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

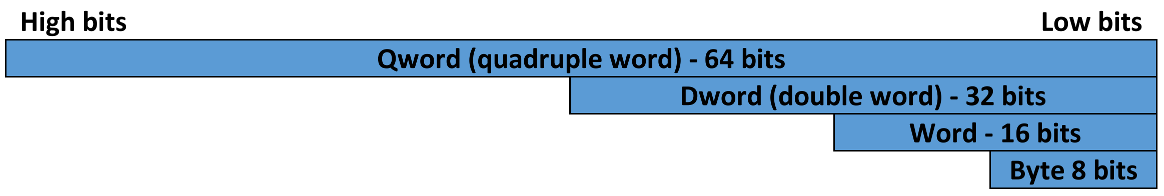
Registers

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.

The CSX64 processor contains 16 64-bit general-purpose registers (which will be referred to as $0-$15 in this manual in accordance with the CSX64 assembly language). An executing program will spend the vast majority of its time managing the contents of these general-purpose registers, which can be used for processing both integral and floating-point computations (in modern processors floating-point values are stored in special registers on the FPU – floating-point unit – but CSX64 ignores this and makes the general-purpose registers more general-purpose). Processors also contain several other registers for managing how the processor itself functions (e.g. execution pointer, status flags, etc.), but user access to those registers is usually very restricted (for good reason).

Register Subdivisions

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


High bits
Low bits
Qword (quadruple word) - 64 bits
Dword (double word) - 32 bits
Word - 16 bits
Byte 8 bits

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 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.

Because having 4 subdivisions is desirable because it's a power of 2, back in the 32-bit ye olden days there was an additional partition that was the high 8 bits of the 16-bit segment. Nowadays, while this is still a usable partition, it is not supported by some processor instructions and is essentially obsolescent. CSX64 does not include this partition at all.

Flags Register

The processor itself needs to keep track of key pieces of status information. The majority 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 Register			
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	Adjust Flag
5	0x0020		Reserved
6	0x0040	ZF	Zero Flag
7	0x0080	SF	Sign Flag
8	0x0100	TF	Trap Flag (currently not used in CSX64)
9	0x0200	IF	Interrupt Enable Flag (currently not used in CSX64)
10	0x0400	DF	Direction Flag (currently not used in CSX64)
11	0x0800	OF	Overflow Flag
12-31	…		
32	0x0000 0001 0000 0000	FSF	File System Flag
33-63	…		

The flags in orange are reserved (in the real Intel x86_64 standard most of flags 12-31 are not reserved and have real meanings, but they are not used by CSX64 and are thus omitted and marked as reserved). While it is legal to read/write to 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. In short: do not base your code on reserved flags.

The green flags are reserved in the Intel x86_64 standard but are used by CSX64 for its own purposes. Executing client code cannot modify these flags by any means. They are meant to be used as restraints on client code that are provided by the virtual operating system. That said, external code (e.g. system calls) is free to modify them, though this is considered in poor taste.

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). In CSX64 this task is managed by $14 and $15 respectively. Upon program initialization, $14 and $15 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, $15 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 operations assume that $15 points to the top of the stack. If this is not the case (i.e. you have destroyed the stack by modifying $15), then you should not use these commands at all.

$15 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*** $15 by 128.

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.

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 should 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 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 $\texttt{a.etc*10}^{\texttt{b}}$, 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 write in CSX64 machine code will be able to run on any machine that can run the interpreter.

Machine code is ***very*** 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. 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])   ([64: 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 operation's machine code specification will detail when it should or should not be provided.

Memory Address Format

In CSX64, memory addresses have a width of 64 bits, and are always in a standard format:

```
[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	`Z = 1`
NZ	NE	Not Zero / Not Equal		`Z = 0`
P	PE	Parity / Parity Even		`P = 1`
NP	PO	Not Parity / Parity Odd		`P = 0`
O		Overflow		`O = 1`
NO		Not Overflow		`O = 0`
C		Carry		`C = 1`
NC		Not Carry		`C = 0`

Label	Alias	Name	Signage	Condition
S		Sign		`S = 1`
NS		Not Sign		`S = 0`
A	NBE	Above	Unsigned	`CF = 0 and ZF = 0`
AE	NB	Above or Equal		`CF = 0`
B	NAE	Below		`CF = 1`
BE	NA	Below or Equal		`CF = 1 or ZF = 1`
G	NLE	Greater	Signed	`ZF = 0 and SF = OF`
GE	NL	Greater or Equal		`SF = OF`
L	NGE	Less		`SF != OF`
LE	NG	Less or Equal		`ZF = 1 or 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. In CSX64, if an instruction's documentation doesn't state that it modifies a particular flag, the value of said flag is unchanged (though this is not the case on *all* modern processor architectures).

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 (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 segment of $3, 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):

```
15 3c 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 32-bit segment of R3 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 $7. We thus need to use CMP's binary specification, which in turn uses the address specification:

Which gives us the binary form:

```
29 3d f0 70 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:

```
0a 00 08 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.

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:

```
15 3c 11 00 00 00

29 3d f0 70 c8 00 00 00 00 00 00 00

0a 00 08 e8 03 00 00
```

