OS LAB1: ASSEMBLY TUTORIAL

Miro Mannino

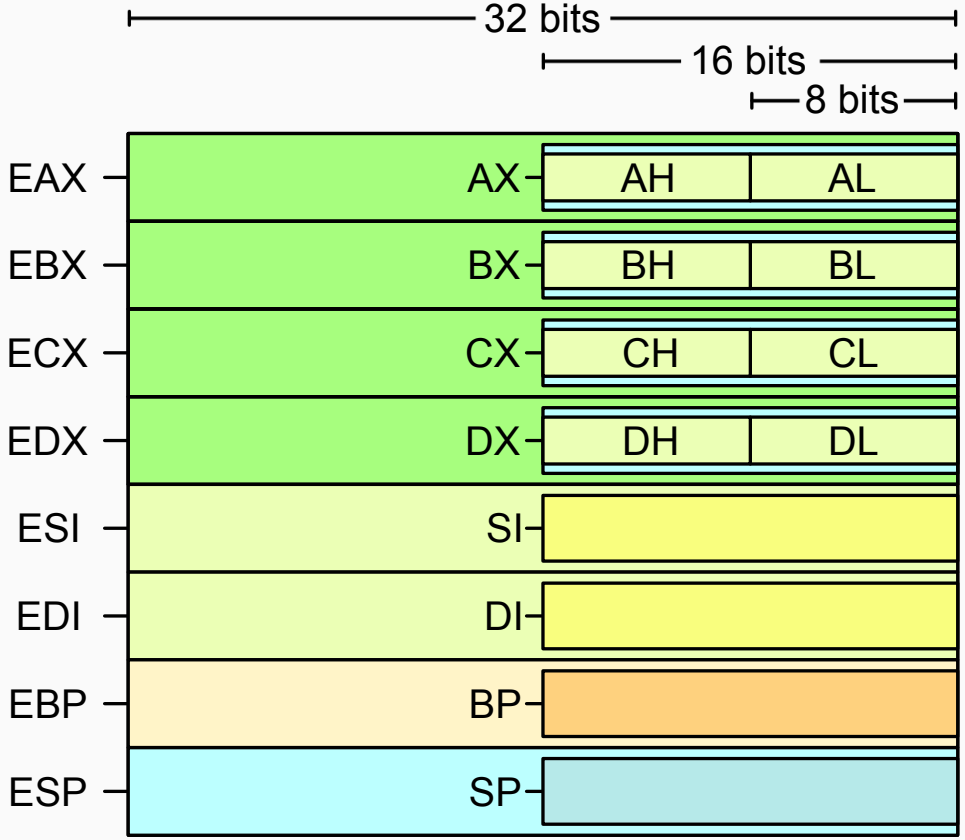
02 September 2022

x86 registers

x86 registers

The x86 architecture contains eight general-purpose 32-bit registers.

These registers are mainly used to perform address calculations, arithmetic, and logical calculations.



x86 registers

Conventionally these registers are used in the following way:

EAX	Accumulator for operands and results
EBX	Used to address memory in the DS segment
ECX	Used as counter for string or loops
EDX	Used as accumulator's backup (e.g. I/O pointer)
ESI	Instruction source pointer in the DS register operations.
EDI	Instruction destination pointer in the ES segment
EBP	Pointer to data on the stack in the SS segment
ESP	Stack pointer in the SS segment

x86 segment registers

There are also other segment registers:

