CSX64
Processor Specification

Contents

Purpose

The CSX64 virtual processor was designed primarily to be an educational tool for learning low-level programming. The processor acts as a 64-bit machine code interpreter with its own instruction set that includes hardware support for integral and floating-point computations. The instruction set was designed around the Intel x86_64 processor architecture to be as realistic and conforming as possible, while cutting out all of the hardware-dependent aspects of real machine code that can be cumbersome when trying to learn how it works.

The benefit of using the CSX64 virtual processor to learn assembly language (and especially machine language) is that its instruction set and system calls are not platform dependent: both the machine code programs and the interpreter itself are portable to any machine that supports the .NET framework. Additionally, CSX64 has a high degree of interoperability, being native to C# and thus compatible with all .NET languages, as well as being portable to C and C++ with minimal effort. Due to this, it's incredibly easy to create extensions for the CSX64 processor to handle just about anything, ranging anywhere from email messaging to video rendering.

Processor Details

General-Purpose Registers

Registers are small storage cells built directly into a processor that are vastly faster than main memory (RAM) but are also vastly more expensive per byte. Because of this price factor, there is not typically much room in a processor for storing data. However, due to their speed, they are the place where data is manipulated during program execution. The execution of a typical (non-trivial) program is: move data from memory to registers, perform computations, move processed data from registers to memory, repeat.

General-purpose registers are used for processing integral instructions (the most common type) and are under the complete control of the programmer (unlike many other registers in the processor that are at best indirectly modifiable). In x86_64 architectures (like CSX64), there are 16 64-bit general purpose registers.

Although you can access the full 64-bit contents of any general-purpose register at any time, it is often unnecessary or even undesirable to do so. For this reason, the low 32, 16, and 8 bits (and semi-low 8-bits of RAX, RBX, RCX, and RDX) of each 64-bit register can be used independently of the high bit portions, as depicted below:

Full 64 bits	Low 32	Low 16	High 8 of Low 16	Low 8
RAX	EAX	AX	AH	AL
RBX	EBX	BX	BH	BL
RCX	ECX	CX	CH	CL
RDX	EDX	DX	DH	DL
RSI	ESI	SI		SIL
RDI	EDI	DI		DIL
RBP	EBP	BP		BPL
RSP	ESP	SP		SPL
R8	R8D	R8W		R8B
R9	R9D	R9W		R9B
R10	R10D	R10W		R10B
R11	R11D	R11W		R11B
R12	R12D	R12W		R12B
R13	R13D	R13W		R13B
R14	R14D	R14W		R14B
R15	R15D	R15W		R15B

It should be noted that these smaller subdivisions of the registers are still in fact just subdivisions of the same physical register. If you modify one subdivision, the contents of all the subdivisions of that particular register will see the change. This is a common pitfall for beginning to learn low-level programming, but it is easily avoided by being aware of it.

Perhaps now you're wondering about the strange naming conventions for the first 8 registers compared to the last 8. Back in the ye olden days, there were only 8 (32-bit) registers, and they each had specific purposes:

Register	Name	Purpose
EAX	Accumulator Register	Used for accumulating arithmetic results
EBX	Base Register	Base pointer in memory accesses
ECX	Counter Register	Holds loop counters
EDX	Data Register	Used in data access and some arithmetic
ESI	Source Index Register	Index in a source array
EDI	Destination Index Register	Index in a destination array
EBP	Stack Base Pointer	Pointer to stack base
ESP	Stack Pointer	Pointer to top of stack

Of course, registers eventually became “general-purpose”, which means they didn’t necessarily have to be used for these things. Nevertheless, the names stuck. When 64-bit architectures came around, they added another 8 registers, but gave them uniform names.

The Stack

For all non-trivial programs, you’re going to want to be able to have a notion of function-calling. This means the program needs some place to store a return address, jump to the top of the function, then eventually jump to the return address at some point. As you should already know, a stack structure is used to tackle this problem.

Up till now, it’s unlikely you’ve had to deal with the stack directly. In most languages (even low-level ones) the stack is completely hidden from the programmer. In these languages we can only indirectly impact it. We already know that declaring a variable sets aside space on the stack, and that calling a function uses the stack as well. The difference now is that in assembly language and machine code, the programmer is responsible for managing the stack. In fact, if you don’t want a stack in your program, you can just as easily obliterate it. It’s all just numbers. The processor doesn’t care. That said, the processor will also not try to stop you from irreparably damaging the stack structure that is set up for you by default, so if you want a stack in your program (which any sane person should), take careful note of the following:

In the standard Intel x86_64 architecture, the stack is managed by RBP and RSP (base pointer and stack pointer – more on this in Calling Conventions). Upon program initialization, RBP and RSP are set to the address of the top of the stack, which ***begins at the high side*** of the program’s available memory and ***grows downward***. Because of this, RSP will always point to the most-recently-added item on the stack.

To add an item to the stack you can use the PUSH instruction. To remove an item, you can use the POP instruction. To call a function you can use the CALL instruction, which pushes the (64-bit) return address onto the stack and then jumps to the function. To return from a function you can use the RET operation, which pops a 64-bit address off the stack and jumps to it. All of these instructions assume that RSP points to the top of the stack. If this is not the case (e.g. you have destroyed the stack by “clobbering” – and not restoring – RSP), then you should not use these instructions at all.

RSP may be also be modified directly without damaging the stack structure, but care should be taken when doing so. For instance, if you want to set aside 128 uninitialized bytes on the stack, rather than pushing garbage data one by one until you got 128 bytes, you could simply ***decrement*** RSP by 128.

Floating-Point Unit (FPU)

The FPU is a coprocessor that is built into a CPU to facilitate hardware-support for floating-point arithmetic. Being a coprocessor, the FPU is a bit tricky to deal with at first, as it has its own set of flags, registers, and an entirely different structure from the usual CPU instructions.

FPU Registers

The FPU has 8 80-bit registers named ST0-ST7, though some assembly languages refer to them as ST(0)-ST(7). These registers are 80-bit to reduce rounding errors from floating-point computations involving the usual 32-bit (“single precision”) and 64-bit (“double precision”) floating-point formats, but can also be used to enable 80-bit (“extended precision”) computations. Conveniently, this representation has 64 significand bits, meaning an FPU register can losslessly store any 64-bit integer (so you could technically use the FPU for integral operations as well if you really wanted to).

The trickiest part about the FPU is that the ST0-ST7 registers are set up as a sort of stack, where ST0 refers to the top of the stack and increasing ST numbers travel down the stack. The FPU internally keeps track of a 3-bit flag representing which physical register is referred to by the name ST0.

When you refer to an FPU register by its ST number, the FPU (internally) adds this 3-bit index of the top of the stack to the number you specified, and then takes the remainder of division by 8 (or equivalently bitwise AND with 7) to get the index of the physical register to perform the operation on.

Although the FPU registers are accessed in this weird stack system, you may still refer to any FPU register at any time. However, the FPU also keeps track of the state of its registers. For instance, it knows which registers have meaningful data in them. You may only load a value into an “empty” register, and you may only use “non-empty” registers in floating-point calculations.

When you push a value onto the FPU stack the FPU first decrements the top of stack index by 1 (wrapping from 0 to 7), puts the value there, and marks that register as non-empty.

When you pop a value off the FPU stack the FPU first gets the value at the top of the stack, marks that register as empty, and finally increments the top of stack index by 1 (wrapping from 7 to 0).

Because of this, it is also possible to overflow or underflow the FPU stack, which are given their own error codes in CSX64.

Because pushing or popping a value changes which ST numbers refer to what, you should take extra care to document your FPU code well.

Most operations require an operand be in ST0. If this is not the case, you may use various instructions such as FST, FLD, FMOVcc, and FXCH to reorganize the stack.

More information on the FPU can be found here (though CSX64 may not support all of these instructions): http://www.website.masmforum.com/tutorials/fptute/index.html.

Flags Register

The processor itself needs to keep track of key pieces of status information. Some of this information is held in a special "flags" register, which holds many bitfields representing different processor states. The CSX64 flags register conforms to the Intel x86_64 standard but uses several reserved flags in the high 32 bits for its own special purposes. The Flags are enumerated in order in the table below:

FLAGS			
Bit	**Mask**	**Abbreviation**	**Description**
0	0x0001	CF	Carry Flag
1	0x0002		Reserved (initialized to 1)
2	0x0004	PF	Parity Flag
3	0x0008		Reserved
4	0x0010	AF	Auxiliary Carry/Adjust Flag
5	0x0020		Reserved
6	0x0040	ZF	Zero Flag
7	0x0080	SF	Sign Flag
8	0x0100	TF	Trap Flag
9	0x0200	IF	Interrupt Enable Flag
10	0x0400	DF	Direction Flag
11	0x0800	OF	Overflow Flag
12-13	0x3000	IOPL	I/O Privilege Level
14	0x4000	NT	Nested Task
15	0x8000		Reserved
EFLAGS			
Bit	**Mask**	**Abbreviation**	**Description**
16	0x0001 0000	RF	Resume Flag
17	0x0002 0000	VM	Virtual 8086 Mode
18	0x0004 0000	AC	Alignment Check
19	0x0008 0000	VIF	Virtual Interrupt
20	0x0010 0000	VIP	Virtual Interrupt Pending
21	0x0020 0000	ID	Enables CPUID instruction
22-31	…		Reserved
RFLAGS			
Bit	**Mask**	**Abbreviation**	**Description**
32	0x0000 0001 0000 0000	FSF	File System Flag
33-63	…		

Just like the general-purpose registers, the 64-bit flags register (RFLAGS) is partitioned twice into the low 32 bits (EFLAGS) and the low 16 bits (FLAGS). Unlike general-purpose registers, there is no 8-bit segment.

The flags in orange and green are reserved. While some systems allow reading/writing reserved flags, it may be the case that at some point in the future that same bit position will be given a meaning. This would thus potentially break any code built around the assumption that they would not be modified by anything except your code. Additionally, some systems don't allow you to modify these flags at all (e.g. CSX64). In short: do not base your code on reserved flags.

The green flags are put to special use by CSX64. Just like the orange flags, executing client code cannot modify green reserved flags by any means. The green flags specifically are meant to be used as restraints on client code that are provided by the virtual operating system.

Error Codes

During program execution, the processor may encounter an error. When this happens in CSX64, it will set an error code and immediately halt execution. Reviewing the error code emitted and any recent modifications to your program may help you diagnose the cause and fix the bug.

With careful coding, all of these errors can be trivially avoided.

Error Code	Value	Description
None	0	No error has occurred.
OutOfBounds	1	A memory access operation has exceeded the amount of memory allocated to the process.
UnhandledSyscall	2	A system call was not successfully resolved.
UndefinedBehavior	3	An instruction or system call resulted in a state that is not well-defined.
ArithmeticError	4	An arithmetic operation failed (e.g. divide by zero).
Abort	5	Program was aborted by external code.
IOFailure	6	An error occurred during a file system operation.
FSDisabled	7	A system call requiring file system access was made without FSF being set.
AccessViolation	8	An instruction or system call was requested to modify data without permission.
InsufficnentFDs	9	Attempted to open a file without any available file descriptors.
FDNotInUse	10	Attempted to perform an IO operation on a file descriptor that was not in use.
NotImplemented	11	An instruction was used that has not been implemented.
StackOverflow	12	A stack operation went outside the stack segment.
FPUStackOverflow	13	Attempt to push a value onto a full FPU register stack.
FPUStackUnderflow	14	Attempt to pop a value from an empty FPU register stack.
FPUError	15	An error occurred during and FPU computation.
FPUAccessViolation	16	Attempt to use an FPU register that doesn't have a value.
AlignmentViolation	17	Attempt to read/write memory that is not properly aligned.

Main Memory

Because the registers in the processor are much too small to hold any real amount of information, a larger storage base is required for any non-trivial program. Main memory (RAM) serves as this data reservoir and is where the vast majority of your data will be held. Fortunately, coming from mid to high-level languages has prepared you well for this. You should already be quite familiar with the ins and outs of using and navigating memory, including the concepts of addressing and pointers. In the world of assembly language, your knowledge of addressing and pointer arithmetic will make or break you as a programmer.

In CSX64, upon program initialization, a section of memory is allocated to your program by the virtual operating system, enough to contain the contents of the program as well as an amount of additional space for other things such as the stack. The program's contents are then loaded into this memory starting at address zero.

On any reasonable modern system, data is addressed in 8-bit words, otherwise known as bytes. These 8-bit words cannot be subdivided further (i.e. you cannot take the address of anything smaller), though individual bits may be modified by bitwise operations, as per usual.

Endianness

In order to store values that are larger than 1 byte, we must use several (contiguous) bytes. However, this introduces a problem: how do we arrange the data in the individual bytes we have available? Of course, there are many ways to do this (n! where n is the number of bytes), but there are really only two methods that are sensible: put the most significant bytes first (big-endian) or last (little-endian). There are also other byte orderings that are used in the real world, but they've died off for the most part (in fact even big-endian is virtually dead).

Let's conceptualize this with base 10 numbers:

Memory				**Value**	
Low Address			**High Address**	**Little-Endian**	**Big-Endian**
4	1	9	0	914	4190
2	4	6	3	3642	2463
7	7	0	5	5077	7705
0	0	1	8	8100	18

You may be wondering why anyone in their right mind would pick little-endian over big-endian, as little-endian essentially reads the numbers backwards. You would, in fact, have a good argument. However, you're looking at the data as a human. While big-endian byte ordering is more readable, little-endian has performance benefits:

One of the most compelling reasons to use little-endian systems is that it makes accessing the low-order bits of a larger numeric type easier, as you don't have to first offset the address before fetching the value. If given the address of a 64-bit integer, that same address can also be used to get the value truncated down to its low-order 32-bits. This sort of thing comes up in low-level programming more often than you'd think, and since high-level programming is basically just a syntactic sugar wrapper for low-level programming, it helps everyone out in the end.

Numeric Types

Integer

The majority of processor operations involve the manipulation of integers of various widths. In fact, the integer is really the core of all modern computing, as all integers can be stored perfectly (i.e. no rounding) as a finite series of binary digits, which is perfect for a computer, as that's all it's available to anyway.

The computer is a pure logic machine. It runs entirely on true or false. There is no in between. The reason behind this lies in manufacturing tolerances. At the heart of every processor is an intricate structure of electrical components. The simplest of such components for our purposes are logic gates (e.g. *and*, *or*, *xor*, *not*), which take in one or more electrical inputs and produce one or more electrical outputs based on the voltages. For instance, an *and* gate outputs high voltage if all of its input voltages are high, and outputs low voltage otherwise.

So, what does it mean for the voltage to be high or low? Well, this is where the natural imperfection of the universe comes into play: there's no strict definition. You could take two different logic gates, run the same voltages through all of the inputs, and ultimately get different results. This is because the gates are not perfectly identical: one might have slightly more resistance on its output or on one or more inputs due to flaws in the material and manufacturing.

Because of this, designing a circuit to differentiate similar voltages unavoidably results in it not being guaranteed to be repeatable (as well as vastly increasing the manufacturing cost due to the stricter manufactory specifications that would be required to pull it off). And so, we take the extremes and use binary: all or nothing – very easy to distinguish, very reliable.

For information on how to convert integers to and from binary, see the examples page on binary integers.

Unsigned Integers

Unsigned integers have no sign, hence the name. This is the simplest type of binary integer, as it's exactly what you would expect: pure binary numbers. No tricks. Because they are unsigned, however, they can only range 0, 1, 2, …, meaning they're mainly good for dealing with raw binary data or values you know absolutely cannot be negative (e.g. size of an array, temperature in kelvin, your age, etc.)