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

```
R3 += 17;

if (R3 == arr[R7]) 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) 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 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:32 $3, 17

cmp:32 $3, [200 + 4*$7]

jz:32 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.

CSX64 Assembly Language

CSX64 comes with a thorough, built-in assembler designed loosely around the standard Intel syntax (with several major changes to appeal to modern users, such as the naming conventions of the registers).

Before we describe the syntax of operations and assembler utilities, we need to go over some of the basics of CSX64 assembly language (much of which holds for all assembly languages).

Segments

In a typical assembly language, your program is broken up into several segments. 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 elsewhere), which could potentially speed up execution by a sizeable amount due to fewer cache misses. Additionally, operating systems typically instruct the processor to enforce certain read/write/execute privileges for certain ranges of address. Here we'll explore the most common segments.

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.

Rodata 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. a 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, 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)	A character enclosed in single quotes	`'A'`

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`

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 value is simply a value that is used to refer to a register. As discussed in the section on registers, CSX64 has 16 general-purpose registers, indexed 0-15. To refer to one of these registers, you designate the index of the register, preceded by "`$`" (e.g. `$3` would refer to register 3).

In fact, the index may be any instant imm, with the only limitations being that compound expressions must be parenthesized and the evaluated instant imm used as the register index must be integral and in the range 0-15. The following table summarizes these rules:

Expression	Referenced Register
`$3`	Register 3
`$abc`	Depends on value of abc, which must be instant
`$(3+1)`	Register 4
`$3+1`	Error – compound register index must be parenthesized
`$(abc*3 + 1)`	Depends on value of abc, which must be instant
`$(a + b)/2`	Error – compound register index must be parenthesized
`$-abc`	Error – compound register index must be parenthesized
`$(-abc)`	Depends on value of abc, which must be instant

The registers in a typical assembly language are labeled in a "standard" manner that is rather cumbersome and antiquated. The labels are in fact abbreviations for their original purposes, which are no longer relevant now that virtually every modern computer uses general-purpose registers instead. Because of this, CSX64 takes a more logical approach:

Register Labels in Assembly							
Standard x86_64				CSX64			
64 bits	Low 32	Low 16	Low 8	64 bits	Low 32	Low 16	Low 8
RAX	EAX	AX	AL	$0	$0	$0	$0
RBX	EBX	BX	BL	$1	$1	$1	$1
RCX	ECX	CX	CL	$2	$2	$2	$2
RDX	EDX	DX	DL	$3	$3	$3	$3
RSI	ESI	SI	SIL	$4	$4	$4	$4
RDI	EDI	DI	DIL	$5	$5	$5	$5
RBP	EBP	BP	BPL	$6	$6	$6	$6
RSP	ESP	SP	SPL	$7	$7	$7	$7
R8	R8D	R8W	R8B	$8	$8	$8	$8
R9	R9D	R9W	R9B	$9	$9	$9	$9
R10	R10D	R10W	R10B	$10	$10	$10	$10
R11	R11D	R11W	R11B	$11	$11	$11	$11
R12	R12D	R12W	R12B	$12	$12	$12	$12
R13	R13D	R13W	R13B	$13	$13	$13	$13
R14	R14D	R14W	R14B	$14	$14	$14	$14
R15	R15D	R15W	R15B	$15	$15	$15	$15

In standard x86_64 assembly, the register name you use indicates the size of the operation (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. 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 what size you wanted (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 solves this problem by instead attaching the size information to the operation (e.g. `mov:32 $5, [var]` or the unknown size case `mov:32 [var], 100`). This solves two problems: the inconsistent format of an unknown size issue, as well as the need to memorize the table of poorly-named register partitions used by x86_64. Even better, you need only specify the size once in CSX64, whereas you might need to specify it twice in x86_64 (e.g. `mov edi, eax`). This could be annoying if you ever decided to change the edi to rdi, as `mov rdi, eax` will not compile because the registers involved have conflicting sizes. CSX64's size system solves this more elegantly as `mov:32 $5, $0`, and conversion to a different size is trivial: `mov:64 $5, $0`.

If that weren't convenient enough, not specifying a size for an operation implies that you don't want a partition and thus defaults it to 64-bit mode (e.g. you can use `mov $5, $0` instead of `mov:64 $5, $0`). Since you'll be dealing with addresses and pointers frequently – which are 64-bit in CSX64 – this is a handy shorthand to be aware of.

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 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 the 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*($1+4) + 64]` is converted to `[8*$1 + 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*$5 - 8*$6 - 2*$5]` is converted to `[4*$5 - 8*$6]` 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*$5 - 8*$6 - 4*$5 + 2*$7]` is converted to `[-8*$6 + 2*$7]` and is thus legal).

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 simply 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 executable. Because the order of the source files in the result is undefined we also need to keep track of an offset that applies to every label defined in that file. 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 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 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.

Local Symbol

In addition to local labels, CSX64 assembly generalizes this locality to symbols as well. They function in exactly the same way, but refer to a normal imm, rather than strictly a label.

Address Difference

While it's true that a labels are not instant imms due to the missing offset, the difference of two labels in the same file 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.

Label Macros

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

Label Macro	Description
$	Expands to a pointer to the current position in the current segment.
$$	Expands to a pointer to the start of the current segment.

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.

Operation 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(:size) 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 and an optional explicit size parameter (which has different meanings depending on the operation). If the size parameter is not specified, it defaults to 64 (bits). The size parameter can be any instant imm, with the sole exception being that compound expressions (i.e. expressions involving any operators) must be parenthesized (this is in order to distinguish it from the argument section). Furthermore, the size parameter (or the operation if none is provided) must be separated from the argument section by white space. An operation assembles into exactly one machine code instruction (e.g. add, subtract, compare).