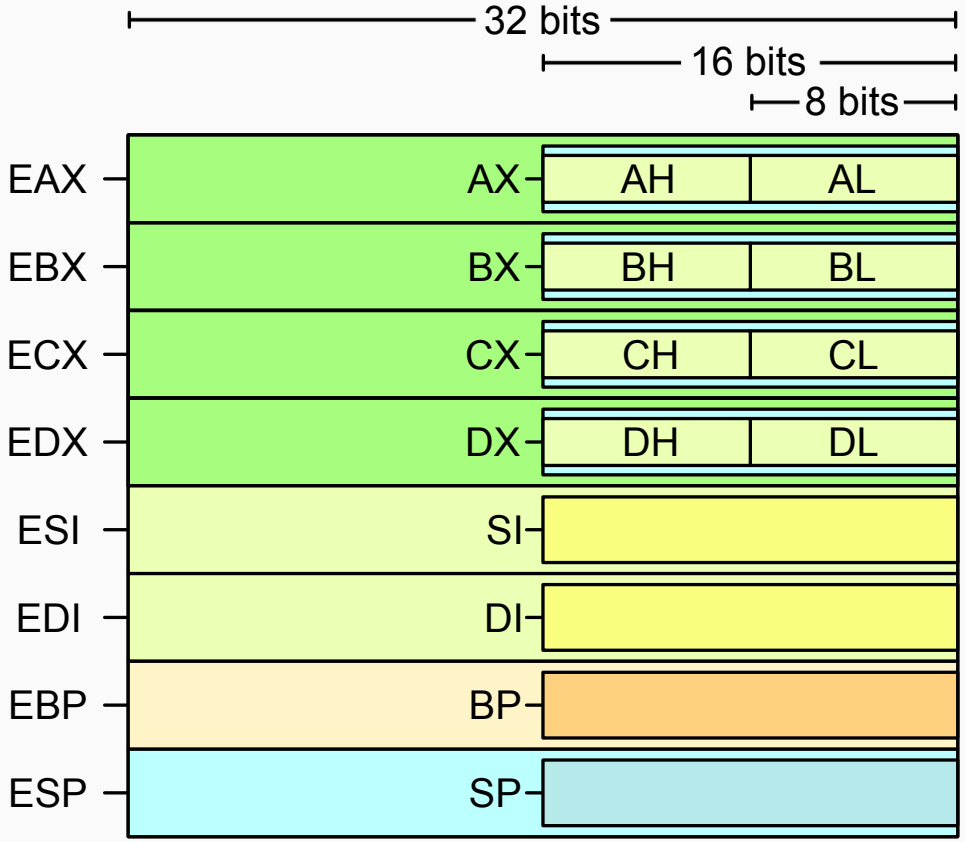
CS	Code segment register
SS	Stack segment register
DS	Data segment register
ES	Data segment register
FS	Data segment register
GS	Data segment register

The four data segment registers provide flexible and efficient ways to access data.

x86 registers

32-bit registers EAX, EBX, ECX, and EDX can also be treated as a 16-bit register or as two 8-bit registers. For example, the least significant 2 bytes of EAX are called AX, and the least significant byte of AX is called AL.

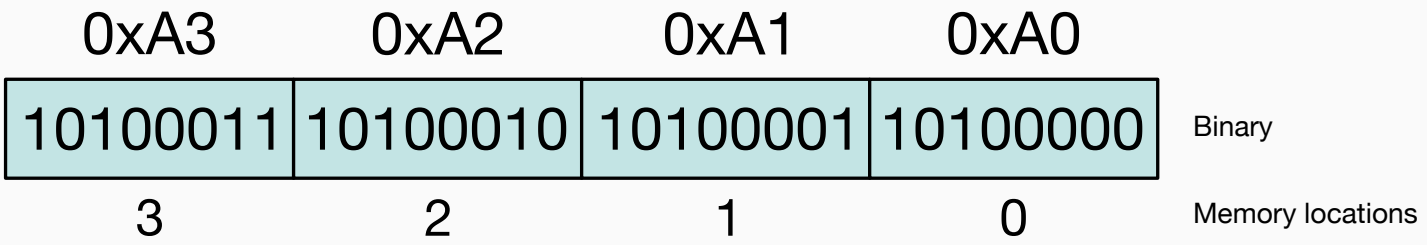
32-bit registers ESI, EDI, EBP, and ESP can also be treated as a 16-bit register.



ENDIANNESS

The x86 architecture is little-endian. This means that multi-byte values are written with the least significant byte first.

For example, the 32-bit number 0xA3A2A1A0 is represented as:



Notice that if EAX contains this number, AX would be 0xA1A0, and AL would be 0xA0.

Note how this order only affects the bytes, not the bits.

x86 Assembly Instructions

The following templates are used for instructions:

```
instr ; instruction with no argument
instr arg ; instruction with one argument
instr arg1, arg2 ; instruction with two arguments
instr arg1, arg2, arg3 ; instruction with three arguments
```

For example, mov is an instruction to copy data from a source to a destination.

```
mov src, dest
```

x86 Assembly instructions are **MANY**, so here we are just going to cover the basics, not single instructions. Refer to Generating assembly from C code to get examples.