Whenever this criterium is met, it is sometimes better to use unsigned rather than signed values due to their simplicity and higher range, which will be elaborated on in the signed integer section. However, care should be taken when mixing signed and unsigned numbers, as will be explained.

Signed Integers

Unlike unsigned integers, signed integers can be used to represent negative numbers as well. In virtually anything made since the 1960's, this is done via an encoding scheme known as 2's complement.

In 2's complement, the highest bit (sign bit) is used to represent the sign of the number (0 for positive, 1 for negative). However, taking the negative of a binary number under 2's complement is not as simple as just flipping the sign bit (and it's good this is not the case, as you're about to find out).

To take the negative of a signed integer, we apply the algorithm `~v + 1` where `v` is the value to negate and `~` is the bitwise not). Because of the bitwise not, the sign bit is also flipped. As you can see, this also means `-0 = 0`, which is a good thing – this would not be the case had we simply flipped the sign bit.

Rather conveniently (but really by definition), 2's complement results in a beautiful encoding scheme in which addition and subtraction are exactly the same process regardless of signage (i.e. signed or unsigned). Moreover, the low half of multiplication is also the same, regardless of signage, though the high half of the product (which is generally ignored in all but assembly/machine code) does differ for signed and unsigned values. And lastly, conveniently (but also really by definition), a positive signed number (zero sign bit) is identical to its unsigned counterpart, though the opposite is only true for unsigned values with a zero high bit.

Of course, using a bit to represent the "sign" also means we have 1 less bit to play with, meaning the largest magnitude of a signed integer is half the largest magnitude of an unsigned integer.

And, as a consequence of not having a positive and negative zero, there is one more negative number than positive numbers, meaning there is always exactly one non-zero value for any width of signed integer whose negative is itself. This value is always the largest-magnitude negative value (e.g. for 8-bit signed integers: -128 is a valid negative number, but its negative is still -128, as 128 is not a valid positive number due to the sign bit being set).

Floating-Point

Integers are very simple, both to understand and to convert from "human script" (base 10) to binary. However, they are very limited numerically (e.g. `1 / 2 = 0`). It would be nice to have fractional numbers as well, but this also has complications.

If we used a fixed-width integer to represent a binary number extending into negative powers of 2 (just as decimal does where 0.23 is really $2*10^{-1} + 3*10^{-2}$), we would have a structure that would either not have many decimal places or would have a maximum value much smaller than its same-width integer would.

If we used a type that could expand or contract as needed to store the integral and decimal portions it would be very big (not to mention it would need to be allocated dynamically), and thus would likely not fit in a register for processing. Because of this, operations involving it would be highly entangled with main memory, which would slow things down tremendously. And even then, what about numbers that repeat infinitely? Where should we cut them off?

Conveniently, scientists have given us the answer with their inspirationally-titled *scientific notation*. If this entry in the CSX64 manual hasn't phased you yet, you're well accustomed to scientific notation already. The idea is to use a fixed-width binary number with negative powers of 2 (as discussed above), but also multiply it by a power of 2, which solves the issue of it being limited in range compared to an integer of equal width. Another key aspect of scientific notation is that it is of the form $a.etc*10^b$, where `a` is always in the range [1, 10). To scale the number around to other magnitudes, we simply change `b`, hence the name floating-point. Along with this comes the concept of significant digits (i.e. regardless of the value, only a certain number of digits will be stored, and the rest are discarded as zeroes), which is perfect for our uses, as we simple can't efficiently pull off a variable number of significant digits.

And so, we come to the big question: how do we encode these supposed floating-point monstrosities?

64-bit – Double Precision

The CSX64 processor contains hardware support for handling 64-bit (“double precision”) IEEE 754 floating-point values. There are plenty of utilities that will convert decimal numbers into their respective floating-point representations, but it’s occasionally beneficial to understand the underlying encoding scheme of the structure (e.g. say you wanted to extract what the power of 2 is without the overhead of taking its absolute value, then its logarithm, then casting it to an integer via truncation, as well as domain shifting due to not being able to take the logarithm of zero). If you understood its structure, it would simply be a right shift, a bitwise and, and a subtraction, all of which are vastly faster than logarithms or potential function calls.

Here’s a graphical representation of the data structure:

High bits **Low bits**

1-bit Sign	**11-bit Exponent**	**52-bit Fraction**

Sign – A single bit to represent the sign of the number (i.e. 0 for positive, 1 for negative).

Exponent – An 11-bit exponent encoded in excess-1023 (i.e. the “real” power + 1023). This 1023 value is known as the bias (b) of the representation and is different for different widths of floating-point representations (and in fact is always $2^{n-1} – 1$ where n is the number of bits used to encode the exponent).

Fraction – A 52-bit fractional portion that omits the leading 1 (and is therefore technically 53 bits for the purpose of calculating precision) (e.g. 1.00101101… would be stored as 00101101…).

Because the fraction portion is ordinarily assumed to have a hidden 1 in the front (which is not stored in the binary representation), it would be impossible to represent zero. Because of this, having an exponent field of zero denotes a denormalized value, where there is not assumed to be a hidden 1. This also means that the true exponent is interpreted as `−b+1` to account for this sudden loss of a high order significant digit (rather than the `−b` it would be under normalized interpretation: `e−b`), thus closing the gap between de/normalized values.

There is also a concept of infinity, which is represented as an exponent field of all 1’s and a fraction field of all zeros. The sign bit can be changed to represent positive and negative infinity.

Additionally, there is a concept of invalid arithmetic: NaN (Not a Number). These come in two varieties: QNaN (quiet), and SNaN (signaling). QNaN values are errorlessly produced by some floating-point operations (e.g. dividing zero by zero). The result of any arithmetic involving QNaN will also result in QNaN, thus safely propagating through a series of operations. SNaN however, will cause the floating-point module in a modern processor to emit an error when used in any operations (though this is usually “quietened” to a QNaN, thus propagating through arithmetic and not halting program execution).

A summary of 64-bit floating-point values is as follows, where x denotes either a 1 or a 0:

Floating Point Values			
Sign	**Exponent (e)**	**Fraction (f)**	**Value**
0	00…00	00…00	+0
0	00…00	00…01 ⋮ 11…11	Positive Denormalized Real $0.f*2^{(-b+1)}$
0	00…01 ⋮ 11…10	xx…xx	Positive Normalized Real $1.f*2^{(e-b)}$
0	11…11	00…00	+∞
0	11…11	00…01 ⋮ 01…11	SNaN
0	11…11	1x…xx	QNaN
1	11…11	1x…xx	QNaN
1	11…11	00…01 ⋮ 01…11	SNaN
1	11…11	00…00	−∞
1	00…01 ⋮ 11…10	xx…xx	Positive Normalized Real $-1.f*2^{(e-b)}$
1	00…00	00…01 ⋮ 11…11	Positive Denormalized Real $-0.f*2^{(-b+1)}$
1	00…00	00…00	-0

The positive and negative zero values can both be used to represent zero, but fortunately will compare equal under floating-point comparison. They can be thought of as special cases of denormalized real values (where the fractional portion is zero).

There are several well-defined special cases for the result of floating point operations, some of which are listed below. In general, if the result of an operation has a conceptual value, it will result in that value (e.g. adding infinities of the same sign will result in another infinity with the same sign). If it does not produce a logical result (e.g. adding infinities of opposite signs), it will result in QNaN.

Operation	Result	Operation	Result
±finite / ±∞	±0	±∞ / ±∞	QNaN
±nonzero / ±0	±∞	±0 / ±0	
±nonzero * ±∞		±0 * ±∞	
±∞ + ±∞		±∞ − ±∞	

For more information about how to encode floating-point values, refer to Encoding IEEE 754 64-bit Floating-Point Values in the Appendix.

For more information on the structure of the IEEE 754 format, refer to:

http://steve.hollasch.net/cgindex/coding/ieeefloat.html.

32-bit – Single Precision

CSX64 also supports IEEE 754 32-bit ("single precision") floating-point computations. This format is identical to the above explained 64-bit format, except that there are 23 bits to encode the fractional portion (which still excludes the leading 1) and 8 bits to record the exponent (meaning the exponent bias is 127 instead of 1023).

Here's a graphical representation of the data structure:

High bits		**Low bits**
1-bit Sign	**8-bit Exponent**	**23-bit Fraction**

All other special properties (e.g. infinity, NaN, denormalization) function identically to its 64-bit counterpart, and have the same formats as shown in the table above.

In the past, 32-bit floating point was used because it was faster than 64-bit, which made it the obvious choice for anything that didn't need the extra precision or range. Nowadays however, the speed difference is negligible (in fact, 64-bit is sometimes actually faster than 32-bit, depending on the processor). The main reason to use 32-bit in modern times is just to save space. If you need an array of one million floating point values and you don't need high precision, you might as well use 32-bit and save four million bytes. This also has the effect of making code smaller, meaning more can fit into a cache line, thus reducing cache misses and ultimately speeding up execution.

Machine Code

As discussed elsewhere, a processor is essentially just a logic machine. Values are encoded in a binary, put through some logical manipulation algorithm (which may be incredibly complex), and ultimately transformed into some meaningful result. But how does the processor know how to execute a program? Well, just like numbers, programs themselves must somehow be encoded into binary. This is what's known as machine code, and it lies at the heart of every computer. There is nothing closer to the underlying hardware than machine code.

Machine code varies from processor to processor, which makes it very difficult to program in (to the point that absolutely no one does it). It is, however, an excellent academic exercise. Fortunately, The CSX64 virtual processor is cross-platform, meaning any program you *were to* write in CSX64 machine code will be able to run on any machine that can run the interpreter.

Machine code is ***very*** technical (in fact, ***maximum*** technical), and most instructions require certain key pieces of information to be in specific bit fields of the binary code. Below is an overview of the standard formats used by CSX64 binary.

Register Format

Nearly all instructions involve (or can involve) at least one of the general-purpose registers. As there are 16 of such registers, they are designated by a 4-bit register id, where the registers are enumerated 0-15 in the order they're listed in the General-Purpose Registers section. Any instruction format specification denoting a source or destination register (or abbreviated "reg", "r", "r1", "r2", etc.) uses this format unless otherwise specified.

Size Format

Most instructions require a width of data to work on. These data widths are designated by a 2-bit size code. Any operation format specification denoting size uses this format unless otherwise specified.

Size Code	Size	Size Name
0	8 bits	byte
1	16 bits	word
2	32 bits	dword
3	64 bits	qword

Multiplier Format

Some operations may specify one or more size multipliers (most notably memory addressing operations). These multipliers are designated by a 3-bit multiplier code. Any operation format specification denoting a multiplier (or abbreviated "mult", "m", "m1", "m2", etc.) uses this format unless otherwise specified.

Multiplier Code	Multiplier Value	Multiplier Code	Multiplier Value
0	0	4	8
1	1	5	16
2	2	6	32
3	4	7	64

Reading CSX64 Machine Code Specifications

Because machine code requires certain bits in certain places to work properly, we need a standardized way of expressing this format. This manual uses the following format:

```
[1: item1][6: item2][1: item3]   ([5: v1][1:][2: v2])   ([16: val])
```

Bracketed item – a labeled collection of bits. The label is used elsewhere in the specification page to explain what putting a value there will do. The number to the left of the label specifies the number of bits in the bitfield. If the bitfield has no name, it is padding, and the value does not matter (though you will likely prefer it be filled with zeroes to not give the appearance of meaning where there is none). Bracketed items are collected into words with spacing between one another. The location of a bracketed item in a word determines the position of the bitfield in the word, with the left being the high bits of the word. Words follow the usual rules of endianness, but of course this is only an issue for multi-byte words.

Any parenthesized word may or may not be expected to be there, depending on the other values provided in the binary format. If this is the case, the specific instruction's machine code specification will detail when it should or should not be provided.

Memory Address Format

In CSX64, memory addresses as operands to an instruction are standardized and take the following form:

```
[1: base][3: m1][1: -m2][3: m2]   ([4: r1][4: r2])   ([64: imm])
```

If `base` is 1, `imm` is assumed to be provided. If `base` is 0, `imm` is 0 for the address calculation.

If `m1` or `m2` is nonzero, `r1` and `r2` are assumed to be provided.

If `−m2` is `1`, `r2` is first negated before being used in the address computation (but `r2` is not modified).

The resulting address is thus `imm + m1*r1 + m2*r2`, where m1 and m2 are expanded to their multcode values and r1 and r2 are 64-bit register values.

Conditions

At the hardware level, conditionals are processed based on the value of a special flags register. These flags are modified in different ways by many operations and are used to perform conditional operations such as if statements. These conditions are listed below:

Label	Alias	Name	Signage	Condition
Z	E	Zero / Equal	N/A	`ZF = 1`
NZ	NE	Not Zero / Not Equal		`ZF = 0`
P	PE	Parity / Parity Even		`PF = 1`
NP	PO	Not Parity / Parity Odd		`PF = 0`
O		Overflow		`OF = 1`
NO		Not Overflow		`OF = 0`
C		Carry		`CF = 1`
NC		Not Carry		`CF = 0`

Label	Alias	Name	Signage	Condition
S		Sign		`SF = 1`
NS		Not Sign		`SF = 0`
B	NAE	Below	Unsigned	`CF = 1`
BE	NA	Below or Equal		`CF = 1 or ZF = 1`
A	NBE	Above		`CF = 0 and ZF = 0`
AE	NB	Above or Equal		`CF = 0`
L	NGE	Less	Unsigned	`SF != OF`
LE	NG	Less or Equal		`ZF = 1 or SF != OF`
G	NLE	Greater		`ZF = 0 and SF = OF`
GE	NL	Greater or Equal		`SF = OF`

Because not all instructions affect the flags in the same way, it is necessary to reference the documentation for any operation before using its resulting flags. If an instruction states that a particular flag is undefined, it should not be used after that instruction until it is modified in a meaningful, definitive way. If a flag is not mentioned, it is not modified.

Translating Machine Code

Translating your program's intent into machine code is an easy (albeit ludicrously tedious) task. All you have to do is follow the binary format the specification indicates, putting the values in the correct positions.

The first thing we need to think about is how a computer knows what it's supposed to do and when. This is accomplished by OP codes, which are numbers that a computer is hardwired (not really, nowadays, but you get the idea) to see and perform a specific action. In a standard Intel x86_64 processor there is a register named RIP (64-bit instruction pointer) that holds a pointer to the current instruction being executed. When the processor goes to execute an instruction, it looks at the byte(s) at that location and acts accordingly.

However, this isn't enough information. For all but the simplest instructions the processor will need to know more information to perform the action, such as locations in memory to load or store values. An example of this was shown in the section on CSX64 memory addresses. In these cases, we need to follow the specific operation's binary format to the letter.

For example, to add 17 to the 32-bit register partition ECX, we would first look up the ADD instruction entry and get its OP code (which is provided in hex) and fill in values for its binary specification. As a side note, some instructions have OP codes exceeding 8-bits (e.g. Jcc). In these cases, the order of the individual bytes of the OP code are how they should be arranged (i.e. do not try to account for endianness). For our example, this gives us the following binary form (in hexadecimal):

```
12 24 10 11 00 00 00
```

Remember that we need to account for endianness in multi-byte values (except multi-byte OP codes). Since the vast majority of computers today are little-endian, this example uses little-endian encoding, meaning the low order bytes of 17 come first and make the value look huge when it actually isn't.

