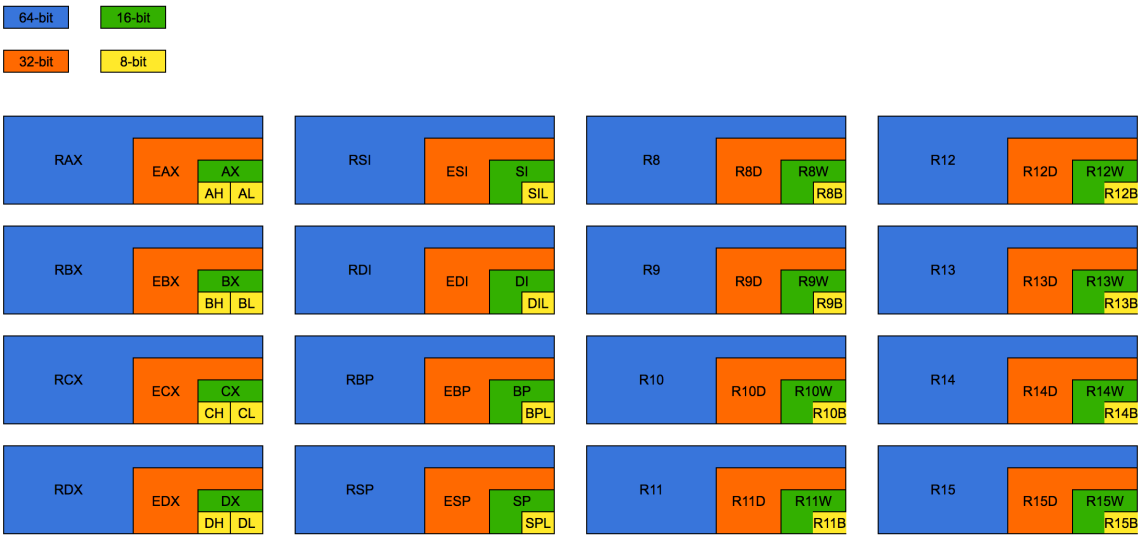
Registers are small amount of fast storage within the CPU. Some of them have a specific role, and some other are free to be used by the user. Registers are limited, you'll only be able to use a bunch of them in your code.

In the next section, we will, describe the most known and most used registers. The number of registers depends on your CPU. Some have more than others. The registers described in the next sections are present on all recent CPUs.

General-Purpose Registers (GPR)

There are **16 registers** to be used as you wish. However, some of them have specific roles, and will sometimes hold values you won't want to override.

For example, RAX can be used as you wish, but whenever you will call a function or a syscall, this register will hold their return value; thus, if something was store there, it'll be erased.



As you can see on the image above, registers are designed to be used with different sizes.

For example, let's take RAX again, and let's say you just called a function that returned an `int`. We know that an `int` is `4 bytes` (32-bit), so instead of retrieving the return value of the function you just called using RAX, you better use EAX. If the function returns a `char` (1 byte/8-bit), you'll use AL.

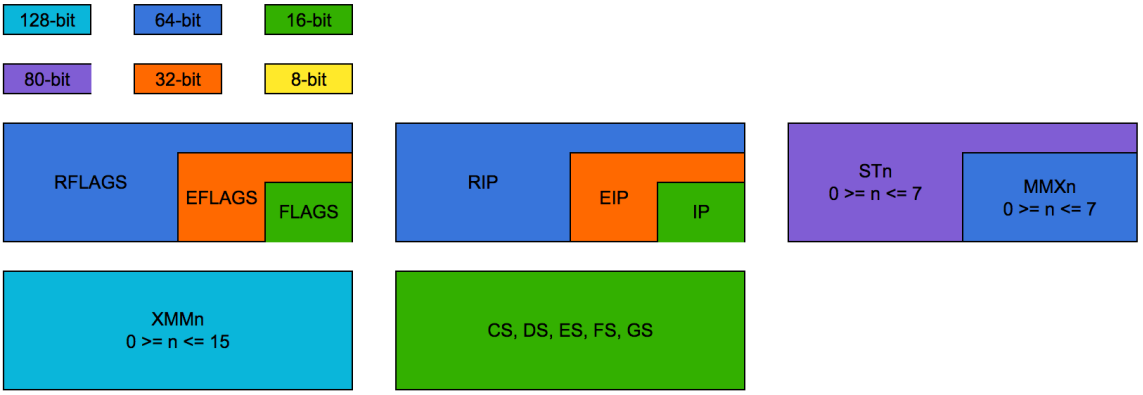
The same behaviour applies for all registers, and instructions are size-dependent. It means

that if you tell an instruction to read or write from/into a 16-bit register, you have to provide it a value of the same size. Don't worry, we'll talk about instructions a bit later.