The argument section consists of zero or more comma-separated arguments for the selected operation. Note that if arguments are specified without an operation, the first argument will be taken as the operation, which is very likely not desirable.

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 line 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.

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.

Providing a size parameter has no effect.

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).

Providing a size parameter has no effect.

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

Declare (db, dw, dd, dq, dn)

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 left-to-right with a word size given by the directive used. DN uses the instruction's provided size code (which – as usual – defaults to 64-bit if not given explicitly). DN is not in the NASM standard and is added for compatibility with CSX64's symbolic instruction size system. DB, DW, DD, and DQ lock the word size to 8, 16, 32, and 64 bits respectively (regardless of any potential explicit instruction size code).

Additionally, if an argument is a line of text enclosed in double quotes, Declare will write each character in the string as if it were a character literal (i.e. `db "hello", 0` is identical to `db 'h', 'e', 'l', 'l', 'o', 0`). It should be noted that assembly languages do not append a null-terminator for you when emitting strings, which is why the above example added a null character explicitly. If your algorithm doesn't require a null terminator (e.g. direct use of sys_write), feel free to omit it. As a side note, writing a string with a larger-width declare variant (e.g. dd) in most assembly languages would actually merge characters together, as "abcd" in most assembly languages can be a single, 32-bit character literal 'abcd'. This is not currently the case in CSX64.

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 Emit 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 emission 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, resn)

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 64 bytes). 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.

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.

Providing a size parameter has no effect.

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 not case sensitive. 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. segments in a given assembly file must be contiguous).

Providing a size parameter has no effect.

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

Typically, you will want a function to use the CALL instruction, which will push the address of the next instruction onto the stack and then jump to the function. In which case it **must** return with a RET instruction (or equivalent stack manipulations). However, you could also push the desired return point and use a JMP operation to begin executing the function. 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).

Or you could instead 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).

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 = [$15 + 8], b = [$15 + 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. 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 in x86_64 or $14 in CSX64). The base pointer's purpose is to point to the top of the stack at the point where we began the process of pushing arguments and ultimately calling the function. Additionally, before beginning this process, we first push the **current value** of the base pointer onto the stack and then load the base pointer with the **new value** of the stack pointer. 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 the stack pointer with the current base pointer (since they were equal prior to when we started pushing arguments). Then you'd need to pop the old value of the base pointer off the stack and back into the base pointer register.

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 base of the stack rather than the top of the stack, which may change due to creating/destroying local variables or other stack frames (e.g. assuming 32-bit args: a = [$14 - 8], b = [$14 - 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 argument 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.

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 (assuming you're not loading the values into registers from memory) 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. Any calling convention, in addition to all the information on how to pass arguments, clean up, etc. should also define any registers that should not be modified (or at least no net change).

For instance, the base/stack pointer registers should have the same value after the function returns, 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 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.

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 it back after the return/cleanup (so you can ensure the value is not destroyed by the function).

Moreover, if you're about to use a call-safe register you need to store it somewhere before using it and then get that value back before returning (so that you had no net effect on it).

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 betting to use common methods because they have code that's already written using those). We'll now go over one of the best methods, which is both efficient and common.

1) Use CALL/RET.
2) Put arguments left-to-right in registers $1-$9.
    a. If the data type will not fit, consider taking a pointer.
        i. If modifications of the value at the call site are unacceptable, consider passing it on the stack or cloning it elsewhere and passing a pointer to the copy.
    b. If there are simply too many arguments, put the rest on the stack from right to left.
3) Place the return value in $0.

4) Registers $10-$15 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, you should thus assume $1 contains the number of arguments, and $2 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.

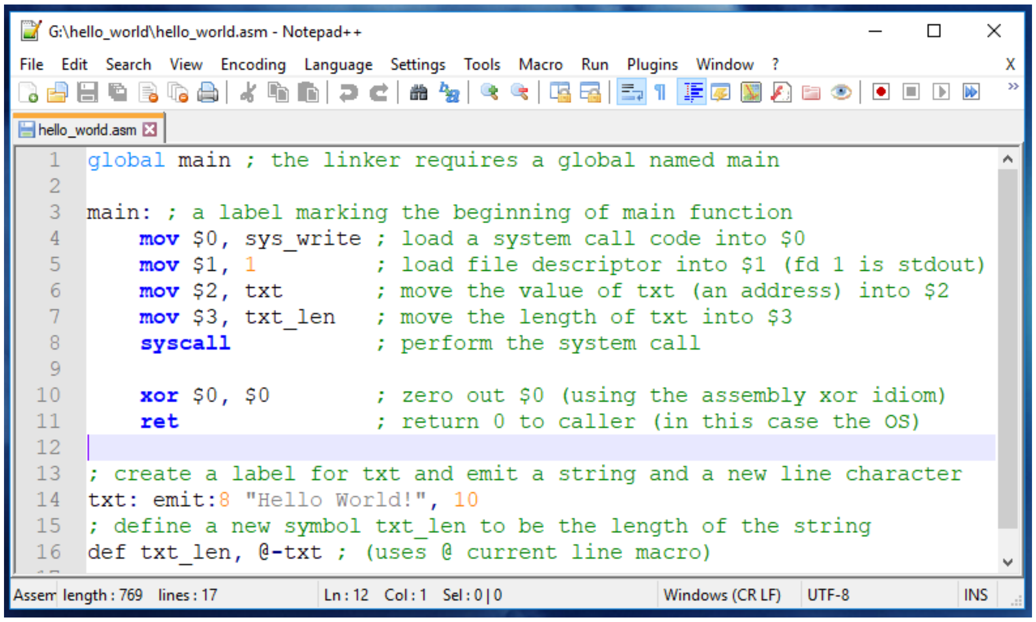
Well, you’ve made it this far, so how about we finally talk about how to do all this?

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).

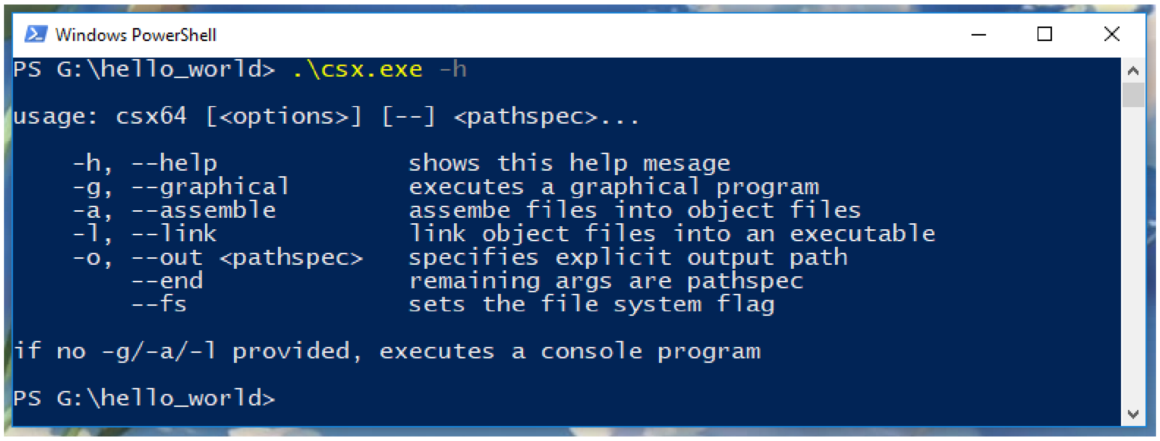
A few things to note:

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).

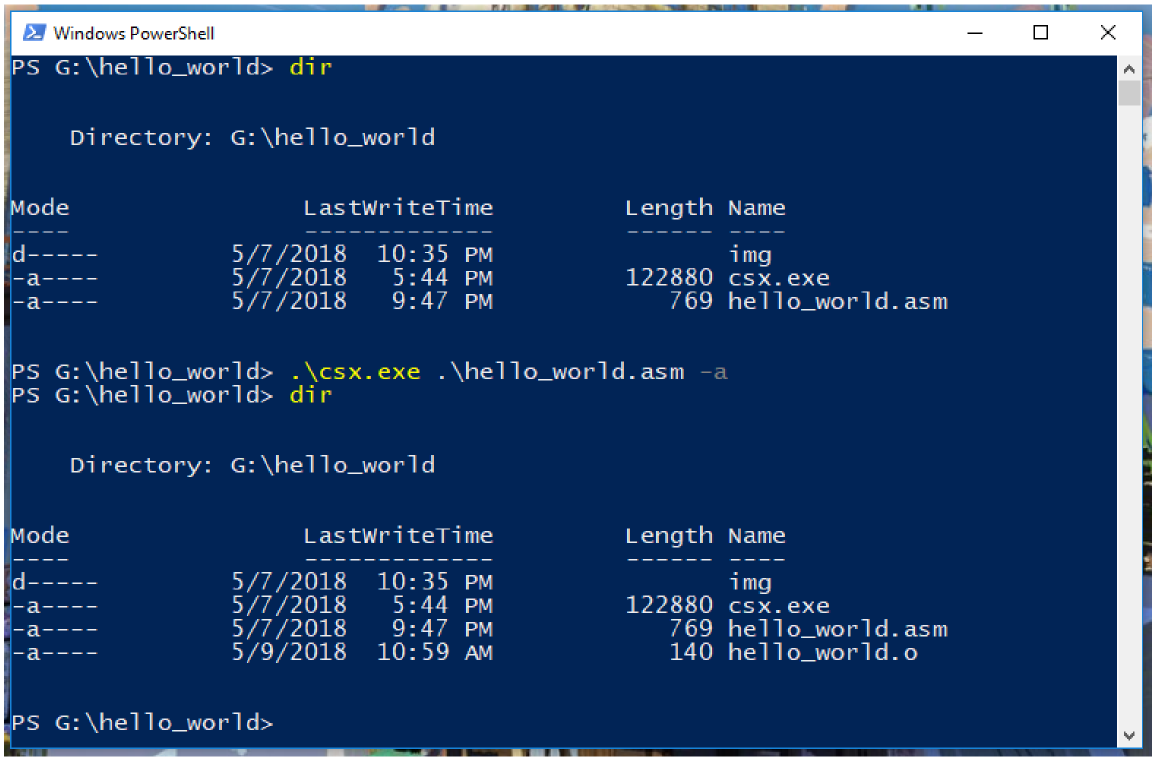
This will be the first occurrence of actually using the CSX64 application (csx.exe). Providing the -h option will list all of the available options:



```
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:

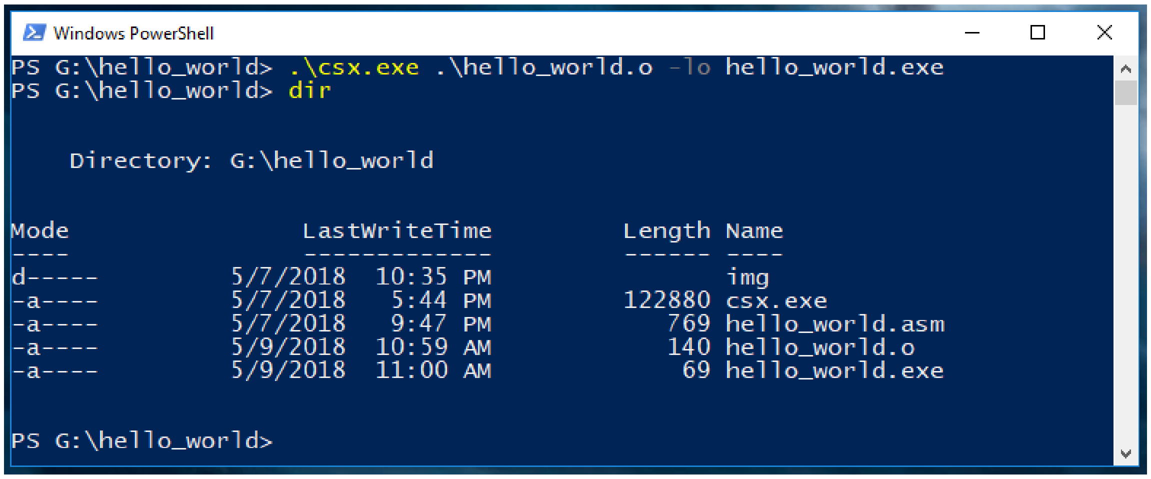


```
PS G:\hello_world> dir


    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:

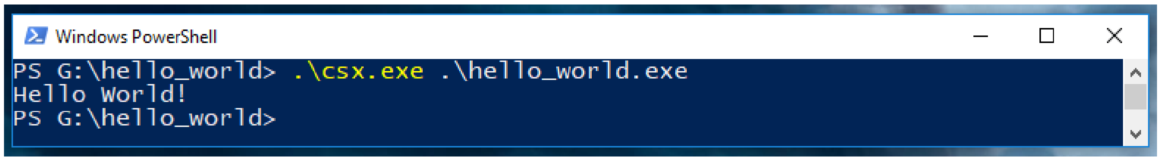

Windows PowerShell

```
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.

Operations List

Operation	Description
NOP	No operation
HLT	Stops processor execution
SYSCALL	Invokes virtual operating system utilities
MOVcc	Conditional move
SWAP	Swaps two values
ZX	Zero extend
SX	Sign extend
MUL	Unsigned multiply
IMUL	Signed multiply
DIV	Unsigned divide
IDIV	Signed divide
ADD	Add
SUB	Subtract
PLACEHOLDER	
PLACEHOLDER	
PLACEHOLDER	
SHL	Shift left
SHR	Shift right
SAL	Shift arithmetic left
SAR	Shift arithmetic right
ROL	Rotate left
ROR	Rotate right
AND	Bitwise AND
OR	Bitwise OR
XOR	Bitwise XOR
CMP	Compare
TEST	Test
INC	Increment
DEC	Decrement
NEG	Negate
NOT	Bitwise not
ABS	Absolute value
CMPZ	Compare to zero
LEA	Load effective address
Jcc	Conditional jump
FADD	Floating-point add
FSUB	Floating-point subtract
FMUL	Floating-point multiply
FDIV	Floating-point divide
FMOD	Floating-point mod
POW	Floating-point power
SQRT	Floating-point square root
EXP	Floating-point exponentiate