Now, say we wanted to compare the result to an element of an array (of 32-bit integers) in memory that starts at address 200. Let's also say that we want the index of the item in the array to be the current value of RDI. We thus need to use CMP's binary specification, which in turn uses the address specification:

Which gives us the binary form:

```
27 24 20 b0 50 c8 00 00 00 00 00 00 00
```

Now let's say we want to jump to address 1000 if the values were equal (i.e. result was zero – equal condition is an alias for zero condition). We thus use Jz's binary specification, which gives us the binary form:

```
0b 00 0a e8 03 00 00
```

Note that in this case we chose to use the 32-bit version of Jz, which is an option if you know your program segment is going to fit within a 32-bit address space. And, if you really wanted to, you could also use the 16-bit version in this case. Care should be taken when making these judgments: while the file you put this in might exist only in a small enough address range for this to work, when other files are merged together this may no longer be the case. In general, 32 and 64-bit addresses are the way to go. By default, CSX64 assembly will take the 64-bit route, which is wasteful but safe.

Congratulations: you've just gone through the process of translating your first few lines of machine code! Now, putting it all together, we add 17 to the 32-bit segment of R3, compare it to a value from an array in memory with base address 200 and index in R7, then jump somewhere if they were equal. The full resulting machine code is thus:

```
12 24 10 11 00 00 00

27 24 20 b0 50 c8 00 00 00 00 00 00 00

0b 00 0a e8 03 00 00
```

Which is *essentially* the same as the following in C:

```
R2 += 17;

if (R2 == arr[R5]) goto somewhere;
```

Assembly Language

As you can imagine, writing programs in machine code is completely impractical. It requires you to reference the binary specification for every single operation you want to perform, and makes **you** do the grunt work of translating everything into binary and putting things in the proper bit fields (not to mention making you painfully aware of endianness). For all of those reasons and many, many more, programs are never written in machine code except for academic purposes. Assembly language serves as the bridge between the programmer and pure machine code and is the closest programming "language" (i.e. human-readable instructions) to the underlying hardware.

Assembly languages are tailored to specific processors (though most assembly languages for different processors still contain many instructions that go by the same name and have the same syntax to promote code reusability). In essence, assembly language is just a human-readable form of machine code that is translated directly into machine language by an assembler.

As an example of what assembly language might look like, the following is the CSX64 assembly language equivalent of the example in the above section on translating machine code:

```
add ecx, 17

cmp ecx, [200 + 4*rdi]

jz 1000
```

As you can see, we specified the same values for each instruction that we translated before. The difference is that this is readable – you can look at this and know what it's doing. That's the only purpose of assembly language.

CSX64 Assembly Language

CSX64 comes with a thorough, built-in assembler that uses the Intel syntax (the other big player in assembly is the AT&T syntax).

Before we describe the syntax of operations and assembler utilities, we need to go over some of the basics of CSX64 assembly language. Being first and foremost a realistic assembly experience, much of this holds for all assembly languages.

Segments / Sections

In a typical assembly language, your program is broken up into several segments/sections. This is done because operating systems themselves require executables to be split up into several segments. This is in part done for efficiency in memory access by having related pieces of binary data be contiguous (e.g. all executable code is together, and variables are all together elsewhere), which ultimately speeds up execution by a sizeable amount as a consequence of how the memory circuits are designed and connected to the CPU (e.g. fewer cache misses). Additionally, operating systems typically instruct the processor to enforce certain read/write/execute privileges for certain ranges of addresses. Here we'll explore the most common segments, which are all supported in CSX64 assembly.

Text Segment (.text)
The text segment holds all of your executable code and will typically dwarf the other segments in terms of size. The text segment is read-only and is the only segment that is executable (attempting to write to the text segment at runtime or execute anything outside it results in an AccessViolation error).

Data Segment (.data)
The data segment holds all your static duration (i.e. not on the stack) variables that are initialized to specific values. This will typically be used only for global variables. The data segment is read-write.

To write to the data segment you must use the Declare directive.

It is an assemble-time error to put executable instructions in the data segment.

Read-Only Data Segment (.rodata)
The rodata segment is like the data segment except that it is read-only (attempting to write to the rodata segment at runtime results in an AccessViolation error).

BSS Segment (.bss)
The BSS segment (Block Started by Symbol) is a segment that has no contents in terms of data or instructions. It consists only of a number that represents its length that the operating system then expands upon program initialization to a zero-filled, contiguous block of memory with said length (hence the name). This is used when you would ordinarily put something in the data segment, but you don't care about initializing it to a specific value (e.g. an IO buffer array).

To add to the BSS segment, you must use the Reserve directive (it is an assembly-time error to attempt to write anything to the BSS segment).

imm – Immediate
An imm, otherwise known as an immediate, is a value that is placed directly inside an operation's binary specification. In assembly language this is a literal number, such as 0, 7, or 3.14. These serve the purpose of providing values to fill in the bit fields of an operation's binary specification, as we did above.

However, an imm doesn't have to just be a pure number, it may be an expression (e.g. `(1 + 3) * 8.3`). It may even involve symbolic names: (e.g. `(a + b) / 2`). In the most general sense, an imm is anything that doesn't come from a register or memory.

The following table lists all the imm operators available in CSX64, in order of decreasing precedence, where grouped operators have equal precedence:

Usage	Name	Result	Result Type	Evaluation Order
`+A`	Identity	`A`	Type of A	Right-to-Left
`-A`	Negative	`-A`		
`~A`	Bitwise Not	`~A`		
`!A`	Logical Not	`1 if A is zero, otherwise 0`	Integral	
`*A`	To Floating	`A as floating`	Floating	
`/A`	To Integral	`A as integral via truncation`	Integral	
`A * B`	Multiply	A * B	Floating if A or B is floating, otherwise integral	Left-to-Right
`A / B`	Divide	A / B		
`A % B`	Mod	Remainder of A / B		
`A + B`	Add	A + B	Floating if A or B is floating, otherwise integral	Left-to-Right
`A - B`	Subtract	A - B		
`A << B`	Shift Left	A << B	Floating if A or B is floating, otherwise integral	Left-to-Right
`A >> B`	Shift Right	A >> B		
`A < B`	Less	1 if condition met, otherwise 0	Integral	Left-to-Right
`A <= B`	Less or Equal			
`A > B`	Great			
`A >= B`	Great or Equal			
`A == B`	Equal	1 if condition met, otherwise 0	Integral	Left-to-Right
`A != B`	Not Equal			
`A & B`	Bitwise And	A & B	Floating if A or B is floating, otherwise integral	Left-to-Right
`A ^ B`	Bitwise Xor	A ^ B	Floating if A or B is floating, otherwise integral	Left-to-Right
`A | B`	Bitwise Or	A | B	Floating if A or B is floating, otherwise integral	Left-to-Right
`A && B`	Logical And	1 if A and B are nonzero, otherwise 0	Integral	Left-to-Right
`A || B`	Logical Or	1 if A or B is nonzero, otherwise 0	Integral	Left-to-Right
`A ?? B`	Null-Coalescing	A if A is nonzero, otherwise B	Type of the branch taken	Left-to-Right
`A ? B : C`	Ternary Conditional	B if A is nonzero, otherwise C	Type of the branch taken	Right-to-Left

These operators may be chained in any way desired, with parenthesis being available for grouping if needed.

Furthermore, there are several ways to denote numeric literals, which are described below.

Type	Description	Example
Hexadecimal Integer	Prefaced by "0x"	0xdeadbeef
Binary Integer	Prefaced by "0b"	0b10011011
Octal Integer	Prefaced by "0"	0712
Decimal Integer	Otherwise	167
Character (Integer)	Up to 8 characters enclosed in single, double, or back quotes. Back quotes enable C-style character escapes.	'ABCD' "abcdefgh" `hello\n`
Decimal Floating-Point	Begins with a digit and contains a decimal point and/or e or E with an exponent (which may optionally have a sign) for exponential notation. "∞" for infinity. "NaN" for NaN (case sensitive).	0.1 3.14 2.5e4 1.67e-11 -∞ NaN

Character Literals

One oddity in assembly language is that a character literal may contain more than one character. What's important to remember is that assembly language is not a typed language: everything is just bits that you can move to different places and interpret in different ways. Because of this, a character is just a number, and most assemblers will allow you to put several of them together.

When you do this, the assembler strings the 8-bit characters together to create a larger number. The assembler does this in such a way that, if you were to write this multi-byte character literal to memory, the characters would end up in memory in the correct order, with the first character in the literal being at the address you wrote to (i.e. it will account for endianness for you).

Character Escapes

If a character literal is enclosed in `back quotes`, C-style escapes are enabled. The following table lists the available escapes in CSX64 (and NASM, which CSX64 is loosely based on).

Escape Sequence	Resulting Value
\'	'
\"	"
\`	`
\\	\
\?	?
\a	Bell (alert) (ASCII 7)
\b	Backspace (ASCII 8)
\t	Tab (ASCII 9)

Escape Sequence	Resulting Value
`\n`	`New Line (ASCII 10)`
`\v`	`Vertical Tab (ASCII 11)`
`\f`	`Form Feed (ASCII 12)`
`\r`	`Carriage Return (ASCII 13)`
`\e`	`Escape (ASCII 27)`
`\377`	`Up to 3 octal digits - literal byte`
`\xFF`	`Up to 2 hex digits - literal byte`
`\u1234`	`Unicode character (not yet supported)`
`\U12345678`	

Instant Imm

For many operations, an imm is required to be "instant". This simply means that its value must be calculatable at that time. For instance, if an imm has expression `(a + b) / 2`, it is not instant until both `a` and `b` are instant. By definition, all numeric literals are instant, and thus expressions consisting entirely of numeric literals are also instant.

r – Register

A register operand refers to the contents of a register. As discussed in the section on registers, CSX64 has 16 general-purpose registers. To refer to one of these registers, you simply designate the name of the partition you want to use (e.g. RAX, EDI, SP, etc.).

The register name you use indicates the size of the operand (i.e. how much data is moved, processed, etc.). For instance, using EDI to load a value from memory (e.g. `mov edi, [var]`) loads a 32-bit value from memory into the 32-bit partition of RDI, whereas using DIL would only move 1 byte.

m – Memory

A memory value is an expression that evaluates to the address of some value in memory. In CSX64 assembly language, addresses are enclosed in brackets "`[…]`" with the address expression inside. An address expression follows the same rules as any imm, with the added ability to use up to two (64-bit) registers in computing the address, as discussed in the section on memory addresses.

Just like the machine code version of addressing (or rather because of it), the registers used in the calculation must be connected by addition or subtraction to the rest of the expression, may be multiplied by an integral 0, 1, 2, 4, 8, 16, 32, or 64, and only one of the registers may be subtracted (i.e. negated before adding). Registers may optionally be augmented by unary plus or negation operators. Unary plus is no-op, and negation is equivalent to adding the negative of the register.

The assembler can deduce the overall signedness of any register in the expression. For instance, if a register is connected by an even number of negations or subtractions, it understands that it is in fact not being negated.

If an address expression is not of the standard form, the assembler will attempt to use algebra to put it into a legal form if possible. (e.g. `[8*(rdi+4) + 64]` is converted to `[8*rdi + 96]`). This is only done for multiplication: other algebraic simplifications must be done by the programmer.

Finally, the assembler will combine multipliers of the same register (e.g. `[6*rdi - 8*rbx - 2*rdi]` is converted to `[4*rdi - 8*rbx]` and is thus legal). In fact, more than two registers may be specified, so long as only two of them have nonzero multipliers and only one of them is negated (e.g. `[4*rax - 8*rbx - 4*rax + 2*rcx]` is converted to `[-8*rbx + 2*rcx]` and is thus legal).

Size Directive

As stated in the section on register operands above, the register used determines the amount of data that is processed (i.e. the assembler knows RAX is 8-bit, ESP is 32-bit, etc.). However, this system presents a problem: what if you use a statement that doesn't contain any registers (e.g. `mov [var], 100`)? How would the assembler know what size you meant?

To solve this, you have to explicitly tell the assembler the size of the memory operand (e.g. `mov dword ptr [var], 100`). Additionally, in some assembly languages you need to specify the size as "`dword ptr`" and in others as just "`dword`". This becomes quite annoying, as neither is compatible with the other (CSX64 uses the "`dword ptr`" form).

The size directives used by CSX64 are as follow (not case sensitive):

Size Directive	Size
`BYTE PTR`	`8 bits`
`WORD PTR`	`16 bits`
`DWORD PTR`	`32 bits`
`QWORD PTR`	`64 bits`

Symbols

Traditionally, assembly doesn't have a notion of variables or data types: everything is just data. However, it becomes necessary to have symbolic names for referencing different values. For instance, if your program needed an IP address in several places spread out over many files, giving it a symbolic name would make the code far less obscure and easier to read and maintain. Without symbolic names, changing the IP address would mean having to hunt down every occurrence of it and change them all by hand.

Another compelling argument is in addresses: would you rather refer to your function as "the one at address 300" or "the one named cross_product"? And even more compelling: how do you even know what the absolute address of something will be? You'd have to count the number of bytes generated by each and every statement in your program and keep track of the position of everything manually. And even *if* you did that, if multiple files are used the order of those files in the final executable may vary. The simplest solution is to have the assembler and linker take care of all the heavy lifting for you.

Symbols in CSX64

A symbol is any user-created name that has an associated imm (though most assemblers/linkers defined a few symbols for you, some of which you couldn't come up with on your own, as they hold information about the resulting executable itself).

Symbols can be used in any imm expression, even before they've been defined (though that would make them not instant imms). Symbols should be used anywhere you would otherwise put a value that you will likely need to reference elsewhere.

It is important to remember that assembly language does not have a notion of variables. Symbols are not memory locations to put something in. In fact, they don't impact the resulting executable at all, and are simply a means of making assembly language more convenient.

All symbols must have a legal symbol name, which (mostly) follows the same rules as most programming languages: all characters in the name must be alphanumeric, underscores, or periods, and the first character may not be a number. Beginning a symbol with a period has special meaning.

Symbols that are not labels are created by the EQU directive.

Predefined Symbols

The CSX64 assembler defines several instant symbols automatically:

Symbol Name	Value
`__time__`	The time of the assembly process expressed as the number of 100 nanosecond intervals since January 1, 0001.
`__version__`	The version of CSX64, where increasing values indicate newer versions.
`__pinf__`	Floating-point positive infinity.
`__ninf__`	Floating-point negative infinity.
`__nan__`	Floating-point NaN.
`__fmax__`	The maximum (finite) floating-point value.
`__fmin__`	The minimum (finite) floating-point value.
`__fepsilon__`	The smallest positive floating-point value.
`__pi__`	The mathematical constant pi.
`__e__`	The mathematical constant e.

Additionally, there may be other instant symbols predefined by the assembler that were imported from external extensions. In fact, the vanilla CSX64 system predefines all its system call codes.

The CSX64 linker also defines several symbols automatically. Of course, being defined by the linker means these symbols are unavailable during assemble-time, which means none of these symbols can be used where an instant imm is required.

Symbol Name	Value
`__heap__`	The sum of the lengths of all the user-defined segments. This is thus the low side of the heap/stack segment that is allocated by the virtual operating system upon program initialization.

Labels

Labels are the tools assembly languages use to keep tabs on addresses. They are simply symbols that will refer to the address of something in the resulting executable. To do this, the assembler takes note of where the label was defined in the file. However, this is not enough. When you create an executable, you are essentially merging several source files into one result. Because the order of the source files in the result is undefined we also need to keep track of an offset to apply to the label to ultimately give it the absolute address that is needed. This offset will have to be defined by the linker when it goes about actually merging the files into an executable after each has been assembled individually. The important takeaway is that, because of this, labels cannot be used where an instant imm is required. All of this will be explained in more detail in Assembly Process.