INTEL VS AT&T ASSEMBLY SYNTAX

Note that the order of the operands can change depending on the assembler.

```
mov dest, src ; Intel Syntax
mov src, dest ; AT&T assembly syntax (GNU assembly syntax)
```

AT&T Assembly Syntax:

- Compatibility with the GCC inline assembly syntax
- Register names are prefixed with % (e.g. %*eax*)
- Constants are prefixed with $
- As the example above, the source is on the left, the destination on the right

AT&T ASSEMBLY SYNTAX: OPERATION SUFFIXES

Instructions are generally suffixed with the letters ”b”, ”s”, ”w”, ”l”, ”q” or ”t” to determine what size operand is being manipulated.

- b = byte (8 bit).
- s = single (32-bit floating point).
- w = word (16 bit).
- l = long (32 bit integer or 64-bit floating point).
- q = quad (64 bit).
- t = ten bytes (80-bit floating point).

If the suffix is not specified, the operand size is inferred from the size of the destination register operand.

ASSEMBLER DIRECTIVES

Assembler directives begin with a period “.”

- These directives are not instructions
- `.byte`, `.word`, `.asciz` reserve some memory
- `.text` is used to mark the beginning of the code segment
- `.data` is used to mark the beginning of the data segment
- `.bss` holds zero-initialized data
- `.globl` defines a list of global symbols

MEMORY ADDRESSING

There are several ways addresses can be used (here we show them with the mov instruction but it can be done on other instructions).

```
mov $6, %eax          # copies the value 6 in EAX

mov %ebx, %eax        # copies the value in EBX in EAX

mov (%ebx), %eax      # copies 4 bytes from the memory
                      # address in EBX into EAX register

mov $message1, %eax   # Pointer to message1 is copied in EAX

message1:
    .asciz "Hello!"   # asciz puts a 0 byte at the end
```

REAL MODE

Real Mode is a simplistic 16-bit mode that is present on all x86 processors, all x86 processors begin execution in Real Mode for retro compatibility reasons.

- Modern operating systems run in Protected Mode, due to Real Mode limitations.
- Older operating systems (e.g. DOS) instead were running in Real Mode, since it was the only mode available at that time.
- Programs can access any memory address (no protections)
- 1MB memory
- Accessing more than 64kB requires to use segments

This is the mode you are going to use for your projects.

REAL MODE: SEGMENTS

Real Mode uses addresses as two values: segment and offset

- The address is written as `segment:offset` (e.g. `0x1:0x42`)
- A segment is 64kB
- The offset part is 16-bit
- The real address is $16 * segment + offset$
- Address range is 0*x*00000 to 0*xFFFFF*
- 1MB of data can be seen with a full 20-bit address

Note, the full address notation on AT&T syntax is:

```
segment:displacement(base register, index register, scale factor)
```

This is translated to:

```
16 * segment + base register + displacement + index register * scale factor
```

CALLS

Calling a function can be done using the instruction CALL

- Before calling the function all the parameters for the function are added to the stack.
- The CALL instruction pushes the address of the next instruction into the stack (i.e. the instruction to execute after the function returns).
- It modifies the instruction pointer %eip to the first instruction in the function.

INTERRUPTS

Interrupts are like calls to subroutines but initiated by peripheral hardware instead of `CALL`.

Interrupts are asynchronous and can occur anytime during the execution.

Interrupt handling is a better alternative than polling, which is periodically reviewing the status of peripherals.

Interrupts can be generated by software using the assembly instruction `INT X` where X is the number of the interrupts. For example, `INT 0x13`.

RETURNING FROM INTERRUPTS

At the end of the interrupt service subroutine, the instruction `RETFIE` (i.e. Return From Interrupt) is used.

After that, the program flow will be restored to where it was initially, recovering the return address previously stored on the stack (similar to `CALL`).

GENERATING ASSEMBLY FROM C CODE

Assembly can be generated from C source code. Let's take this simple Hello world as an example we save in a file called `hello.c`:

```
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

Now we compile it and we run:

```
$ gcc -o hello hello.c
$ ./hello
Hello, world!
```

GENERATING ASSEMBLY FROM C CODE

Now we want instead to generate the equivalent assembly code instead:

```
gcc -S -m32 -fno-asynchronous-unwind-tables hello.c
```

This creates a file ”hello.s”.

To compile to an executable instead from the assembly code and run:

```
$ gcc -o hello_2 -m32 hello.s
$ ./hello_2
Hello, world!
```

Try creating simple code and searching online for the corresponding assembly parts. This is a good way to learn! Also, this method can be used to have the compiler helping you out with assembly!

Questions?