GPR Specific Roles

- `RSP`: Stack pointer
- Points to end of the stack (lowest address)
- This register will automatically be updated every time you push something on the stack or pop something from it
- Used implicitly by `PUSH`, `POP`, `CALL`, `RET`, `ENTER`, `LEAVE`, ...
- `RBP`: Frame pointer
- This register is never automatically updated but its most common purpose is to save a pointer to the current stack frame
- Used implicitly by `ENTER` and `LEAVE`
- `RCX`: Counter for loop and string instructions
- `RSI`: Source pointer for string instructions
- `RDI`: Destination pointer for string instructions

Other Registers



- `RFLAGS`: For conditional jumps
- `RIP`: Current instruction pointer
- `ST0...7` (80-bit): legacy floating point numbers
- `MMX0...7` (64-bit) & `XMM0...15` (128-bit): vectorial instructions
- Segment registers & numerous other registers: for operating system instructions

Flags

Flags are set by most instructions. For example when you compare two values using `CMP`, some flags will be set to determine the difference between those two values. The same flags will then be implicitly used if you choose to use a conditional jump.

- `CF`: Unsigned carry (integer overflow)
- `OF`: Signed overflow
- `ZF`: Zero (result is null)
- `SF`: Sign (result is negative, leftmost bit = 1)

- PF: Parity (rightmost bit = 0)

Instructions

Now that we've seen how to store data with registers, it is now time to see how to manipulate data and perform some arithmetic using the different assembly instructions.

All instructions and directives are case insensitive.

Like in C, indentation does not matter in assembly as well as blank lines, but a good code is a readable and understandable code.

Assembly Directives

- Comment
- A semi-colon sets the rest of the line as a comment
- `; this is a comment`
- Set architecture with `BITS <N>`
- To be done at the very beginning of the file
- In our case: `BITS 64`
- Change section with `SECTION .<name>`
- The code is written in the `.text` section
- Static data is declared in the `.data` section (or `.rodata` for read-only data)
- Set symbol: `<label>:`
- Labels are like anchors to which you can jump
- They are not functions, but they can be considered as so
- Export symbol: `GLOBAL <label>`
- Make a symbol visible for external files
- Makes the symbol `<label>` callable
- Import symbol: `EXTERN <label>`
- If the symbol `<label>` has been exported with `GLOBAL`, `EXTERN` makes it available in your file
- Put bytes (static data): `DB <data>`/`RESB <length>`

Static Sections

- Code: `.text`
- Read-only data: `.rodata`
- Read/write data: `.data`
- Unitialized data: `.bss`
- See `nm` and `objdump`

Example

```
alex@~/0x09-libasm/Concept$ cat example_0.asm
BITS 64
```

```
global my_function      ; EXPORT our function 'my_function'
extern another_function     ; IMPORT the function 'another_function'

section .data
               ; Declare static data
    my_str db "Holberton", 0Ah, 0
    ; "Holberton", followed by a new line (0A hexa), and \0


section .text
               ; Code section

my_function:           ; This is a symbol
    ; Do some
    ; some
    ; stuff
    ; here
alex@~/0x09-libasm/Concept$ nasm -f elf64 example_0.asm
alex@~/0x09-libasm/Concept$
```

Instructions Syntax

Depending on the instruction, the order of the parameters can differ.

- `INSTR SRC/DEST` or `INSTR SRC/DEST, SRC`
- Source (SRC): immediate value, register or memory
- Destination (DEST): register or memory
- Not both arguments can be memory
- Type determined from other parameter if available or specified explicitly using a prefix:
- `BYTE` (8-bit)
- `WORD` (16-bit)
- `DWORD` (32-bit)
- `QWORD` (64-bit)
- Example with the `ADD` instruction
- Syntax: `ADD SRC, DEST`
- Interpreted as `SRC = SRC + DEST`
- `ADD RAX, RDX`
- Can be interpreted as `RAX = RAX + RDX`
- Here, both values are registers
- `ADD RAX, 12`
- Can be interpreted as `RAX = RAX + 12`
- Here, 12 is a direct value

Memory Access

- [immediate + register + register * coefficient]