Labels should be used when you need to refer to something's address (e.g. function, loop, or data location).

Local Label

A local label is a label that is associated with a non-local label. Because assembly language is prolific with labels, you can easily run into the problem of having clashing label names. Rather than naming your labels "loop_top_1", "loop_top_2", etc., you can give them local names. A local label is considered "tied" to the most-recent non-local label.

In most assembly languages – and indeed in CSX64 – local labels are created and referenced by putting a period in front of the normal label name (e.g. ".top", ".end", ".ret" are all local labels). In the NASM assembler – and in CSX64 – putting a period in front of a symbol name actually just appends the last label that didn't have a period in front of it to the front of the local symbol, so local label ".top" under non-local "foo" actually creates a symbol named "foo.top". Because of this, you can technically reference another function's local symbols if you really wanted to for some reason.

Address Difference

While it's true that labels are not instant imms due to the missing offset, the *difference* of two labels in the same file and same segment *is*. This is often useful for finding the length of an array. For instance, if we define a string like so:

```
string_start: db "hello world!"
string_end:
```

Then the imm (`string_end - string_start`) is instant (and has a value of 12). This is because the assembler can recognize that the base offsets of `string_end` and `string_start` are being canceled by the subtraction.

This is frequently used to find the length of a string, and less-frequently to compute the length of an array. Just remember, this only works for arrays defined at compile time. Adding or removing data from an array at runtime will not change the result of this value because these values are computed at compile time. Adding data would be overwriting whatever happened to come after the array.

Label Macros

For convenience, the CSX64 assemble provides a few label macros:

Label Macro	Description
$	Expands to the address of the current line.
$$	Expands to the address of the start of the current segment in the result.

These are meant to be a convenience for use in specific scenarios. For instance, this could be used to find the length of a string by defining a symbol immediately after the string definition with the value `$-the_string`, which is much simpler than creating another label before performing the equivalent subtraction.

Instruction Syntax

Assembly language is notably different from other, higher-level languages in that it is very mechanical. The best example of this is the utter uniformity of its statements. CSX64 assembly uses a very standardized approach, where every line is separated into several sections:

```
(label:) (op arg1, arg2, ...) (;comment)
```

The optional label section consists of a new label to introduce, which may optionally be made local by appending a period to the front. The colon is not part of the label name.

The optional operation section consists of one operation to perform. An operation (unless it is a directive – see below) assembles into exactly one machine code instruction (e.g. ADD, SUB, MUL).

The argument section consists of zero or more comma-separated arguments for the selected operation.

The optional comment section begins with a semicolon and continues to the end of the line. It is ignored during assembly. Much like in C-derivative languages, a semicolon marks the end of a statement in assembly. The difference, however, is that in assembly you can't put two statements on the same line, which is why the semicolon in assembly is used as a comment character.

In CSX64, the parsing of a line of assembly begins by splitting the line into its constituent components. From there, **each token is stripped of *all* white space** (except inside a string or character literal). Because of this, spacing in any expression is completely up to the programmer (i.e. "`(a+b)*(b+c)`" and "`(a + b) * (b + c)`" are identical). This also means that numeric literals can be spaced out to be easier to read (e.g. `1 000 000 000` and `1000000000` will be parsed identically). This is especially useful for writing binary literals.

Directives

CSX64 assembly has several pseudo-instructions that appear to be instructions in the assembly code but don't necessarily correspond to the generation of any binary code in the resulting executable. Primarily, these directives offer a means of controlling some aspect of the assembly process.

Global

The Global directive takes one or more symbol names and adds them to the export table (i.e. the symbol names in this file that are accessible from other assembly files). A symbol can be made global before being defined. Local symbols cannot be made global.

A symbol name that is made global but is never defined is an assemble error.

Typically, an assembly file will begin with a series of global directives exporting functions and global variables. This makes it easy for a programmer using that file to open it and immediately see what kinds of things it makes available. This is similar to the concept of header files in C/C++.

Extern

The Extern directive takes one or more symbol names and adds them to the import table (i.e. the global symbols from other files that are accessible from this file). While an external symbol can be used prior to the Extern directive, it is best practice to put all Extern directives at the top of the file (usually after listing globals).

A symbol name that is made external but is also defined in the file is an assemble error.

Declare (DB, DW, DD, DQ)

Declare is meant to be used in the data segment (but can also be used in the text segment if the need arises). It takes one or more imms and writes them directly into the binary file in-place from left to right, accounting for the endianness of the current system. Each argument is written with a word size given by the directive used. DB for bytes, DW for words, DD for dwords, and DQ for qwords.

Additionally, if an argument is a line of text enclosed in single, double, or back quotes, Declare will write the entire string as if it were a series of maximum-width character literals for the word size requested. Or, in other words, writes the characters each as bytes in the correct order and pads the end with zeroes to reach a multiple of the word size used. This is summarized below:

As a String	Series of Character Literals	Series of Bytes
db “hello”	db ‘h’, ‘e’, ‘l’, ‘l’, ‘o’	db ‘h’, ‘e’, ‘l’, ‘l’, ‘o’
dd “hello world”	dd ‘hell’, ‘o wo’, ‘rld’	db ‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘ ‘, ‘w’, ‘o’, ‘r’, ‘l’, ‘d’, 0
dq “happy”	dq ‘happy’	db ‘h’, ‘a’, ‘p’, ‘p’, ‘y’, 0, 0, 0

It should be noted that assembly languages do not append a null-terminator for you when writing strings. If your program requires strings to be null-terminated, append a zero after the string (does not need to be a character literal – e.g. db `hello world!\n`, 0). If your algorithm doesn’t require a null terminator (e.g. direct use of sys_write), feel free to omit it.

Declare is the means by which you allocate space for static duration variables (i.e. not on the stack) and assign them initial values (e.g. in the data segment). To do this, you first determine what data size you need. Let’s say we want to make a variable to hold a 32-bit RGBA color code. Now we pick out a suitable initial value (optional). We then give that information to DD and attach a label to it for convenience (so we can access it in the code later): `rgba_color: dd 0xff4286f4`. This writes our color code directly to the binary file as a 32-bit integer. Thus, `rgba_color` now points to a unique 32-bit location in the binary file, where we can load and store values as we wish (i.e. a global variable). Care should be taken when selecting declare sizes and values. If we wrote `dw 0xff4286f4`, then the value that was **actually written** would be `0x86f4`. Since we asked it to write that color code as a 16-bit value, our value would be truncated.

If an argument begins with a hash (#) it denotes a multiplier. When a multiplier is encountered, the previous argument is repeated that many times in the resulting binary code (where a multiplier of 1 is no-op). A multiplier must resolve to an instant imm integer that is greater than zero. A multiplier cannot immediately follow a string, nor can it immediately follow another multiplier. For convenience, if the first argument is a multiplier, an integral zero is taken as the previous argument to be repeated (though if this is all you'll be declaring it would likely be better to put this variable in the BSS segment).

Reserve (RESB, RESW, RESD, RESQ, REST)

Reserve is like Declare except that it can only be used in the BSS segment and cannot be used to initialize the data to a specific value (by virtue of being in the BSS segment). Additionally, Reserve takes only a single argument, which is the number of words of the given size to allocate (and zero) upon program initialization (e.g. resw 32 reserves 32 contiguous 16-bit locations). All of the normal suffixes function the same as their Declare counterparts except the addition of REST, which uses a word size of 10 bytes (in other assembly languages this is usually used for extended-precision floating point values, but CSX64 does not yet support this).

EQU

The EQU directive is used to give a value to a label. When a label is encountered it is added to the symbol table and given a value that points to the current position in the current segment. The EQU directive replaces this default value (of the label on the same line) with a given imm (e.g. "`symbolic_name: equ 23 * 25`" creates a symbol named "symbolic_name" that has a value of 575. While labels are not normally allowed outside of any segment declaration, they are permitted if EQU is used along with it (this is commonly used for defining global constants).

Segment / Section

The Segment / Section directives (synonyms) change the segment that subsequent code will be added to. The segment to change to is given as the one (and only) argument, which is case-sensitive in most assembly languages, but is not in CSX64. Segment names are those listed in the segment section (prefaced with a '.'). It is an assemble-time error to redeclare a segment that has already been declared (i.e. parts of a segment must be contiguous).

Calling Conventions

Unlike in literally every other programming language, in assembly you have choices to make about how data is passed to functions and how you call and return from them. Although there are many methods, we'll talk about some of the common ones that you'll probably want to employ.

Let's say you have a function named "add" that takes two integers, adds them, and returns the result. In any other language you would use this as "add(a, b)". The compiler would then take what you give it and provide it to the function in a way that is invisible to the programmer. Assembly is not so kind however. Remember, there is no such thing as a function in assembly. There is only binary data that we elect to give meaning to via a specific interpretation. What we consider to be a function in assembly is actually just a labelled position in the code that happens to have some notion of a call/return mechanism. The question then is how to use them as functions.

Calling a Function

To call a function, we typically use the CALL instruction, which will push the (64-bit) address of the next instruction onto the stack and then jump to the function. In this case, we use the RET instruction, which pops a 64-bit value off the stack and jumps to it.

However, you could also do these stack manipulations manually. For instance, you could push the desired return point and use a JMP instruction to jump to the function (rather than a CALL). This way, if the function is RET-compatible, the function will return not to the point of invocation, but to wherever you specified with the address you manually pushed onto the stack (though this is considered – for good reason – very, very bad form).

JMP can also be used to jump to the value of a register. Thus, you could even put the return address in a register, in which case the function should not use RET or stack manipulation, but simply jump to the address held in that register (this is less deplorable because it's not invisible to the called function – this could be used for pseudo-inlining so that main memory doesn't slow anything down, but the function doesn't have to be copied everywhere it was used).

There are many, many ways you could potentially set up your function-calling system. For instance, you could put the address of a pointer to the return point in a register. You could even push the address of a function that should be called that returns an address in a register that should be treated as the first function's return point.

For the sake of simplicity and readability, however, do yourself a favor and just use CALL/RET.

And that's just for calling and returning. This problem also arises in passing the arguments to the function.

The Stack Approach

One of the most common calling conventions is to push arguments onto the stack in reverse order. This right-to-left ordering is so that the first argument appears in memory at an address less than the second, which appears at an address less than the third, etc. The function could then refer to the arguments with an offset relative to the top of the stack, keeping in mind that the top of the stack will point to the 8-byte return address (e.g. assuming 32-bit args: a = [RSP + 8], b = [RSP + 12]).

With this approach there's also the question of argument cleanup. You have 2 options: have the caller pop the arguments back off the stack (the good, well-mannered approach), or have the function you called do it (the old, potentially dangerous way). This will be discussed in the section on Register Safety.

Stack Framing

In real-world use, it's often desirable to be able retrace a path down the call chain (e.g. for debugging purposes). With nothing else but the stack pointer this would be impossible (though a compiler for a language such as C could still manage to pull it off because it knows more information about every function that exists, which is why many C compilers don't do what we're about to discuss). For this reason, we use what's called the stack base pointer (RBP). Before beginning the process of pushing args for a function call, we first push the **current value** of the base pointer onto the stack and then load the base pointer with the **new value** of RSP. In this way, the base pointer always points to a copy of its previous value (i.e. a linked list of base pointer values).

After the function returns, you'd need to undo all of your argument pushes, but this can be simplified by loading RSP with the current value of RBP (since they were equal prior to when we started pushing arguments). Then you'd just need to pop the old value of RBP off the stack and back into RBP.

This system makes it so that the base pointer always points to the base of a smaller "call stack", where you can refer to arguments with fixed offsets from the static **base** of the stack rather than the mutable **top** of the stack, which may change due to creating/destroying local variables or other stack frames (e.g. assuming 32-bit args: a = [RBP - 8], b = [RBP - 4]).

This system is called stack framing (for this reason, the base pointer is sometimes called the frame pointer), and the linked list of base pointer values is thus a linked list of stack frames. To get the stack frame from n function calls back, all you'd need to do is dereference the base pointer n times. You can tell a lot about what's going on under the hood by examining the stack frames for a function call tree, including the return address of the called function, which you could use to figure out where the function was called from. Nowadays, compilers frequently don't do all of this unless you explicitly tell it to with a command line option or some form of interactive debugging mode.

Keep in mind that even if you use the stack method you don't have to also create stack frames. They're entirely just a means of manually debugging a program without the use of more advanced, usually integrated debuggers. If you choose not to use stack frames, you're free to use the base pointer register as any other general-purpose register.

The Register Approach

You might have been wondering something along the lines of "if we have these super fast register things why are we not using them for function calling too?" Congratulations, you've stumbled onto what is quickly becoming the standard, modern method of function calling. Previously this wasn't entirely feasible for various reasons, one of which being that certain instructions require things to be in specific, preset registers – which would entail a lot of register shuffling – and another being that the typical 32-bit processor only had 8 registers (2 of which were always occupied for managing the base/stack pointers).

In the register approach, instead of pushing the arguments onto the stack in a specific order, you load arguments into registers in a specific order. Then you call the function, the function returns, and you're done. As arguments were not pushed onto the stack, nothing needs to be popped off the stack after a call. Additionally, because there is less manipulation of memory, the whole process is faster by orders of magnitude (unless the values were sourced from memory anyway).

There are trade-offs with this approach, however. With the stack approach you can use data types that won't fit in a single register (though with the register approach you could simply take a pointer to it instead), and you can accept many more arguments. With the register approach you potentially have much faster execution and don't have to concern yourself with many pushes and pops (one of the most common problems in assembly is not matching a push with a pop).

Return Value

Then there's the issue of the return value. In this respect both methods are the same: either put the return value in a register or take a pointer argument for the desired return value destination in memory. This is because you will eventually be returning from the function. You can't exactly push a return value onto the stack and then pop back to the invocation point (or at least you shouldn't, as that's what a dangling pointer really is).

Register Safety

One thing that should be obvious is that, when you call a function, that function will have to use registers to do things. If you have a value in a register (perhaps not even as an argument – e.g. a loop index) and then you call a function, there's no telling what the value in that register will be when it returns. When a function abandons the value in a register and uses it for its own purposes, that function "clobbers" the register. Any calling convention, in addition to all the information on how to pass arguments, clean up, etc. should also define any registers whose values are preserved across function calls.

For instance, the base/stack pointer registers should obviously have the same value after the function returns, as otherwise the stack would suffer irreparable damage and you'd be popping the wrong values into the wrong places. It's actually fairly interesting how few steps it takes before this leads to an error that causes the program to crash (usually from dereferencing a pointer that's unimaginably out of bounds). It's for this reason that it was noted in the Stack Approach that having the function you called clean up the stack is the bad, old way of doing things. If the function is doing the cleanup, then the function has a net effect on the stack register. This on its own is not dangerous, but it makes it much easier to mismatch a push/pop when they aren't together, or – just as bad – cleaning up twice.

If you're about to call a function and you have registers that aren't call-safe and that hold values you'll still need after the function returns, you need to store them somewhere before the call procedure – typically on the stack – and get them back after the return/cleanup (so you can ensure the value is not clobbered by the function).

Moreover, if you're about to use a call-safe register you need to store its current value somewhere before using it and then get that value back before returning (so that you don't clobber the call-safe register).