Operation	Description
LN	Floating-point natural log
FNEG	Floating-point negate
FABS	Floating-point absolute value
FCMPZ	Floating-point compare to zero
SIN	Floating-point sine
COS	Floating-point cosine
TAN	Floating-point tangent
SINH	Floating-point hyperbolic sine
COSH	Floating-point hyperbolic cosine
TANH	Floating-point hyperbolic tangent
ASIN	Floating-point arcsine
ACOS	Floating-point arccosine
ATAN	Floating-point arctangent
ATAN2	Floating-point binary arctangent
FLOOR	Floating-point floor
CEIL	Floating-point ceiling
ROUND	Floating-point round
TRUNC	Floating-point truncate
FCMP	Floating-point compare
FTOI	Floating-point to integer
ITOF	Integer to floating-point
PUSH	Push value onto stack
POP	Pop value from stack
CALL	Call function
RET	Return from function
BSWAP	Swap byte order of a value
BEXTR	Extract a bitfield
BSLI	Isolate the lowest-order set bit
BLSMSK	Get a mask containing up to and including the lowest-order set bit
BLSR	Clear the lowest-order set bit
ANDN	Bitwise and with a negated value
GETF	Get the flags register
SETF	Set the flags register
LOOP	Decrement register and jump if nonzero
FX	Floating-point extend
SLP	Sleep

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 CSX64.

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]
```

GETF – Get Flags

OP Code:

03

Description:

Loads the current flags data to the specified register or memory. Specifying a size code allows for fetching smaller widths from the flags register, which is good because in CSX64 assembly you'll typically only be interesting in the low 5 bits (i.e. the public flags).

Usage:

```
GETF r/m
```

Flags Affected:

None.

Format:

```
[GETF]   [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← Flags

     mem = 1: [address]
          M[address] ← Flags
```

SETF – Set Flags

OP Code:

04

Description:

Loads the flags register from the specified source. If using a size other than 64-bits, the high flags will be cleared.

Only the public flags will be modified by this instruction.

Usage:

```
SETF imm/r/m
```

Flags Affected:

None.

Format:

```
[SETF]   [4: reg][2: size][2: mode]
     mode = 0: [size: imm]
          Flags ← imm

     mode = 1:
          Flags ← reg

     mode = 2:
          Flags ← [address]

     mode = 3: UND
```

SETcc – Conditional Set

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

Description:

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

Specifying a size code has no effect (this is to conform with standard x86_64).

Usage:

```
SETcc r/m
```

Flags Affected:

None.

Format:

```
[SETcc]    [4: dest][3:][1: mem]
     mem = 0:
          dest ← 1 if condition met, otherwise 0
     mem = 1: [address]
          M[address] ← 1 if condition met, otherwise 0
```

MOV – Move

OP Code:

06

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(:size) r, imm/r/m

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

Flags Affected:

None.

Format:

```
[MOV]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← imm
     mode = 1: [address]
          dest ← M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← src
          mode2 = 1: [address]
               M[address] ← src
     mode = 3: [size: imm]   [address]
          M[address] ← imm
```

MOVcc – Conditional Move

Condition	OP Code	Condition	OP Code	Condition	OP Code	Condition	OP Code
Z	07 00	NZ	07 01				
S	07 02	NS	07 03	A	07 0a	G	07 0e
P	07 04	NP	07 05	AE	07 0b	GE	07 0f
O	07 06	NO	07 07	B	07 0c	L	07 10
C	07 08	NC	07 09	BE	07 0d	LE	07 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(:size) r, imm/r/m

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

Flags Affected:

None.

Format:

```
[MOVcc]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← imm
     mode = 1: [address]
          dest ← M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← src
          mode2 = 1: [address]
               M[address] ← src
     mode = 3: [size: imm]   [address]
          M[address] ← imm
```

XCHG – Exchange

OP Code:

08

Description:

Exchanges two values.

Does not support memory-to-memory transfers.

Usage:

```
XCHG(:size) r, r/m

XCHG(:size) m, r
```

Flags Affected:

None.

Format:

```
[XCHG]   [4: r1][2: size][1:][1: mem]
     mem = 0: [4:][4: r2]
          r1 ← r2
          r2 ← r1
     mem = 1: [address]
          r1 ← M[address]
          M[address] ← r1
```

JMP – Jump

OP Code:

09

Description:

Conditionally jumps execution to the provided address. Specifying a size code enabled the use of address widths of different sizes.

These operations form the basis of all conditional branching.

When specifying a size other than 64-bits, the value is zero extended unless using a relative offset (mode = 3 in the binary specification), in which case it is sign extended before being added to the current execution position.

The memory variant is notable for use with function pointers.

Usage:

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

Flags Affected:

None.

Format: (IP is the Instruction Pointer i.e. current execution position)

```
[JMP]   [4: reg][2: size][2: mode]
     mode = 0: [size: val]
          IP <- val

     mode = 1:
          IP <- reg

     mode = 2: [address]
          IP <- M[address]

     mode = 3: [size: val]
          IP <- IP + (sx)val
```

Jcc – Conditional Jump

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

Description:

Conditionally jumps execution to the provided address. Specifying a size code enabled the use of address widths of different sizes. The condition labeled “None” above is an unconditional jump (JMP).

These operations form the basis of all conditional branching.

When specifying a size other than 64-bits, the value is zero extended unless using a relative offset (mode = 3 in the binary specification), in which case it is sign extended before being added to the current execution position.

The memory variant is notable for use with function pointers.

Usage:

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

Flags Affected:

None.

Format: (IP is the Instruction Pointer i.e. current execution position)

```
[Jcc]   [4: reg][2: size][2: mode]
        mode = 0: [size: val]
                if condition met:
                        IP <- val
        mode = 1:
                if condition met:
                        IP <- reg
        mode = 2: [address]
                if condition met:
                        IP <- M[address]
        mode = 3: [size: val]
                if condition met:
                        IP <- IP + (sx)val
```

LOOP – Loop

OP Code:

0b

Description:

Decrements $0 (full 64-bits) and jumps to the provided address if the result is non-zero. Used for simple count-based iteration.

Care should be taken that $0 is not initially zero, as the pre-decrement would cause an underflow and ultimately perform 2^64-1 loop iterations (which is very likely not what you intended).

Usage:

```
LOOP(:size) imm/r/m
```

Flags Affected:

None.

Format:

As JMP.

LOOPcc – Conditional Loop

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

Description:

As LOOP except that the specified condition must be true in addition to $0 being non-zero after decrementation for it to perform the jump.

Usage:

```
LOOPcc(:size) imm/r/m
```

Flags Affected:

As LOOP.

Format:

As LOOP.

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. Specifying a size code enabled the use of address widths of different sizes (only for the address being jumped to; the return address pushed onto the stack is always 64-bit).

Usage:

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

Flags Affected:

None.

Format:

As JMP.

RET – Return

OP Code:

0e

Description:

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

Usage:

```
RET
```

Flags Affected:

None.

Format:

```
[RET]
     pop address
     jmp address
```

PUSH – Push

OP Code:

0f

Description:

Pushes a value onto the stack.

The size parameter determines the size of the value that is pushed.

Usage:

```
PUSH(:size) imm/r/m
```

Flags Affected:

None.

Format:

```
[PUSH]   [4: reg][2: size][2: mode]
     mode = 0: [size: imm]
     mode = 1: use reg
     mode = 2: [address]
     mode = 3: UND
```

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][2:]
     pop value
     dest ← 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.

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 LA a powerful tool for computing integral arithmetic.

Usage:

```
LEA r, m
```

Flags Affected:

None.

Format:

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

ZX – Zero Extend

OP Code:

12

Description:

Zero extends a register value to the specified size.

Converting to the same size or a smaller size is no-op.

Usage:

```
ZX(:to_size) r, instant imm from_size
```

Flags Affected:

None.

Format:

```
[ZX]    [4: reg][2: from_size][2: to_size]
       (to_size) reg ← zero extend (from_size) reg
```

SX – Sign Extend

OP Code:

13

Description:

Sign extends a register value to the specified size.

Converting to the same size or a smaller size is no-op.

Usage:

```
SX(:to_size) r, instant imm from_size
```

Flags Affected:

None.

Format:

```
[SX]   [4: reg][2: from_size][2: to_size]
       (to_size) reg ← sign extend (from_size) reg
```

FX – Floating-Point Extend

OP Code:

14

Description:

Extends (or contracts) a floating-point value.

Specifying sizes other than 64 or 32 bits results in an Undefined Behavior Error.

Usage:

```
FX(:to_size) r, instant imm from_size
```

Flags Affected:

None.

Format:

```
[FX]    [4: reg][2: from_size][2: to_size]
             (to_size) reg ← floating extend (from_size) reg
```

ADD – Add

OP Code:

15

Description:

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

Usage:

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

ADD(:size) 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][2: mode]
     mode = 0: [size: imm]
          dest ← dest + imm
     mode = 1: [address]
          dest ← dest + M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← dest + src
          mode2 = 1: [address]
               M[address] ← M[address] + src
     mode = 3: [size: imm]   [address]
          M[address] ← M[address] + imm
```

SUB – Subtract

OP Code:

16

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(:size) r, imm/r/m

SUB(:size) 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:

```
[SUB]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← dest - imm
     mode = 1: [address]
          dest ← dest - M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← dest - src
          mode2 = 1: [address]
               M[address] ← M[address] - src
     mode = 3: [size: imm]   [address]
          M[address] ← M[address] - imm
```

MUL – Unsigned Multiply

OP Code:

17

Description:

Computes the full product of multiplying two unsigned values, which will have a width of twice the operand size. In the case of a 64-bit product, the high bits are stored in $1, while the low bits are stored in $0.

The first value to multiply must already by loaded into $0. The second value however, may be an immediate, register, or memory value.

Usage:

```
MUL(:size) imm/r/m
```

Operation:

```
byte:  word      $0 ← $0 * val
word:  dword     $0 ← $0 * val
dword: qword     $0 ← $0 * val
qword: dqword $1:$0 ← $0 * val
```

Flags Affected:

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

Format:

```
[MUL]   [4: reg][2: size][2: mode]
     mode = 0: [size: imm]
     mode = 1: use reg
     mode = 2: [address]
     mode = 3: UND
```

IMUL – Signed Multiply

Form	OP Code
Unary	`18 0`
Binary	`18 1`
Ternary	`18 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 and stores the result in a register
Form	**Usage**
Unary	`IMUL(:size) imm/r/m`

Binary	`IMUL(:size) r, imm/r/m`
	`IMUL(:size) m, imm/r`

Ternary	`IMUL(:size) r, r/m, imm`
Form	**Flags Affected**
Unary	OF and CF are both set if there are significant bits in the high portion of the product (i.e. truncation yields a different value) and cleared otherwise. SF is set if the resulting value is negative (i.e. the highest bit of the result is set).
Binary	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.
Ternary	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.
Form	**Format**
Unary	As MUL.
Binary	As ADD.

Ternary	`[IMUL]   [4: dest][2: size][1:][1: mem]`
	`     mem = 0: [4:][4: reg]   [size: imm]`
	`          dest ← reg * imm`
	
	`     mem = 1: [address]   [size: imm]`
	`          dest ← M[address] * imm`

DIV – Unsigned Divide

OP Code:

19

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 $0 (with the high bits in $1 in the case of 64-bit division). The denominator however, may be an immediate, register, or memory value.

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

Usage:

```
DIV(:size) imm/r/m
```

Operation:

```
byte:
     word $0 / byte val
     byte $0 ← quotient
     byte $1 ← remainder
word:
     dword $0 / word val
     word  $0 ← quotient
     word  $1 ← remainder
dword:
     qword $0 / dword val
     dword $0 ← quotient
     dword $1 ← remainder
qword:
     dqword $1:$0 / qword val
     qword  $0 ← quotient
     qword  $1 ← remainder
```

Flags Affected:

CF is set if the resulting remainder is nonzero and cleared otherwise.

Format:

```
[DIV]   [4: reg][2: size][2: mode]
     mode = 0: [size: imm]
     mode = 1: use reg
     mode = 2: [address]
     mode = 3: UND
```

IDIV – Signed Divide

OP Code:

1a

Description:

Computes the full quotient and remainder of the division of two signed values. The numerator has twice the width of the operand size and must already be loaded into $0 (with the high bits in $1 in the case of 64-bit division). The denominator however, may be an immediate, register, or memory value.