- immediate: immediate value (explicit constant)
- register: general-purpose register
- coefficient: 1 (default), 2, 4 or 8
- All are optional
- Type determined from other parameter if available or specified explicitly using a prefix:
- `BYTE` (8-bit)
- `WORD` (16-bit)
- `DWORD` (32-bit)
- `QWORD` (64-bit)
- Example 1: `MOV RDX, [RBX + RCX * 4]`
- Move the `64-bits` at the address `RBX + RCX * 4` into `RDX`
- Example 2: `MOV BYTE [RDI + 1337], 98`
- Move the `8-bit` value `98` at the address `RDI + 1337`

Main Instructions

- Data movement: `MOV`, `XCHG`, `PUSH`, `POP`
- Type conversion:
- `CBW`: Convert Byte to Word
- `CWD`: Convert Word to Doubleword
- `CDQ`: Convert Double to Quad
- Arithmetic:
- `NEG`: Two's Complement Negation
- `INC`: Increment
- `DEC`: Decrement
- `ADD`: Arithmetic Addition
- `SUB`: Subtract
- `IMUL`: Signed Multiply
- `MUL`: Unsigned Multiply
- `IDIV`: Signed Integer Division
- `DIV`: Divide
- Bitwise:
- `NOT`: One's Complement Negation
- `AND`: Logical And
- `OR`: Inclusive Logical OR
- `XOR`: Exclusive OR
- Bitshifts:
- `SHL`: Shift Logical Left

- SHR: Shift Logical Right
- SAL: Shift Arithmetic Left
- SAR: Shift Arithmetic Right
- ROL: Rotate Left
- ROR: Rotate Right
- Resultless (flags only, for conditional jumps):
- CMP: Compare
- TEST: Test For Bit Pattern

Branching

Instructions used to move the current instruction pointer

- Unconditional jump: JMP
- Conditional jump (depends on RFLAGS):
- See Jump Instructions Table
- Function call: `CALL <symbol>`
- Equivalent to `PUSH RIP` + `JMP <symbol>`
- Function return: RET
- Equivalent to `POP RIP`
- System function call (kernel interface): SYSCALL

I strongly advise you to put put this page in your favorites, you're gonna need it a lot.

Memory

As said earlier, you are given a set of registers in order to manipulate your data with the different instructions. But whenever you'll need to save data, you'll need to push it into the stack. But before doing so, you'll need to setup the stack when entering in a new function, and restore it when leaving your function.

Stack

- Grows downwards
- Contains the function local state
- RSP = top of stack
- RBP = frame pointer (beginning of local variables), for convenience only

Stack Frame Setup

- Optional
- RSP varies (pushes and pops), but RBP fixed
- Easier to access local variables: `RBP – constant`
- Function prologue: `PUSH RBP` + `MOV RBP, RSP`
- Function epilogue: `MOV RSP, RBP` + `POP RBP`

- Alternatives: `ENTER` and `LEAVE`

  I advise you to watch this video that explains very well why and how the stack is setup when a procedure is called.

Functions and system calls

Function calling

To call a function in assembly, you'll need to setup the parameters to be passed to it:

- The **6 first integer/pointer parameters** have to be stored in order in `RDI`, `RSI`, `RDX`, `RCX`, `R8` and `R9`
- The **8 first floating-point number (FPN) parameters** have to be stored in order in `XMM0...7`
- All the remaining parameters have to be pushed in the stack
- For variadic function (like the `printf` family):
- The **number of FPN parameters** has to be stored in `RAX` (Not the number of parameters!)
- The `RBP`, `RBX`, `R12`, `R13`, `R14` and `R15` registers must be preserved by callee
- All Other registers may be altered at will
- Return value in `RAX` (integer/pointer) or `XMM0` (FPN)
- Finally, call the function with `CALL <function>`

System Call

The calling convention is slightly different for system calls:

- The same registers are used for parameters, except `R10` is used instead of `RCX` for the 4th parameter
- It is only possible to pass integers and pointers, **no floating-point parameters**
- System call number in `RAX` (see `man 2 syscalls`)
- `RCX` & `R11` may be overwritten
- Finally, call the system call with `SYSCALL`
- Return value in `RAX`, on error **`RAX`** = **`-errno`** (between −4095 and −1)

Exercises

That was A LOT of information.. But don't worry, before you jump in the project we're gonna go together through some easy exercises, for you to be sure to understand.

I'll give you a problem, and its solution, but make sure to try to solve each problem on your own before watching the solution. It's up to you, but I would suggest that you search the internet for help, because you won't have the solutions for the project. Look at the solution only if you are sure that your exercise is completed.

Add me

Write a simple procedure in Assembly that takes two `integers` as parameters (32-bit), and returns their sum as an `integer` (32-bit).

For this problem you don't have to use any local variable, so you don't have to take care of the stack yet.