Standard Calling Conventions

So then, you may be thinking this is probably a bit mind-numbing. Rest assured, the method you use to accomplish a call/ret system doesn't matter as long as you're consistent (though it's better to use common methods since they already have code that's written for them). We'll now go over an efficient calling convention that I'd recommend you use. Other standard conventions can be found here:

https://en.wikipedia.org/wiki/X86_calling_conventions

1) Use CALL/RET (do not perform bastardized jumps).
2) Put the first 6 integral arguments left-to-right into RDI, RSI, RDX, RCX, R8, R9 (or partitions thereof).
    a. Put the first 8 floating-point arguments right-to-left on the FPU register stack.
    b. Put any extra arguments on the stack from right-to-left.
    c. If the data type will not fit in a register, consider taking a pointer, otherwise put it on the stack.
3) Place the return value in RAX (or a partition thereof)
4) Registers RBP, RSP, and R13-R15 are call-safe.

Command Line Arguments

It's important to remember that the "main" function of an assembly program is still a function and can take arguments. This, of course, you should already know. The only issue now is: where does the operating system put them when it first begins executing your program? They will always either be on the stack or in registers. In fact, most systems (including CSX64) put them in both places to ensure your program works regardless of which calling convention (of the primary 2) you use for main.

As you should be using the register approach described above, you should thus assume RDI contains the number of arguments, and RSI contains a pointer to an array of pointers to C-style (null-terminated) strings. Remember that pointers are 64-bit in CSX64.

As a side note, even when running in 32-bit mode on a 64-bit machine, the operating system will generally zero the high bits of the two 64-bit registers so that they can be used as if they were 64-bit values for compatibility reasons. However, their representation on the stack is less well-defined and you may need to reference the system's documentation. Another reason to use registers instead.

Assembly Process

When you write an assembly program you are writing an assembly source file. Assemblers take a **single** assembly file and create a **single** object file. Object files are the machine code translation of an assembly file plus additional data pertaining to missing information (i.e. info on how to account for symbols that were not defined in that file, including the base of all labels), the definitions of global symbols, etc.

An object file itself is not executable: it needs to have all of its missing information filled in. To do this, a separate program known as the linker takes **all** the object files and cross-references them to fill in any missing symbol definitions. Ultimately, the linker puts all the object files together and creates the final executable.

The purpose of this is to have a system where only files that were modified need to be reassembled when any change is made. This of course also means that large files (e.g. library implementations) do not need to be assembled every time you write a program that references them. Furthermore, many software developers prefer to distribute their libraries as object files, as this simultaneously mitigates the assembly time of clients of the library and prevents them from having the source code (although the object file could be disassembled, all the comments, spacing, and most symbol names would be gone forever, leaving the disassembled file a virtually-illegible pile of spaghetti.

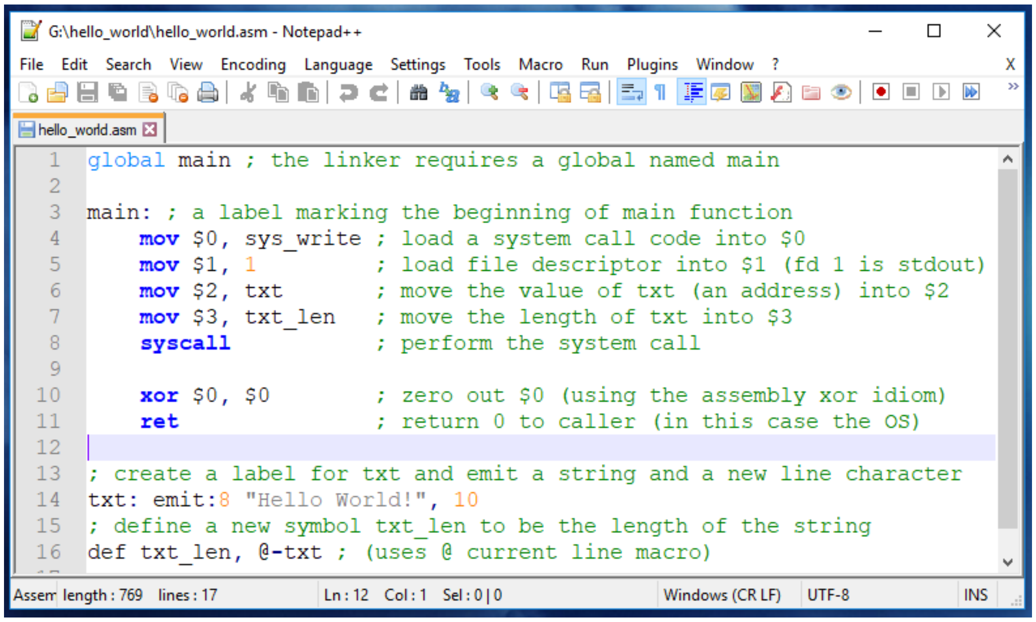
Well, you've made it this far, so how about we finally talk about how to do all this?

Development

The development process is the only part that should concern you (as assembly is a cruel, vindictive mistress). Fortunately, you should also already be very familiar with it. This step just involves opening an editor of your choice and typing up your program.

What kind of educator would I be if we didn't make a Hello World! project? Let's begin.

Open your editor of choice (I'll be using Notepad++). You can save your file as whatever you wish, but .asm for an assembly file will likely be more convenient for everyone involved.


G:\hello_world\hello_world.asm - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
hello_world.asm
Assem length : 769 lines : 17
Ln : 12 Col : 1 Sel : 0 | 0
Windows (CR LF) UTF-8 INS

```
 1  global main ; the linker requires a global named main
 2
 3  main: ; a label marking the beginning of main function
 4      mov $0, sys_write ; load a system call code into $0
 5      mov $1, 1         ; load file descriptor into $1 (fd 1 is stdout)
 6      mov $2, txt       ; move the value of txt (an address) into $2
 7      mov $3, txt_len   ; move the length of txt into $3
 8      syscall           ; perform the system call
 9
10      xor $0, $0        ; zero out $0 (using the assembly xor idiom)
11      ret               ; return 0 to caller (in this case the OS)
12
13  ; create a label for txt and emit a string and a new line character
14  txt: emit:8 "Hello World!", 10
15  ; define a new symbol txt_len to be the length of the string
16  def txt_len, @-txt ; (uses @ current line macro)
```

Here you see the basic structure of an assembly program and just how tedious even printing a single line of text can be (but rest assured there are ways around many of these problems).

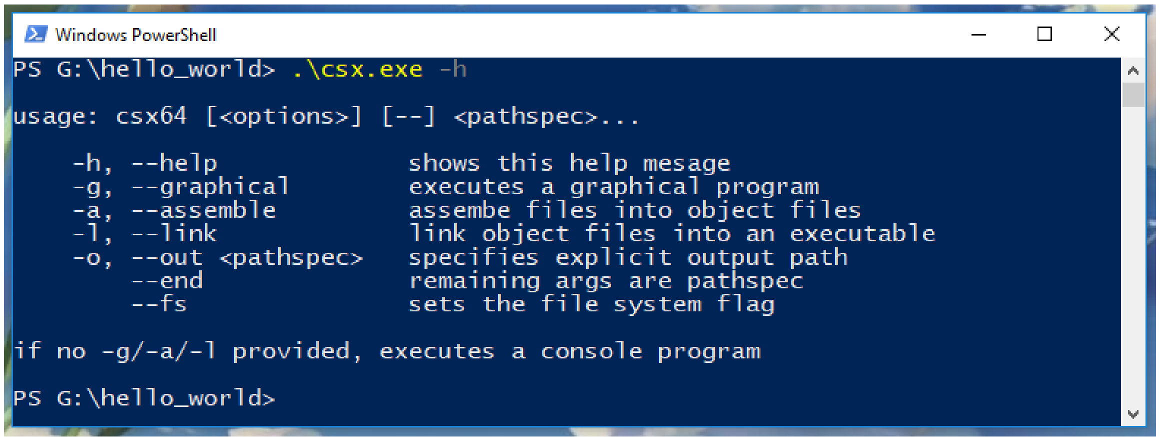
A few things to note:

1) We defined a “function” called main and marked it as global for the linker.
2) We used several statements to get the processor into a state we wanted before actually performing the system call (this also applies to any function call).
3) We set $0 (return value) to zero using the xor idiom. It is much more efficient both in terms of executable size and execution time to use xor a register with itself than to set it to zero, as setting I to zero would need to store the 8-byte value (of zero), that will be loaded into the register, while xor will just take the 1-byte register address pair. In other languages if you set something to zero the compiler will usually optimize that assignment into this form.
4) Because this is a function that is CALL-compatible, we must return (line 11).
5) We need to put the string to print out somewhere. Static variables are typically placed at the bottom or top of an assembly file. We create a label for the variable (in this case a string) and write out its value using the Emit directive.
6) sys_write requires we know the length of the string (because it actually deals with writing binary data, not text, but this isn’t a problem since the binary representation of text is text). To do this we introduce a new symbol named txt_len immediately after emitting the string with value @-txt using the @ macro to get the current line’s address and subtracting the address of the beginning of the string. This gives us the difference in bytes, and since a character is 1 byte, this is also the number of characters in the string. You could also do this by creating another label immediately after emitting the string and using that in place of @.

Assembling

From here we have to assemble it into an object file before we can turn it into an executable. In this example I’ll be using PowerShell. At the time of writing this manual, PowerShell does not support file redirection. If you're on Windows and need file redirection, consider using Command Prompt (cmd.exe) or downloading a reputable third-party console emulator (e.g. Git Bash).

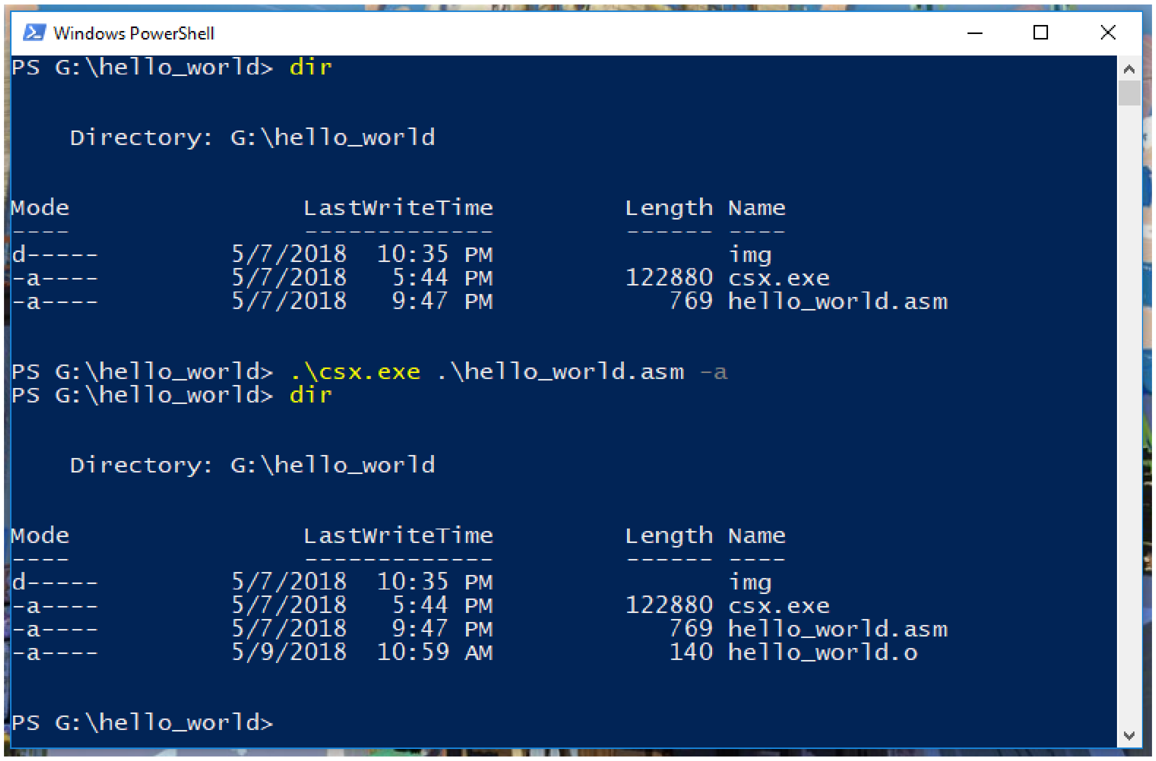
This will be the first occurrence of actually using the CSX64 application (csx.exe). Providing the -h option will list all of the available options:



```
PS G:\hello_world> .\csx.exe -h

usage: csx64 [<options>] [--] <pathspec>...

    -h, --help              shows this help mesage
    -g, --graphical         executes a graphical program
    -a, --assemble          assembe files into object files
    -l, --link              link object files into an executable
    -o, --out <pathspec>    specifies explicit output path
        --end               remaining args are pathspec
        --fs                sets the file system flag

if no -g/-a/-l provided, executes a console program

PS G:\hello_world>
```

We want to assemble hello_world.asm into an object file, so we need the -a option.

The following shows the directory before assembling the file, the command to assemble it, and the directory afterwards:

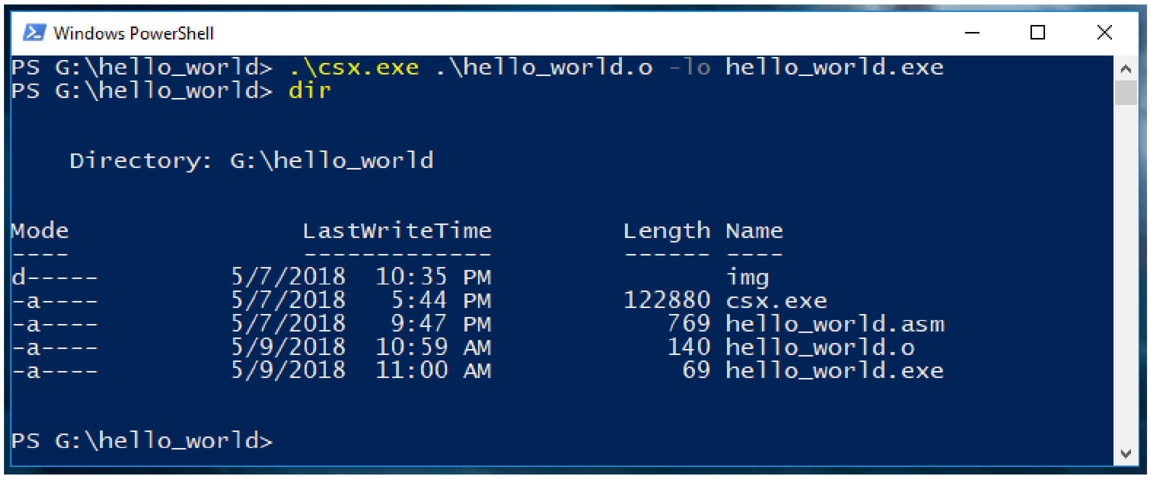


```
PS G:\hello_world> dir


    Directory: G:\hello_world


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----         5/7/2018  10:35 PM                img
-a----         5/7/2018   5:44 PM         122880 csx.exe
-a----         5/7/2018   9:47 PM            769 hello_world.asm


PS G:\hello_world> .\csx.exe .\hello_world.asm -a
PS G:\hello_world> dir


    Directory: G:\hello_world


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----         5/7/2018  10:35 PM                img
-a----         5/7/2018   5:44 PM         122880 csx.exe
-a----         5/7/2018   9:47 PM            769 hello_world.asm
-a----         5/9/2018  10:59 AM            140 hello_world.o


PS G:\hello_world>
```

As you can see, we have now created our object file "hello_world.o". If there were any assemble errors, they would have been displayed and the object file would not have been created (or overwritten if it already existed).