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

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

Usage:

```
IDIV(:size) imm/r/m
```

Operation:

```
byte:  word R0 / byte val
       byte R0 ← quotient , byte R1 ← remainder
word:  dword R0 / word val
       word  R0 ← quotient , word  R1 ← remainder
dword: qword R0 / dword val
       dword R0 ← quotient , dword R1 ← remainder
qword: dqword R1:R0 / qword val
       qword  R0 ← quotient , qword  R1 ← remainder
```

Flags Affected:

CF is set if the resulting remainder is nonzero and cleared otherwise.

SF is set if the resulting quotient is negative (i.e. highest bit of quotient is set) and cleared otherwise.

Format:

```
[IDIV]   [4: reg][2: size][2: mode]
     mode = 0: [size: imm]
     mode = 1: use reg
     mode = 2: [address]
     mode = 3: UND
```

SHL – Shift Left

OP Code:

1b

Description:

Shifts the destination (first argument) to the left by source (second argument) bits.

The amount to shift by is given by an 8-bit unsigned integer. The shift count is masked to 6 bits if the source is 64 bits, or to 5 bits otherwise. 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(:size) r, imm/r/m 8

SHL(:size) 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:

```
[SHL]   [4: dest][2: size][2: mode]
     mode = 0: [8: imm]
          dest ← dest << imm
     mode = 1: [address]
          dest ← dest << (8)M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← dest << (8)src
          mode2 = 1: [address]
               M[address] ← M[address] << (8)src
     mode = 3: [8: imm]   [address]
          M[address] ← M[address] << imm
```

SHR – Shift Right

OP Code:

1c

Description:

Shifts the destination (first argument) to the right by source (second argument) bits.

The amount to shift by is given by an 8-bit unsigned integer. The shift count is masked to 6 bits if the source is 64 bits, or to 5 bits otherwise. Shifting by 0 is no-op.

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

Usage:

```
SHR(:size) r, imm/r/m 8

SHR(:size) 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:

```
[SHR]   [4: dest][2: size][2: mode]
     mode = 0: [8: imm]
          dest ← dest >> imm
     mode = 1: [address]
          dest ← dest >> (8)M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← dest >> (8)src
          mode2 = 1: [address]
               M[address] ← M[address] >> (8)src
     mode = 3: [8: imm]   [address]
          M[address] ← M[address] >> imm
```

SAL – Shift Arithmetic Left

OP Code:

1d

Description:

Shifts the destination (first argument) to the left by source (second argument) bits.

The amount to shift by is given by an 8-bit unsigned integer. The shift count is masked to 6 bits if the source is 64 bits, or to 5 bits otherwise. 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(:size) r, imm/r/m 8

SAL(:size) 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:

```
[SAL]   [4: dest][2: size][2: mode]
        mode = 0: [8: imm]
                dest ← dest << imm
        mode = 1: [address]
                dest ← dest << (8)M[address]
        mode = 2: [3:][1: mode2][4: src]
                mode2 = 0:
                        dest ← dest << (8)src
                mode2 = 1: [address]
                        M[address] ← M[address] << (8)src
        mode = 3: [8: imm]    [address]
                M[address] ← M[address] << imm
```

SAR – Shift Arithmetic Right

OP Code:

1e

Description:

Shifts the destination (first argument) to the right by source (second argument) bits.

The amount to shift by is given by an 8-bit unsigned integer. The shift count is masked to 6 bits if the source is 64 bits, or to 5 bits otherwise. Shifting by 0 is no-op.

Vacated bits are filled with the sign bit. For non-negative numbers, this instruction is equivalent to SHR.

Usage:

```
SAR(:size) r, imm/r/m 8

SAR(:size) 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:

```
[SAR]    [4: dest][2: size][2: mode]
     mode = 0: [8: imm]
          dest ← dest >> imm
     mode = 1: [address]
          dest ← dest >> (8)M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← dest >> (8)src
          mode2 = 1: [address]
               M[address] ← M[address] >> (8)src
     mode = 3: [8: imm]   [address]
          M[address] ← M[address] >> imm
```

ROL – Rotate Left

OP Code:

1f

Description:

Rotates the destination (first argument) to the left by source (second argument) bits.

The amount to shift by is given by an 8-bit unsigned integer. Shifting an n-bit value is performed in modulo-n. Shifting by 0 is no-op.

Usage:

```
ROL(:size) r, imm/r/m 8

ROL(:size) 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:

```
[ROL]   [4: dest][2: size][2: mode]
      mode = 0: [8: imm]
            dest ← dest <<< imm
      mode = 1: [address]
            dest ← dest <<< (8)M[address]
      mode = 2: [3:][1: mode2][4: src]
            mode2 = 0:
                  dest ← dest <<< (8)src
            mode2 = 1: [address]
                  M[address] ← M[address] <<< (8)src
      mode = 3: [8: imm]   [address]
            M[address] ← M[address] <<< imm
```

ROR – Rotate Right

OP Code:

20

Description:

Rotates the destination (first argument) to the right by source (second argument) bits.

The amount to shift by is given by an 8-bit unsigned integer. Shifting an n-bit value is performed in modulo-n. Shifting by 0 is no-op.

Usage:

```
ROR(:size) r, imm/r/m 8

ROR(:size) 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:

```
[ROR]    [4: dest][2: size][2: mode]
     mode = 0: [8: imm]
          dest ← dest >>> imm
     mode = 1: [address]
          dest ← dest >>> (8)M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← dest >>> (8)src
          mode2 = 1: [address]
               M[address] ← M[address] >>> (8)src
     mode = 3: [8: imm]   [address]
          M[address] ← M[address] >>> imm
```

AND – Bitwise And

OP Code:

21

Description:

ANDs the destination (first argument) with the source (second argument).

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

Usage:

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

AND(:size) 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:

```
[AND]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← dest & imm
     mode = 1: [address]
          dest ← dest & M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← dest & src
          mode2 = 1: [address]
               M[address] ← M[address] & src
     mode = 3: [size: imm]   [address]
          M[address] ← M[address] & imm
```

OR – Bitwise Or

OP Code:

22

Description:

ORs the destination (first argument) with the source (second argument).

