x86 ASCII Dinosaur Game

A Preprint

Angel Aguayo
University of Arizona
aaguayo30@email.arizona.edu

Nicholas Kunzler
University of Arizona
nkkunzler@email.arizona.edu

April 14, 2020

Abstract

Here comes x86 ASCII Dinosaur Game. Get ready to jump, duck and do all the Dinosaur things.

1 Introduction

Using primarily x86, with a sprinkle of C's ncurses library, in order to make a 2D ASCII Dinosaur plat former. Who knows if it is possible, but it is the thought that counts.

2 History

Intel was founded by two American engineers in 1968 with 2.5 million dollars in funding. Before they developed the x86 architecture, they focused on developing DRAM memory chips. After a series of unsuccessful and successful chips and becoming a public company, Intel pivoted towards microprocessors, beginning with the 4004 in 1971, which was primarily used within Japanese calculators. Following up in 1972, Intel released the 8 bit CPU called the 8008, and eventually the 8080 2 years later. These both used a simplistic instruction set architecture (ISA), with 8 registers and 8-bit instructions. In 1978, Intel altered its ISA with the 8086 microprocessor, using 80x86, known commonly as x86. With 16-bit instructions and 1 megabyte of main memory, this is faster than any of the previous microprocessors. Development began in it in 1976, and it was originally meant as a side project as they were planning to jump to 32-bit with the 8800 they were working on. The technology at the time inhibited the 8800 from continuing, so they hired Stephen Morse, a software and electrical engineer who identified key design flaws with the 8800, to be the sole designer of the 8086. This was the first time they have hired a software engineer to design their microprocessors. With a new software oriented approach of what features to add to make the software more efficient instead of what features can be added in, Intel ended up revolutionizing the industry.

At launch, the 8086 gained only minor traction due to the Z80, their competitors chip, being in most business machines as the standard. Eventually it entered the market of embedded applications as NASA used it for controlling their diagnostic tests. After Morse left Intel in 1979, Intel released the 8088, which drew the attention of IBM, who chose the use the 8088 for their first mass produced personal computer (PC). The 8088 was backwards compatible, where it would send 16 bits out in 8 bit cycles, allowing the x86 ISA to be continued. IBM's PC used off the shelf parts, and as Intel's 8088 was 16-bit and could reduce chip count, IBM went with them. Over time, other companies started cloning IBMs PC, meaning the 8088 got more usage and grew more popular. Due to this popularity, Intel innovated upon the 8088, creating new and improved CPUs that ran based on 32-bit (and eventually 64-bit) instructions, maintaining the 86 suffix on the naming.

With a modern strategy of maintaining backwards compatibility and a need to make instructions faster, Intel ditched the numerical naming for more standard branding such as Pentium and Centrino. With the 8086 being the start of a trend of rapid developments and add ones to the x86 architecture, Intel's CPUs holds a presence today like never before. x86 is the basis of most computer architectures today, all due to IBM deciding to run with it for their PC.

3 Control Structures

x86 consists of one primary control-flow statement, jump. In addition, there are a variety of other instructions that grow from the jump instruction, such as loop and repeat instructions. There is a large variety of control-flow statements that are used for special cases in x86, but as they utilize the main control-flow statement of the jump instruction they will not be included.

There are two primary types of jumps in x86, conditional and unconditional. Conditional jumps are executed in a very similar way to that of BASIC's GOTO or C if statements, and unconditional, jumps that always execute. Unconditional jumps, which are represented in x86 with the instruction JMP, requires one label. A label is a string of text followed by a colon, usually used to represent a new segment of code, that allows the jump instruction to move control of the program from the current location to a new location within the program. Jumps are almost unrestricted in their movement throughout the program as they can traverse to locations earlier or later within the code.

Conditional jumps operate in a similar manner but require a flag and a variety of different jump instructions. The flag for the jump is set through the instruction compare, CMP. This instruction compares two values, either originating from registers or integer constants, and sets a flag value in the EFLAGS register according the result of the CMP instruction [1] from Appendix B. Once the flag has been set from the CMP instruction, there are multiple conditions that can be utilized to determine if a jump should happen. The following major jump instructions include:

JE	Jump if equal
JNE	Jump not equal,
JG	Jump if greater than
JGE	Jump if greater than or equal
JL	Jump less than
JLE	Jump less than equal
JS	Jump if negative
JNS	Jump if not negative

[NOTE] This list does not include all the possible jump instructions but represent instructions widely used, all jump types can be found on the official INTEL x86 User Manual, Volume 1. Sections 5-7 and 7-16 [1].