Linking

We now need to link our object files (in this case just the one) together to create an executable. To do this we use the -l (lowercase L) option:

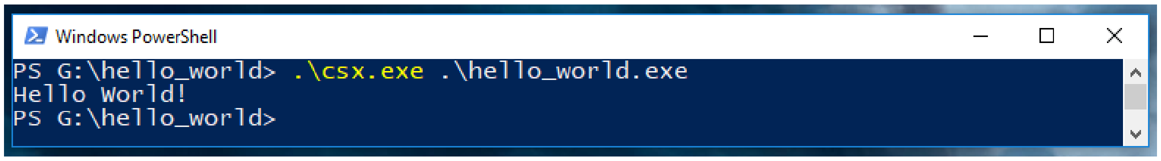

Windows PowerShell

```
PS G:\hello_world> .\csx.exe .\hello_world.o -lo hello_world.exe
PS G:\hello_world> dir


    Directory: G:\hello_world


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----         5/7/2018  10:35 PM                img
-a----         5/7/2018   5:44 PM         122880 csx.exe
-a----         5/7/2018   9:47 PM            769 hello_world.asm
-a----         5/9/2018  10:59 AM            140 hello_world.o
-a----         5/9/2018  11:00 AM             69 hello_world.exe


PS G:\hello_world>
```

Here you see we also provided the -o option (combined as -lo) (any CSX64 short name options can be combined in this way) to specify the output, followed by the path to put the result. This can be omitted, but it will default to "a.exe", which is fairly nondescriptive but may be used if you should ever need it.

Executing

We now have a working executable. But hold your horses, don't try to double click it or run it from the console. It's a CSX64 executable, so it needs to be run by csx.exe. As the -h help option explained, providing no options (specifically no -a or -l options) defaults it to execution mode:


Windows PowerShell

```
PS G:\hello_world> .\csx.exe .\hello_world.exe
Hello World!
PS G:\hello_world>
```

As a side note, all the options after csx.exe will be passed to the program, with the number of arguments in $1 and a pointer to an array of pointers to the arguments in $2, as discussed in Command Line Arguments.

Virtual Operating System

The virtual operating system is the "platform" on which the CSX64 processor runs. In essence, it acts as an extension mechanism for the default processor behavior. Specification-compliant virtual operating systems are effectively dormant until called upon by the execution of a SYSCALL operation and will not affect execution of your program after returning.

Now, we'll go over some of the fundamental aspects of all modern operating systems.

File Descriptors

One of the most important roles of an operating system is in managing a file structure on the hard drive. There are many, many ways to go about this, so we won't get into the specifics of how that's done. However, one aspect of the file system that is always present is a way for client code to access it. This is accomplished via so-called file descriptors, which are essentially an operating system-specific data structure that will be used to grant access to the file system. To do this, you must first request a file descriptor via system call. It will then provide you with the file descriptor address, which for security reasons is usually an index in an array you don't know the position of and consisting of objects you don't know the structure or size of. From there, you can use that file descriptor index in various other system calls to perform the desired file operations.

That said, there is usually a set number of file descriptors for any given system. If you fail to close a file descriptor, the system will still believe it's in use. Moreover, if you attempt to open a file when there are no unused file descriptors available the request will fail, which on some systems may yield a "null" value (not necessarily zero) to indicate the failure, and on others may cause an exception. In CSX64, execution will terminate with the InsufficientFDs error code. Therefore, you should try to keep the number of files you have open at any given time to a minimum and ensure that you close every file you open.

Managed File

A managed file is considered to be entirely controlled by client code and is closed upon termination. This is also the case for most modern operating systems; however, incorporating this into your algorithms for file IO is considered extremely bad practice.

Unmanaged File

An unmanaged file is considered to be only partially controlled by client code and is actually "owned" by some external code. Upon termination or being closed via sys_close, an unmanaged file will not be closed, but its connection to the file descriptor will be severed (thus freeing it for reuse).

Interactive File

If a file is interactive, reading past the end of the file will put the processor in the SuspendedRead state, pending more data from some external source. Execution will resume when there is more data to read. For instance, this could be used to simulate a standard console input stream.

File Mode

When opening a file, the file mode specifies how to go about opening the file. The file modes present in CSX64 are listed below:

File Mode	Value	Description
Create New	1	Creates a new file. Fails if the file already exists.
Create	2	Creates a new file if it doesn't exist or opens an existing file and truncates it to a length of zero bytes.
Open	3	Opens a file that already exists.
Open or Create	4	Opens an existing file if it exists or creates a new file.
Truncate	5	Opens an existing file and truncates it to a length of zero bytes. Attempting to read from a file opened with this mode results in an error.
Append	6	Opens an existing file and immediately seeks to the end.

File Access
When opening a file, the file access specifies what the file descriptor will have access to do.

File Access	Value	Description
Read	1	File is only allowed to read.
Write	2	File is only allowed to write.
Read / Write	3	File is allowed to read or write.

File Seek Origin
Of course, when reading a file, we also need to know where we are in the file. Thus, each file descriptor has an associated position for the read/write cursor. In CSX64 you can get this position with sys_tell. You can also set the position via sys_seek. However, sys_seek has several seek origins, which change how the position you provide is interpreted. These modes are listed below:

Seek Origin	Value	Description
Begin	0	Sets the position relative to the beginning of the file. Zero is the beginning of the file. Higher values indicate higher addresses in the file.
Current	1	Sets the position relative to the current cursor position. Zero is the current position. Higher values indicate higher addresses in the file.
End	2	Sets the position relative to the end of the file. Zero is the end of the file. Higher values indicate lower addresses in the file.

System Calls
A system call is made by the SYSCALL operation and is a request for the virtual operating system to take over and perform some specialized task before resuming execution of client code.

By default, the CSX64 system call table offers several low-level utilities such as file access, though extension libraries are free to add more functionalities or even modify or remove existing features.

The default operating system has a system call code be loaded into $0 prior to executing a SYSCALL operation. Based on the value in $0, various tasks will be performed. Note that a virtual operating system is free to do whatever it wants to do. It may modify memory, registers, flags, or even hidden registers that client code doesn't have access to (e.g. the execution pointer).

The default system call table is listed below:

Name	Code ($0)	Description
sys_read	0	Reads binary data from a file. R1 should contain the file descriptor to access. R2 should contain the address of the location to store the data. R3 should contain the maximum number of bytes to read. The number of bytes read will be placed in R0. ZF is set if zero bytes were read and cleared otherwise. Attempting to read at EOF is not an error, however if the file is interactive, the processor will be set to the Suspended Read state pending data from an external source (e.g. from the keyboard when using the console client). Fails with OutOfBounds if file descriptor is out of range. Fails with FDNotInUse if the file descriptor specified is not in use. Fails with IOFailure if an IO error occurs.
sys_write	1	Writes binary data to a file. R1 should contain the file descriptor to access. R2 should contain the address of the data array to write. R3 should contain the number of bytes to write. Fails with OutOfBounds if file descriptor is out of range. Fails with FDNotInUse if the file descriptor specified is not in use. Fails with IOFailure if an IO error occurs.
sys_open	2	Opens or creates a file and ties it to a file descriptor. Files opened with this call are considered "managed" and will be closed upon termination. R1 should contain the address of the C string to use as the path. R2 should contain the file mode to open with. R3 should contain the file access to open with. The file descriptor opened will be placed in R0. Fails with FSDisabled if FSF is not set. Fails with InsufficientFDs if there are no unused file descriptors. Fails with OutOfBounds if the string went out of memory bounds. Fails with IOFailure if the file could not be opened.

Name	Code ($0)	Description
sys_close	3	Flushes and closes a file opened via sys_open. R1 should contain the file descriptor to close. Fails with OutOfBounds if file descriptor is out of range.
sys_flush	4	Flushes a file descriptor's data buffer. R1 should contain the file descriptor to flush. Fails with OutOfBounds if file descriptor is out of range. Fails with FDNotInUse if the file descriptor specified is not in use. Fails with IOFailure if an IO error occurs.
sys_seek	5	Moves the current read/write position in the file. R1 should contain the file descriptor to seek in. R2 should contain the address in the file to seek to. R3 should contain the seek mode to use. Fails with OutOfBounds if file descriptor is out of range. Fails with FDNotInUse if the file descriptor specified is not in use. Fails with IOFailure if an IO error occurs.
sys_tell	6	Gets the current read/write position in the file. R1 should contain the file descriptor to get the position of. The current position will be placed in R0. Fails with OutOfBounds if file descriptor is out of range. Fails with FDNotInUse if the file descriptor specified is not in use. Fails with IOFailure if an IO error occurs.
sys_move	7	Moves a file in the file system. R1 should contain the address of the C string source path. R2 should contain the address of the C string destination path. Fails with FSDisabled if FSF is not set. Fails with OutOfBounds if either string went out of memory bounds. Fails with IOFailure if the file could not be moved.
sys_remove	8	Removes a file in the file system. R1 should contain the address of the C string path to remove. Fails with FSDisabled if FSF is not set. Fails with OutOfBounds if the string went out of memory bounds. Fails with IOFailure if the file could not be removed.

Name	Code ($0)	Description
sys_mkdir	9	Creates a new directory in the file system. R1 should contain the address of the C string path to create. Fails with FSDisabled if FSF is not set. Fails with OutOfBounds if the string went out of memory bounds. Fails with IOFailure if the directory could not be created.
sys_rmdir	10	Removes a directory from the file system. R1 should contain the address of the C string path to remove. Fails with FSDisabled if FSF is not set. Fails with OutOfBounds if the string went out of memory bounds. Fails with IOFailure if the directory could not be removed.
sys_exit	11	Immediately terminates execution with the specified return value. Returning from main invokes this system call with the value of $0. R1 should contain the return value to report.

NOP – No Operation

OP Code:

00

Description:

Specifies no operation. Usually only used for timing purposes.

Usage:

```
NOP
```

Flags Affected:

None.

Format:

```
[NOP]
```

HLT – Halt Execution

OP Code:

01

Description:

Terminates the program with the Abort error code and yields to the virtual operating system.

In a typical operating system a halted program may be resumed by the system, but this is not the case in vanilla CSX64 (though extensions are free to do so).

Usage:

```
HLT
```

Flags Affected:

None.

Format:

```
[HLT]
```

SYSCALL – System Call

OP Code:

02

Description:

Causes program execution to temporarily yield to the virtual operating system for it to perform some specialized process. The results of this operation are thus virtual operating system-specific.

If the operating system fails to perform the system call, this operation generates an UnhandledSyscall error.

This instruction is the only way to access potential language extensions.

Usage:

```
SYSCALL
```

Flags Affected:

Virtual operating system-dependent.

Format:

```
[SYSCALL]
```

PUSHF/PUSHFD/PUSHFQ – Push Flags

OP Code:

03

Description:

PUSHF pushes the FLAGS register (16-bit) onto the stack.

PUSHFD pushes the EFLAGS register (32-bit) onto the stack.

PUSHFQ pushes the RFLAGS register (64-bit) onto the stack.

When pushing EFLAGS or RFLAGS the VM and RF flags are cleared in the image stored on the stack.

Usage:

```
PUSHF
```

Flags Affected:

None.

Format:

```
[PUSHF]   [8: ext]
     ext = 0: push FLAGS
     ext = 1: push EFLAGS
     ext = 2: push RFLAGS
```

POPF/POPFD/POPFQ – Pop Flags

OP Code:

04

Description:

POPF pops a 16-bit value into the FLAGS register.

POPFD pops a 32-bit value into the EFLAGS register.

POPFQ pops a 64-bit value into the RFLAGS register.

Reserved flags and the VM/RF flags are not modified by this instruction (ignored in the loaded image).

Usage:

```
POPF
```

Flags Affected:

None.

Format:

```
[POPF]    [8: ext]
     ext = 0: pop FLAGS
     ext = 1: pop EFLAGS
     ext = 2: pop RFLAGS
```

CLx/STx – Clear/Set flag

Mnemonic	Flag Affected	OP Code
CLC	CF	05 00
STC		05 80
CLI	IF	05 01
STI		05 81
CLD	DF	05 02
STD		05 82
CLAC	AC	05 03
STAC		05 83

Description:

Clears (CLx) or sets (STx) a bit in the flags register. In compliance with standard x86_64, not all flags are represented by this family of instructions.

Usage:

```
CLx/STx
```

Flags Affected:

None.

Format:

```
[CLx/STx]   [1: value][7: flagid]
```

SETcc – Conditional Set

Condition	OP Code	Condition	OP Code	Condition	OP Code	Condition	OP Code
Z	06 00	NZ	06 01				
S	06 02	NS	06 03	B	06 0a	L	06 0e
P	06 04	NP	06 05	BE	06 0b	LE	06 0f
O	06 06	NO	06 07	A	06 0c	G	06 10
C	06 08	NC	06 09	AE	06 0d	GE	06 11

Description:

Sets the 8-bit destination to 1 if the condition is met, or 0 otherwise.

Usage:

```
SETcc r8/m8
```

Flags Affected:

None.

Format:

```
[SETcc]    [4: dest][2: size][1: h][1: mem]
     mem = 0:               dest <- f(dest)
     mem = 1: [address]    M[address] <- f(M[address])
     (h marks AH, BH, CH, or DH for dest)
```

MOV – Move

OP Code:

07

Description:

Copies a value from some source to a destination.

Does not support memory-to-memory transfers (in accordance with modern processor architectures).

Usage:

```
MOV r, imm/r/m
```

```
MOV m, imm/r
```

Flags Affected:

None.

Format:

```
[MOV]   [4: dest][2: size][1:dh][1: sh]   [4: mode][4: src]
      Mode = 0:
            dest <- f(dest, src)
      Mode = 1: [size: imm]
            dest <- f(dest, imm)
      Mode = 2: [address]
            dest <- f(dest, M[address])
      Mode = 3: [address]
            M[address] <- f(M[address], src)
      Mode = 4: [address]   [size: imm]
            M[address] <- f(M[address], imm)
      Else UND
      (dh and sh mark AH, BH, CH, or DH for dest or src)
```

MOVcc – Conditional Move

Condition	OP Code	Condition	OP Code	Condition	OP Code	Condition	OP Code
Z	08 00	NZ	08 01				
S	08 02	NS	08 03	B	08 0a	L	08 0e
P	08 04	NP	08 05	BE	08 0b	LE	08 0f
O	08 06	NO	08 07	A	08 0c	G	08 10
C	08 08	NC	08 09	AE	08 0d	GE	08 11

Description:

Conditionally copies a value from some source to a destination if the specified condition is met.

Does not support memory-to-memory transfers (in accordance with modern processor architectures).

Usage:

```
MOVcc r, imm/r/m

MOVcc m, imm/r
```

Flags Affected:

None.

Format:

```
[MOVcc]    [4: dest][2: size][1:dh][1: sh]    [4: mode][4: src]
       Mode = 0:
              dest <- f(dest, src)
       Mode = 1: [size: imm]
              dest <- f(dest, imm)
       Mode = 2: [address]
              dest <- f(dest, M[address])
       Mode = 3: [address]
              M[address] <- f(M[address], src)
       Mode = 4: [address]    [size: imm]
              M[address] <- f(M[address], imm)
       Else UND
       (dh and sh mark AH, BH, CH, or DH for dest or src)
```

XCHG – Exchange

OP Code:

09

Description:

Exchanges two values.

Does not support memory-to-memory transfers.

Usage:

```
XCHG r, r/m
```

```
XCHG m, r
```

Flags Affected:

None.

Format:

```
[XCHG]    [4: r1][2: size][1: r1h][1: mem]
      mem = 0: [1: r2h][3:][4: r2]
            r1 ← r2
            r2 ← r1
      mem = 1: [address]
            r1 ← M[address]
            M[address] ← r1
      (r1h and r2h mark AH, BH, CH, or DH for r1 or r2)
```

JMP – Jump

OP Code:

0a

Description:

Jumps execution to the provided address.