The purpose here is to learn how to do simple arithmetic on registers and return a value

Expected output

```
alex@~/0x09-libasm/Concept$ cat add_me_main.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int add_me(int a, int b);

/**
 * main - Program entry point
 * @argc: Arguments counter
 * @argv: Arguments vector
 *
 * Return: EXIT_SUCCESS or EXIT_FAILURE
 */
int main(int argc, const char *argv[])
{
    int a;
    int b;
    int res;

    if (argc < 3)
    {
        dprintf(STDERR_FILENO, "Usage: %s <a> <b>\n", argv[0]);
        return (EXIT_FAILURE);
    }

    a = atoi(argv[1]);
    b = atoi(argv[2]);
    res = add_me(a, b);

    printf("%d + %d = %d\n", a, b, res);

    return (EXIT_SUCCESS);
}
alex@~/0x09-libasm/Concept$ nasm -f elf64 add_me.asm
alex@~/0x09-libasm/Concept$ gcc -c add_me_main.c
alex@~/0x09-libasm/Concept$ gcc add_me.o add_me_main.o
alex@~/0x09-libasm/Concept$ ./a.out 402 98
402 + 98 = 500
alex@~/0x09-libasm/Concept$ ./a.out 1 1
1 + 1 = 2
alex@~/0x09-libasm/Concept$ ./a.out 23424 234234
23424 + 234234 = 257658
alex@~/0x09-libasm/Concept$
```

Solution

Here's a link to the solution of this exercise

Swap

Write a procedure in Assembly that takes two pointers to `int` as parameters (64-bit), and swap the values they point to. Your procedure does not return anything.

The purpose here is to learn how to manipulate data that is not stored in registers but in memory.

Expected output

```
alex@~/0x09-libasm/Concept$ cat swap_main.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

void swap(int *a, int *b);

/**
 * main - Program entry point
 * @argc: Arguments counter
 * @argv: Arguments vector
 *
 * Return: EXIT_SUCCESS or EXIT_FAILURE
 */
int main(int argc, const char *argv[])
{
    int a;
    int b;

    if (argc < 3)
    {
        dprintf(STDERR_FILENO, "Usage: %s <a> <b>\n", argv[0]);
        return (EXIT_FAILURE);
    }

    a = atoi(argv[1]);
    b = atoi(argv[2]);

    printf("Before: a = %d, b = %d\n", a, b);

    swap(&a, &b);

    printf("After: a = %d, b = %d\n", a, b);

    return (EXIT_SUCCESS);
}
alex@~/0x09-libasm/Concept$ nasm -f elf64 swap.asm
alex@~/0x09-libasm/Concept$ gcc -c swap_main.c
alex@~/0x09-libasm/Concept$ gcc swap.o swap_main.o
alex@~/0x09-libasm/Concept$ ./a.out 402 98
Before: a = 402, b = 98
After: a = 98, b = 402
```

```
alex@~/0x09-libasm/Concept$ ./a.out 1234 5678
Before: a = 1234, b = 5678
After: a = 5678, b = 1234
alex@~/0x09-libasm/Concept$
```

Solution

Here's a link to the solution of this exercise

Print alphabet

Write a procedure in Assembly that prints the lowercase alphabet on the standard output. You're not allowed to declare any data in the different data sections. You'll have to use a loop. You'll have to call the `write` system call

The purpose here is to learn how to make a system call, how to build a loop, and how to handle local variables in the stack

Tips:

- Symbols
- `CMP`
- `JMP` and `Jxx` instructions
- `SYSCALL`: The syscall number of `write` is `1`

Expected output

```
alex@~/0x09-libasm/Concept$ cat alphabet_main.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

void print_alphabet(void);

/**
 * main - Program entry point
 *
 * Return: EXIT_SUCCESS or EXIT_FAILURE
 */
int main(void)
{
    print_alphabet();
    write(1, "\n", 1);
    return (EXIT_SUCCESS);
}
alex@~/0x09-libasm/Concept$ nasm -f elf64 alphabet.asm
alex@~/0x09-libasm/Concept$ gcc -c alphabet_main.c
alex@~/0x09-libasm/Concept$ gcc alphabet.o alphabet_main.o
alex@~/0x09-libasm/Concept$ ./a.out
abcdefghijklmnopqrstuvwxyz
alex@~/0x09-libasm/Concept$
```

Solution

Here's a link to the solution of this exercise

Resources

- List of registers
- List of Intel Instructions
- A video to explain how the stack frame works
- Github repository containing the sources used in this concept as well as the exercises

Good luck \o/