Depending on which jump is used and the result of the CMP, the program will either fall through, which is equivalent to the condition of an if statement being false, or jump to the corresponding label, equivalent to the statement being true. It is important to know that jumps are limited to how far they can jump. Both 64 and 32 bit versions of x86, are limited to a maximum of 32 bit offsets from the current execution line [1] Section 7-14 Header 7.3.8.1.

```
_start:
    ; Unconditional Jump
    JMP     _jmp_end

    ; Conditional Jump
    MOV     rax, 0xa
    CMP     rax, 0xa     ; Compare if 10 == 10
    JEQ     _jeq_end     ; Jump if result from compare is equal

_jmp_end:
    ...

_jeq_end:
    ...
```

Although the following instructions are not necessarily unique control-flow statements, as they are shorthand for the control-flow structures listed above, they are never the less useful. The first instruction is LOOP, which does exactly what is seems, iterates while a condition is true. The loop utlizes the RCX, for 64-bit, register as a counter register [1] Sections 7-16 and 7-17 Header 7.3.8.2. Additionally it utilizes a label in order to know where to jump after the loop body has finished executing. To determine when to exit the loop the following instructions can be used:

LOOPE	Loop while equal
LOOPNE	Loop while not equal
LOOPZ	Loop while zero
LOOPNZ	Jump while not zero

Depending on which LOOP instruction is used from the list above, a compare will happen between the RCX register, which defaults to decrements after each loop iteration. If the selected loop instruction determines that the loop has finished executing, the loop will fall through and continue to the line immediately following the LOOP instruction.

The same idea is applied to the string repeat instruction, or REP. However, instead of using the RCX register, the RSI and RDI registers are decremented or incremented to point to the next byte or word in the string [1] Section 7-18 Header 7.3.9.1]. To determine weather or not the registers are incremented or decremented, the EFLAGS will be set to 0 for incremented and 1 for decremented [1] Section 7-19 Header 7.3.9.2.

```
_start:
    MOV     rcx, 10     ; Run loop 10 times

_loop_head:
    ; Loop Body

    ...

    LOOP    _loop_head  ; Will loop as long as rcx >= 0
```

4 Data Types

x86 provides a multitude of fundamental data types upon which can be built on to develop much more sophisticated data structures. Both 32- bit and 64-bit x86 contain the ability to use bytes, words, and doublewords. A byte is allocated 8 bits, a word is allocated 16 bits (2 bytes), and a doubleword is allocated 32 bits (4 bytes). Once the IA-32 architecture was introduced, quadwords and double quadwords were added to the list of fundamental data types, being able to access 64 bits (8 bytes) and 128 bits (16 bytes) respectively [1] Section 4.1. In memory, words, double words, and quadwords do not need to be alligned on natural boundaries, but can instead be placed wherever, but to be efficient, one should aim to place these data types within even address, as well as addresses divisible by 4 and 8.

x86 serves as a weakly typed language, giving the programmer freedom to re assign registers and variables as needed. When assigning, x86 often requires a prefix before the register denoting what size is to be moved. For example:

```
    section .data:
        foo dq 100

    section .text:
        MOV al, byte foo
```

will take a quadword and move a byte of the quadword into the 8 bit register al. Note that this is taken from the least significant end of the variable.

Within each of the fundamental data types, x86 supports additional interpretations of these types, with the most common being numeric, pointer, bit field, and string data types. Numeric data types are integers as well as floating point numbers. Integers are represented as signed or unsigned values, with unsigned ranging from 0 to $2^n - 1$ for n bits, and signed ranging from -2^{n-1} to $2^{n-1} + 1$ for n bits [1] Section 4.2 Header 4.2.1.1. Floating point numbers are represented in single precision, double precision, and double-extended precision.

String data types in x86 make special use of bytes to function. By allocating each ASCII character a single byte (requires more if using UTF-8 formatting), strings are able to be represented as an array of bytes,words, or double words [1] Section 4.5. Statically, they can be allocated at compile time through the .data section

```
    section .data:
        hello   db   "Hello!"
```

If one wishes to allocate them dynamically, this can be achieved through an allocated uninitialized byte array in the .bss section. This can then be filled in throughout the program.

With these basic fundamental data types, composite types can be built. It is up to the programmer to declare more complicated data structures such as arrays and structs. Arrays in x86 can be declared much exactly like a string (a string simply being a byte array), and can be any fundamental data type. Using such arrays, it is possible to create 2D arrays, and build from that to develop other data structures.

5 Subprograms

Everything.

6 Summary

We are currently attempting to make an x86 ASCII Dinosaur Game, did you not read the paper?

7 Other

Only if you want.

References

[1] Intel Corporation. *Intel © 64 and IA-32 Architectures Software Developer's Manual.* Intel, 2019.