When specifying an operand size other than 64-bits, the value is zero-extended before being loaded into RIP.

Usage:

```
JMP imm/r/m
```

Flags Affected:

None.

Format:

```
[JMP]   [4: reg][2: size][2: mode]
     mode = 0:                 reg
     mode = 1:                 h reg (AH, BH, CH, or DH)
     mode = 2: [size: imm]     imm
     mode = 3: [address]       M[address]
```

Jcc – Conditional Jump

Condition	OP Code	Condition	OP Code	Condition	OP Code	Condition	OP Code
Z	0b 00	NZ	0b 01				
S	0b 02	NS	0b 03	B	0b 0a	L	0b 0e
P	0b 04	NP	0b 05	BE	0b 0b	LE	0b 0f
O	0b 06	NO	0b 07	A	0b 0c	G	0b 10
C	0b 08	NC	0b 09	AE	0b 0d	GE	0b 11
CXZ	0b 12	ECXZ	0b 13	RCXZ	0b 14		

Description:

Conditionally jumps execution to the provided address if the condition is met.

These instructions form the basis for all conditional branching.

When specifying an operand size other than 64-bits, the value is zero-extended before being loaded into RIP.

The conditions CXZ, ECXZ, and RCXZ are unique to Jcc and denote the current value of CX, ECX, or RCX being zero.

Usage:

```
Jcc imm/r/m
```

Flags Affected:

None.

Format:

As JMP.

LOOP/LOOPcc – Loop

Mnemonic	Additional Condition	OP Code
LOOP	N/A	`0c 00`
LOOPe	ZF = 1	`0c 01`
LOOPne	ZF = 0	`0c 02`

Description:

Decrements RCX, ECX, or CX (depending on operand size) and jumps to the provided address if the result is non-zero and the additional condition (if present) is true. Used for simple count-based iteration.

In CSX64 assembly, an immediate is treated as a 64-bit value (and thus will use RCX).

Care should be taken that the count register is not initially zero, as the pre-decrement would cause an underflow and ultimately perform 2^{64} loop iterations (which is very likely not what you intended). For this reason, it is common practice to use a Jrcxz variant to bypass the loop structure entirely in the case that the count starts at zero.

Usage:

```
LOOPcc imm/r/m
```

Flags Affected:

None.

Format:

As JMP.

CALL – Call

OP Code:

0d

Description:

Pushes the (64-bit) address of the next instruction onto the stack and jumps execution to the specified address.

This instruction is the backbone of all reasonable calling conventions.

Usage:

```
CALL imm/r/m
```

Flags Affected:

None.

Format:

As JMP.

RET – Return

OP Code:

0e

Description:

Pops a 64-bit value from the stack and jumps to that address.

Usage:

```
RET
```

Flags Affected:

None.

Format:

```
[RET]
```

PUSH – Push

OP Code:

0f

Description:

Pushes a value onto the stack. The size parameter determines the size of the value that is pushed.

Pushing 8-bit values is not allowed (because historically the stack is aligned on 16-bit boundaries). If this behavior is needed, it may be suitable to push a 16-bit value and disregard the 8 bits that weren't wanted.

In CSX64 assembly, pushing an immediate is treated as a 64-bit value. If this is not desired, you may decrement RSP and load the value onto the stack manually.

Usage:

```
PUSH imm/r/m
```

Flags Affected:

None.

Format:

```
[PUSH]    [4: reg][2: size][2: mode]
     mode = 0:                 reg
     mode = 1:                 h reg (AH, BH, CH, or DH)
     mode = 2: [size: imm]     imm
     mode = 3: [address]       M[address]
```

POP – Pop

OP Code:

10

Description:

Removes a `size` byte value from the stack.

Usage:

```
POP(:size) r
```

Flags Affected:

None.

Format:

```
[POP]   [4: dest][2: size][1:][1: mem]
      mem = 0:
            pop value
            dest ← value
      mem = 1: [address]
            pop value
            M[address] ← value
```

LEA – Load Effective Address

OP Code:

11

Description:

Loads a register with the address of a memory argument (without actually getting the memory at that address). If the register partition specified is too small, the value is truncated. For this reason, you will typically want to store the result in a full 64-bit register (which shouldn't be unusual as CX64 uses 64-bit addressing).

Because of the memory address format, this operation is capable of adding an immediate and two registers together simultaneously, where one register may be negative, and both may be multiplied by small powers of 2. This makes LEA a powerful tool for computing integral arithmetic.

Does not support 8-bit addressing (i.e. you can't store the result to an 8-bit register).

Usage:

```
LEA r, m
```

Flags Affected:

None.

Format:

```
[LEA]   [4: dest][2: size][2:]   [address]
     dest ← address
```

ADD – Add

OP Code:

12

Description:

Adds the source (second argument) to the destination (first argument).

Usage:

```
ADD r, imm/r/m

ADD m, imm/r
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the result is negative (high bit set) and cleared otherwise.

PF is set if the result has even parity in the low 8 bits and cleared otherwise.

CF is set if the addition caused a carry out from the high bit and cleared otherwise.

AF is set if the addition caused a carry out from the low 4 bits and cleared otherwise.

OF is set if the addition resulted in arithmetic under/overflow and cleared otherwise.

Format:

```
[ADD]   [4: dest][2: size][1:dh][1: sh]   [4: mode][4: src]
     Mode = 0:
          dest ← dest + src
     Mode = 1: [size: imm]
          dest ← dest + imm
     Mode = 2: [address]
          dest ← dest + M[address]
     Mode = 3: [address]
          M[address] ← M[address] + src
     Mode = 4: [address]   [size: imm]
          M[address] ← M[address] + imm
     Else UND
     (dh and sh mark AH, BH, CH, or DH for dest or src)
```

SUB – Subtract

OP Code:

13

Description:

Subtracts the source (second argument) from the destination (first argument).

SUB can and should be used in place of CMP if the result is needed.

Usage:

```
SUB r, imm/r/m
SUB m, imm/r
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the result is negative (high bit set) and cleared otherwise.

PF is set if the result has even parity in the low 8 bits and cleared otherwise.

CF is set if the subtraction caused a borrow from the high bit and cleared otherwise.

AF is set if the subtraction caused a borrow from the low 4 bits and cleared otherwise.

OF is set if the subtraction resulted in arithmetic under/overflow and cleared otherwise.

Format:

As Add.

MUL – Unsigned Multiply

OP Code:

14

Description:

Computes the full product of multiplying two unsigned values, which will have a width of twice the operand size. The first value to multiply must be in a low partition of RAX, but second value to multiply by may be an imm or register/memory value.

The low bits of the product are stored in the partition of RAX, but the high bits are stored in a partition of RDX (except in the case of 8-bit multiplies, where the high 8 bits are stored in AH).

Usage:

```
MUL imm/r/m
```

Operation:

```
byte:        AX ← AL  * val
word:     DX:AX ← AX  * val
dword: EDX:EAX ← EAX * val
qword: RDX:RAX ← RAX * val
```

Flags Affected:

OF and CF are both set if the high half of the product is non-zero and cleared otherwise.

SF, ZF, AF, and PF are undefined.

Format:

```
[MUL]    [4: reg][2: size][2: mode]
     mode = 0:                 reg
     mode = 1:                 h reg (AH, BH, CH, or DH)
     mode = 2: [size: imm]     imm
     mode = 3: [address]       M[address]
```

IMUL – Signed Multiply

Form	**OP Code**
Unary	`15 0`
Binary	`15 1`
Ternary	`15 2`
Form	**Description**
Unary	As MUL except it performs signed multiplication.
Binary	Multiplies the destination by the source.
Ternary	Multiplies a register or memory value by an imm (args 2 and 3) and stores the result in a register (arg 1).
Form	**Usage**
Unary	`IMUL imm/r/m`
Binary	`IMUL r, imm/r/m` `IMUL m, imm/r`
Ternary	`IMUL r, r/m, imm`
Form	**Flags Affected**
Unary	OF and CF are both set if the high half of the product contains significant bits (i.e. the truncated value is not equal to the full product) and cleared otherwise. SF, ZF, AF, and PF are undefined.
Binary	
Ternary	
Form	**Format**
Unary	As MUL.
Binary	As ADD.
Ternary	`[IMUL]    [4: dest][2: size][1:dh][1: mem]    [size: imm]`
	`mem = 0: [1: sh][3:][4: src]`
	`dest ← reg * imm`
	`mem = 1: [address]`
	`dest ← M[address] * imm`
	`(dh and sh mark AH, BH, CH, or DH for dest or src)`

DIV – Unsigned Divide

OP Code:

16

Description:

Computes the full quotient and remainder of the division of two unsigned values. The numerator has twice the width of the operand size and must already be loaded into a partition of RDX:RAX (described below). The denominator however, may be an immediate, register, or memory value.

The quotient is stored in a partition of RAX. The remainder is stored in a partition of RDX.

Dividing by zero or producing a quotient that cannot be stored in the operand register produces an arithmetic error.

Usage:

```
DIV imm/r/m
```

Operation:

```
byte:
      AX / val
      AL ← quotient, AH ← remainder
word:
      DX:AX / val
      AX ← quotient, DX ← remainder
dword:
      EDX:EAX / val
      EAX ← quotient, EDX ← remainder
qword:
      RDX:RAX / val
      RAX ← quotient, RDX ← remainder
```

Flags Affected:

CF, OF, SF, ZF, AF, and PF are undefined.

Format:

As MUL.

IDIV – Signed Divide

OP Code:

17

Description:

As DIV except that it performs signed division.

If the numerator or denominator is negative, the remainder is platform-dependent.

Usage:

```
IDIV imm/r/m
```

Operation:

```
byte:
      AX / val
      AL ← quotient, AH ← remainder
word:
      DX:AX / val
      AX ← quotient, DX ← remainder
dword:
      EDX:EAX / val
      AX ← quotient, DX ← remainder
qword:
      RDX:RAX / val
      AX ← quotient, DX ← remainder
```

Flags Affected:

CF, OF, SF, ZF, AF, and PF are undefined.

Format:

As MUL.

SHL – Shift Left

OP Code:

18

Description:

Shifts the destination to the left by a number of bits.

The amount to rotate by is given by an 8-bit unsigned integer, which is first masked to 5 bits (6 in the case of 64-bit rotates). Shifting by 0 is no-op.

Vacated bits are filled with 0. This operation is identical to SAL but is a distinct operation to specify intent.

Usage:

```
SHL r, imm/r/m 8
```

```
SHL m, imm/r 8
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the result is negative (high bit set) and cleared otherwise.

PF is set if the result has even parity in the low 8 bits and cleared otherwise.

CF is set to the last bit shifted out (undefined if shift amount exceeds size (in bits) of operand).

OF is undefined if the shift value is not 1, otherwise it’s set if the high 2 bits of the original value were different and cleared otherwise (i.e. sign change).

AF is undefined.

Format:

As ADD except the source is 8-bit regardless of size field.

SHR – Shift Right

OP Code:

19

Description:

Shifts the destination to the right by a number of bits.

The amount to rotate by is given by an 8-bit unsigned integer, which is first masked to 5 bits (6 in the case of 64-bit rotates). Shifting by 0 is no-op.

Vacated bits are filled with 0. For non-negative numbers, this instruction is equivalent to SAR.

Usage:

```
SHR r, imm/r/m 8

SHR m, imm/r 8
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the result is negative (high bit set) and cleared otherwise.

PF is set if the result has even parity in the low 8 bits and cleared otherwise.

CF is set to the last bit shifted out (undefined if shift amount exceeds size (in bits) of operand).

OF is undefined if the shift value is not 1, otherwise it's set to the high bit of the original value.

AF is undefined.

Format:

As ADD except the source is 8-bit regardless of size field.

SAL – Shift Arithmetic Left

OP Code:

1a

Description:

Shifts the destination to the left by a number of bits.

The amount to rotate by is given by an 8-bit unsigned integer, which is first masked to 5 bits (6 in the case of 64-bit rotates). Shifting by 0 is no-op.

Vacated bits are filled with 0. This operation is identical to SHL but is a distinct operation to specify intent.

Usage:

```
SAL r, imm/r/m 8
SAL m, imm/r 8
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the result is negative (high bit set) and cleared otherwise.

PF is set if the result has even parity in the low 8 bits and cleared otherwise.

CF is set to the last bit shifted out (undefined if shift amount exceeds size (in bits) of operand).

OF is undefined if the shift value is not 1, otherwise it's set if the high 2 bits of the original value were different and cleared otherwise (i.e. sign change).

AF is undefined.

Format:

As ADD except the source is 8-bit regardless of size field.

SAR – Shift Arithmetic Right

OP Code:

1b

Description:

Shifts the destination to the right by a number of bits.

The amount to rotate by is given by an 8-bit unsigned integer, which is first masked to 5 bits (6 in the case of 64-bit rotates). Shifting by 0 is no-op.

Vacated bits are filled with the sign bit. Because of this, SAR can be thought of as signed division by a power of 2, followed by a round trip toward negative infinity (this also means an arithmetic right shift on a negative value will never yield zero). For non-negative numbers, this instruction is equivalent to SHR.

Usage:

```
SAR r, imm/r/m 8
```

```
SAR m, imm/r 8
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the result is negative (high bit set) and cleared otherwise.

PF is set if the result has even parity in the low 8 bits and cleared otherwise.

CF is set to the last bit shifted out (undefined if shift amount exceeds size (in bits) of operand).

OF is undefined if the shift value is not 1, otherwise it’s cleared.

AF is undefined.

Format:

As ADD except the source is 8-bit regardless of size field.

ROL – Rotate Left

OP Code:

1c

Description:

Rotates the destination to the left by a number of bits.

The amount to rotate by is given by an 8-bit unsigned integer, which is first masked to 5 bits (6 in the case of 64-bit rotates). Rotating an n-bit value is performed in modulo-n. Shifting by 0 is no-op.

Usage:

```
ROL r, imm/r/m 8
```

```
ROL m, imm/r 8
```

Flags Affected:

CF is set to the last bit rotated out of the high bit.

OF is undefined if the shift value is not 1, otherwise it is set to the exclusive OR of CF (after the rotate) and the high bit of the result.

Format:

As ADD except the source is 8-bit regardless of size field.

ROR – Rotate Right

OP Code:

1d

Description:

Rotates the destination to the right by a number of bits.

The amount to rotate by is given by an 8-bit unsigned integer, which is first masked to 5 bits (6 in the case of 64-bit rotates). Rotating an n-bit value is performed in modulo-n. Shifting by 0 is no-op.

Usage:

```
ROR r, imm/r/m 8

ROR m, imm/r 8
```

Flags Affected:

CF is set to the last bit rotated out of the low bit.

OF is undefined if the shift value is not 1, otherwise it is set to the exclusive OR of the high two bits of the result.

Format:

As ADD except the source is 8-bit regardless of size field.

RCL – Rotate Carry Left

OP Code:

1e

Description:

Rotates the destination to the left by a number of bits.

The operation is performed as if iteratively, where the current value of CF is shifted into the vacated low bit of the value, and the high bit of the value is shifted into CF.

The amount to rotate by is given by an 8-bit unsigned integer, which is first masked to 5 bits (6 in the case of 64-bit rotates). Rotating an n-bit value is performed in modulo-n+1 (n-bit value plus 1 bit for CF). Shifting by 0 is no-op.

Usage:

```
RCL r, imm/r/m 8

RCL m, imm/r 8
```

Flags Affected:

CF is set to the last bit rotated out of the high bit.

OF is undefined if the shift value is not 1, otherwise it is set to the exclusive OR of CF (after the rotate) and the high bit of the result.

Format:

As ADD except the source is 8-bit regardless of size field.

RCR – Rotate Carry Right

OP Code:

1f

Description:

Rotates the destination to the right by a number of bits.

The operation is performed as if iteratively, where the current value of CF is shifted into the vacated high bit of the value, and the low bit of the value is shifted into CF.

The amount to rotate by is given by an 8-bit unsigned integer, which is first masked to 5 bits (6 in the case of 64-bit rotates). Rotating an n-bit value is performed in modulo-n+1 (n-bit value plus 1 bit for CF). Shifting by 0 is no-op.

Usage:

```
RCR r, imm/r/m 8
```

```
RCR m, imm/r 8
```

Flags Affected:

CF is set to the last bit rotated out of the high bit.

OF is undefined if the shift value is not 1, otherwise it is set to the exclusive OR of the high two bits of the result.

Format:

As ADD except the source is 8-bit regardless of size field.

AND – Bitwise And

OP Code:

20

Description:

ANDs the destination with the source.

AND can and should be used in place of TEST if the result is needed.

Usage:

```
AND r, imm/r/m
```

```
AND m, imm/r
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the result is negative (high bit set) and cleared otherwise.

PF is set if the result has even parity in the low 8 bits and cleared otherwise.

OF and CF are cleared.

AF is undefined.

Format:

As ADD.

OR – Bitwise Or

OP Code:

21

Description:

ORs the destination with the source.

Usage:

```
OR r, imm/r/m
```

```
OR m, imm/r
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the result is negative (high bit set) and cleared otherwise.

PF is set if the result has even parity in the low 8 bits and cleared otherwise.

OF and CF are cleared.

AF is undefined.

Format:

As ADD.

XOR – Bitwise Exclusive Or

OP Code:

22

Description:

XORs the destination with the source.

Usage:

```
XOR r, imm/r/m
```

```
XOR m, imm/r
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the result is negative (high bit set) and cleared otherwise.

PF is set if the result has even parity in the low 8 bits and cleared otherwise.

OF and CF are cleared.

AF is undefined.

Format:

As ADD.

INC – Increment

OP Code:

23

Description:

Equivalent to using ADD with a value of 1 except that CF is preserved.

Usage:

```
INC r/m
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the result is negative (high bit set) and cleared otherwise.

PF is set if the result has even parity in the low 8 bits and cleared otherwise.

AF is set if the addition caused a carry out from the low 4 bits and cleared otherwise.

OF is set if the addition resulted in arithmetic under/overflow and cleared otherwise.

Format:

```
[INC]   [4: dest][2: size][1: dh][1: mem]
      mem = 0:
            dest ← dest + 1
      mem = 1: [address]
            M[address] ← M[address] + 1
      (dh marks AH, BH, CH, or DH for dest)
```

DEC – Decrement

OP Code:

24

Description:

Equivalent to using SUB with a value of 1 except that CF is preserved.

Usage:

```
DEC r/m
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the result is negative (high bit set) and cleared otherwise.

PF is set if the result has even parity in the low 8 bits and cleared otherwise.

AF is set if the subtraction caused a borrow from the low 4 bits and cleared otherwise.

OF is set if the subtraction resulted in arithmetic under/overflow and cleared otherwise.

Format:

As INC.

NEG – Negate

OP Code:

25

Description:

Negates the destination. Equivalent to SUB with a destination (LHS) of zero and a source (RHS) of the value to negate.

Because this instruction modifies the flags in the same way was SUB, arithmetic conditionals can be used after this instruction to correctly compare 0 (LHS) to the original value (RHS), or (by carrying the in/equality through to the negative range) to compare the (negated) result (LHS) to 0 (RHS).

Additionally, the only way OF can be set is if both the original value and the result have their high bit set (i.e. both negative). Because of this, OF can be used to test for a failed 2's complement negation.

Usage:

```
NEG r/m
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the result is negative (high bit set) and cleared otherwise.

PF is set if the result has even parity in the low 8 bits and cleared otherwise.

CF is set if the subtraction caused a borrow from the high bit and cleared otherwise.

AF is set if the subtraction caused a borrow from the low 4 bits and cleared otherwise.

OF is set if the subtraction resulted in arithmetic under/overflow and cleared otherwise.

Format:

As INC.

NOT – Bitwise Not

OP Code:

26

Description:

Performs a bitwise NOT on the destination.

Usage:

```
NOT r/m
```

Flags Affected:

None.

Format:

As INC.

CMP – Compare

OP Code:

27 (CMP), 28 (CMPZ)

Description:

After this operation, the flags are set in such a way as to satisfy arithmetic conditionals, where the destination is the left operand and the source is the right operand.

This operation is identical to SUB, but the result is discarded.

SUB can and should be used in place of CMP if the result is needed.

Because comparing a value to zero is frequently needed, most instruction sets also contain an instruction specifically for this purpose (CMPZ in CSX64), though assemblers typically don't offer direct access to this instruction.

Usage:

```
CMP r, imm/r/m
```

```
CMP m, imm/r
```

Flags Affected:

As SUB.

Format:

As SUB.

TEST – Logical Test

OP Code:

29

Description:

As AND, but the result is discarded.

AND can and should be used in place of TEST if the result is needed.

Usage:

```
TEST r, imm/r/m
```

```
TEST m, imm/r
```

Flags Affected:

As AND.

Format:

As AND.

BSWAP – Byte Swap

OP Code:

2a

Description:

Reverses the byte order of the specified value, effectively translating between big/little endian. This is frequently used in networking, where big-endian is very common.

Using this operation on an 8-bit value is no-op.

Usage:

```
BSWAP r/m
```

Flags Affected:

None.

Format:

As INC.

BEXTR – Bitfield Extract

OP Code:

2b

Description:

Extracts a contiguous series of bits from a value. The `size` parameter determines the size of the bitfield source. The bitfield extracted is determined by an additional 16-bit argument, where the high 8 bits determine the position of the lowest bit of the bitfield (zero being the low bit of the source), and the low 8 bits determine the length (in bits) of the bitfield.

Both the position and the size of the bitfield are performed in modulo-n, where n is the size of the source (in bits).

Attempting to extract bits beyond the size of the source is not an error, and the overflowing section is ignored.

Usage:

```
BEXTR r, imm/r/m
```

```
BEXTR m, imm/r
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

AF, SF, and PF are undefined.

All other (public) flags are cleared.

Format:

As ADD except the source is 16-bit regardless of size field.

BLSI – Binary Lowest-Set Isolate

OP Code:

2c

Description:

Isolates the lowest-order set bit, leaving it in its current position. All other bits are cleared. If the source contains no set bits, the result is zero.

Usage:

```
BLSI r/m
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the high bit (sign bit) of the result is set and cleared otherwise.

CF is set if the original value was nonzero and cleared otherwise.

OF is cleared.

AF and PF are undefined.

Format:

As INC.

BLSMSK – Binary Lowest-Set Mask

OP Code:

2d

Description:

Extracts a bitmask containing only the low-order bits up to and including the lowest-set bit. If the source is zero (i.e. no set bits), the result is all 1’s.

Usage:

```
BLSMSK r/m
```

Flags Affected:

SF is set if the high bit (sign bit) of the result is set and cleared otherwise.

CF is set if the original value was zero and cleared otherwise.

ZF and OF are cleared.

AF and PF are undefined.

Format:

As INC.

BLSR – Binary Lowest-Set Reset

OP Code:

2e

Description:

Clears the lowest-order set bit of the source. If source contains no set bits, the result is zero.

Usage:

```
BLSR r/m
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the high bit (sign bit) of the result is set and cleared otherwise.

CF is set if the original value was zero and cleared otherwise.

OF is cleared.

AF and PF are undefined.

Format:

As INC.

ANDN – Bitwise And Not

OP Code:

2f

Description:

Computes the bitwise `and` of two unsigned integers, with the second being inverted before use.

Usage:

```
ANDN(:size) r, imm/r/m
```

```
ANDN(:size) m, imm/r
```

Flags Affected:

ZF is set if the result is zero and cleared otherwise.

SF is set if the high bit (sign bit) of the result is set and cleared otherwise.

OF and CF are cleared.

AF and PF are undefined.

Format:

As ADD.

BT / BTS / BTR / BTC – Bit Test

OP Code	Name	Description
`30 00`	BT – Bit Test	Extracts a bit from a value and loads it to CF
`30 01`	BTS – Bit Test and Set	As BT except the destination's bit is set
`30 02`	BTR – Bit Test and Reset	As BT except the destination's bit is cleared
`30 03`	BTC – Bit Test and Complement	As BT except the destination's bit is flipped

Description:

Given an integral value from a register or memory (first arg), extracts the nth bit (where n is second arg) and stores it in CF, where the least significant bit is index zero. The extraction is performed in modulo-size where size is the number of bits in the value to test (first arg). If the destination (first arg) is a memory value, the result is as if the memory value were loaded to a register and the register version was used (i.e. it accounts for endianness internally).

BT does not modify the destination, but BTS, BTR, and BTC do.

Usage:

```
BT r/m, imm/r 8
```

Flags Affected:

CF is loaded with the extracted bit.

OF, SF, AF, and PF are undefined.

Format:

As ADD except the source is 8-bit regardless of size field.

CWD/CDQ/CQO/CBW/CWDE/CDQE – Convert Word Size

Mnemonic	OP Code	Action
CWD	`31 00`	Sign extend AX into DX:AX
CDQ	`31 01`	Sign extend EAX into EDX:EAX
CQO	`31 02`	Sign extend RAX into RDX:RAX
CBW	`31 03`	Sign extend AL into AX
CWDE	`31 04`	Sign extend AX into EAX
CDQE	`31 05`	Sign extend EAX into RAX

Description:

Sign-extends a value in a partition of RAX into a larger partition of RAX or into an equal-size partition of RDX. These instructions are typically used alongside unary IMUL and IDIV, which require the first operand to be in a partition of RDX:RAX or AX. A typical use for the forms that just extend into a larger partition of RAX is in normal sign extension, as an alternative to the memory version MOVSZ.

Usage:

```
CWD/CDQ/DQO
```

Flags Affected:

None.

Format:

```
[op]
```

MOVZX/MOVSX – Move Zero/Sign Extended

Mnemonic	OP Code	Action
MOVZX	`32 00`	Load a zero-extended value from memory.
MOVSX	`32 01`	Load a sign-extended value from memory.

Description:

Loads a zero or sign extended value from memory. Complying with standard x86_64, not all combinations of sizes are allowed (allowed size conversions are described below).

If a size combination is needed but not provided by this instruction, unsigned values can simply have the full register zeroed and then load the value via a normal MOV. For signed values, use of the CBW family can facilitate the conversion (e.g. sign extend 32-bit to 64-bit could be performed as a normal MOV followed by CDQE).

Usage:

```
MOVZX/MOVSX r, m
```

Flags Affected:

None.

Format:

```
[op]   [4: dest][4: mode]   [1: mem][1: sh][2:][4: src]

       mode = 0: 16 <- 8  Zero

       mode = 1: 16 <- 8  Sign

       mode = 2: 32 <- 8  Zero

       mode = 3: 32 <- 16 Zero

       mode = 4: 32 <- 8  Sign

       mode = 5: 32 <- 16 Sign

       mode = 6: 64 <- 8  Zero

       mode = 7: 64 <- 16 Zero

       mode = 8: 64 <- 8  Sign

       mode = 9: 64 <- 16 Sign

       else UND

       (sh marks AH, BH, CH, or DH for src)
```

Appendix

Converting Binary Integers

Because all modern computing is built upon the foundation of binary integers, you'll need to know how to encode them for the processor to understand. Whereas decimal is base 10 (and each digit position is 10 times bigger than the last), binary is base 2 (and each digit position is 2 times bigger than the last):

Decimal				Binary			
10^3	10^2	10^1	10^0	2^3	2^2	2^1	2^0
1000	100	10	1	8	4	2	1

To get the value of a decimal number, you take each digit and multiply by its weight (position), then add them together.

4107_{10}			1011_2		
Digit	**Weight**	**Digit * Weight**	**Digit**	**Weight**	**Digit * Weight**
4	1000	4000	1	8	8
1	100	100	0	4	0
0	10	0	1	2	2
7	1	7	1	1	1
		Sum: 4107			**Sum: 11**

So now you can turn a binary number into decimal. A neat trick, but if you really want to get into the nitty-gritty of computers, you'll need to be able to do the reverse (and get pretty fast at it). An easy method to do this is to divide by 2 and take the remainder, which will always be either a 0 or a 1. These remainders, when read backwards, will be the binary value:

753_{10}			172_{10}		
Value	**Quotient**	**Remainder**	**Value**	**Quotient**	**Remainder**
753 / 2	376	1	172 / 2	86	0
376 / 2	188	0	86 / 2	43	0
188 / 2	94	0	43 / 2	21	1
94 / 2	47	0	21 / 2	10	1
47 / 2	23	1	10 / 2	5	0
23 / 2	11	1	5 / 2	2	1
11 / 2	5	1	2 / 2	1	0
5 / 2	2	1	1 / 2	0	1
2 / 2	1	0			
1 / 2	0	1			
	Result:	**10 1111 0001_2**		**Result:**	**1010 1100_2**

Alternatively, you could subtract powers of 2, keeping track of which ones fit. The moral of the story is practice, practice, practice. Knowing your powers of 2 is absolutely essential, especially in machine language and assembly.

Proof of 2's Complement

The seemingly magical properties of 2's complement can be a bit hard to understand at first. For instance, why would signed and unsigned addition and subtraction be the same when using 2's complement to represent signed integers? What genius madman, in a fevered binary dream of passion decided to apply bitwise not and incrementation?

Well, you'll be happy to know that this property of working the same for both signed and unsigned addition (and subtraction, since that's the same thing as adding a negative and we're deriving a scheme for taking the negative) was actually the whole idea. Indeed, 2's complement was defined to work with unsigned addition (and subtraction). It was no happy coincidence. Observe:

`v + -v = 0`	We begin with a simple question: `v + what = 0` under unsigned addition
`v + ~v + 1 = 0`	Well, `v + ~v` would be all 1's, and adding 1 would overflow back to 0
`v + -v = v + ~v + 1`	Since these both equal 0, we can equate them
`v + -v + -v = v + ~v + 1 + -v`	We can then add `-v` to both sides
`-v = ~v + 1`	From the first equation: `v + -v = 0`, and we have our answer: `-v = ~v + 1`

Consider this the computer science equivalent of being told there is no Santa Claus. It was algebra all along.

The Heap

In some cases, it is necessary to create a new variable in memory that will outlast its position in the stack. For instance, you may need a function to allocate memory to represent a customer. Now, with only a standard stack this is an impossible task. If the function creates the variable on the stack, pushing and popping will destroy it. If it puts in in static variables or a pre-allocated array, there can only be as many as you created “slots” to put them in. But what if you don’t know how many “slots” to make?

This is the question that is answered by what’s known as dynamic memory allocation. Essentially, we create some pool of memory that can be allocated as needed, typically by a function call. The important thing is that the memory is not in static variables or on the stack. It is somewhere else: the heap. Typically, any modern language would provide a library function to do this, but you can absolutely make your own (potentially specialized) algorithm to do this.

Historically (and carried into modern systems), the heap begins just after the program segment and grows upwards similarly to a stack. This means that the stack and heap are on opposite sides of address space and grow towards one another. This means that either one of them can grow to fill potentially all of the provided space, so long as they don’t cross one another.

Encoding IEEE 754 64-bit Floating-Point Values