Usage:

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

OR(:size) 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:

```
[OR]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← dest | imm
     mode = 1: [address]
          dest ← dest | M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← dest | src
          mode2 = 1: [address]
               M[address] ← M[address] | src
     mode = 3: [size: imm]   [address]
          M[address] ← M[address] | imm
```

XOR – Bitwise Exclusive Or

OP Code:

23

Description:

XORs the destination (first argument) with the source (second argument).

Usage:

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

XOR(:size) 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:

```
[XOR]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← dest ^ imm
     mode = 1: [address]
          dest ← dest ^ M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← dest ^ src
          mode2 = 1: [address]
               M[address] ← M[address] ^ src
     mode = 3: [size: imm]   [address]
          M[address] ← M[address] ^ imm
```

INC – Increment

OP Code:

24

Description:

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

Usage:

```
INC(:size) 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:][1: mem]
      mem = 0:
            dest ← dest + 1
      mem = 1: [address]
            M[address] ← M[address] + 1
```

DEC – Decrement

OP Code:

25

Description:

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

Usage:

```
DEC(:size) 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:

```
[DEC]    [4: dest][2: size][1:][1: mem]
      mem = 0:
            dest ← dest - 1
      mem = 1: [address]
            M[address] ← M[address] - 1
```

NEG – Negate

OP Code:

26

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/equalities through to the negative range) to compare the (negated) result (LSH) 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(:size) 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:

```
[NEG]    [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← -dest
     mem = 1: [address]
          M[address] ← -M[address]
```

NOT – Bitwise Not

OP Code:

27

Description:

Performs a bitwise NOT on the destination.

Usage:

```
NOT(:size) r/m
```

Flags Affected:

None.

Format:

```
[NOT]   [4: dest][2: size][1:][1: mem]
      mem = 0:
            dest ← ~dest
      mem = 1: [address]
            M[address] ← ~M[address]
```

CMP – Compare

OP Code:

28

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.

Usage:

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

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

Flags Affected:

As SUB.

Format:

As SUB.

FCMP – Floating-Point Compare

OP Code:

29

Description:

As FSUB, but the result is discarded.

The flags are set in such a way that signed arithmetic conditionals correctly compare the result to zero.

Usage:

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

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

Flags Affected:

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

SF is set if the result is negative and cleared otherwise.

OF is cleared.

CF is set if the result is positive or negative infinity and cleared otherwise.

PF is set if the result is NaN and cleared otherwise.

Format:

As FSUB.

TEST – Logical Test

OP Code:

2a

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

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

Flags Affected:

As AND.

Format:

As AND.

CMPZ – Compare Zero

OP Code:

2b

Description:

Equivalent to using CMP with a value of zero.

Note that this instruction is not directly available via the assembler. When the assembler encounters a CMP instruction with a second argument that is an instant imm with a value of zero, it uses this instruction instead.

Usage:

```
CMP(:size) r/m, instant imm 0
```

Flags Affected:

Equivalent to using CMP with a value of zero.

Format:

```
[CMPZ]    [4: dest][2: size][1:][1: mem]
      mem = 0:
            cmp dest, 0
      mem = 1: [address]
            cmp M[address], 0
```

FCMPZ – Floating-Point Compare Zero

OP Code:

2c

Description:

Equivalent to using FCMP with a floating-point value of zero.

The flags are set in such a way that signed arithmetic conditionals correctly compare the result to zero.

Note that this instruction is not directly available via the assembler. When the assembler encounters an FCMP instruction with a second argument that is an instant imm with a value of zero, it uses this instruction instead.

Usage:

```
FCMP(:size) r/m, instant imm 0
```

Flags Affected:

Equivalent to using FCMP with a floating-point value of zero.

Format:

```
[FCMPZ]    [4: dest][2: size][1:][1: mem]
     mem = 0:
          fcmp dest, 0.0
     mem = 1: [address]
          fcmp M[address], 0.0
```

FADD – Floating-Point Add

OP Code:

2d

Description:

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

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

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

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

Flags Affected:

None.

Format:

```
[FADD]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← dest + imm
     mode = 1: [address]
          dest ← dest + M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← dest + src
          mode2 = 1: [address]
               M[address] ← M[address] + src
     mode = 3: [size: imm]   [address]
          M[address] ← M[address] + imm
```

FSUB – Floating-Point Subtract

OP Code:

2e

Description:

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

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

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

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

Flags Affected:

None.

Format:

```
[FSUB]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← dest - imm
     mode = 1: [address]
          dest ← dest - M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← dest - src
          mode2 = 1: [address]
               M[address] ← M[address] - src
     mode = 3: [size: imm]   [address]
          M[address] ← M[address] - imm
```

FSUBR – Floating-Point Reverse Subtract

OP Code:

2f

Description:

As FSUB except that it subtracts the destination from the source and stores the result in the destination.

Usage:

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

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

Flags Affected:

None.

Format:

```
[FSUBR]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← imm - dest
     mode = 1: [address]
          dest ← M[address] - dest
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← src - dest
          mode2 = 1: [address]
               M[address] ← src - M[address]
     mode = 3: [size: imm]   [address]
          M[address] ← imm - M[address]
```

FMUL – Floating-Point Multiply

OP Code:

30

Description:

Multiplies the destination (first argument) by the source (second argument).

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

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

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

Flags Affected:

None.

Format:

```
[FMUL]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← dest * imm
     mode = 1: [address]
          dest ← dest * M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← dest * src
          mode2 = 1: [address]
               M[address] ← M[address] * src
     mode = 3: [size: imm]   [address]
          M[address] ← M[address] * imm
```

FDIV – Floating-Point Divide

OP Code:

31

Description:

Divides the destination (first argument) by the source (second argument).

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

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

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

Flags Affected:

None.

Format:

```
[FDIV]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← quotient dest / imm
     mode = 1: [address]
          dest ← quotient dest / M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← quotient dest / src
          mode2 = 1: [address]
               M[address] ← quotient M[address] / src
     mode = 3: [size: imm]   [address]
          M[address] ← quotient M[address] / imm
```

FDIVR – Floating-Point Reverse Divide

OP Code:

32

Description:

As FDIV except that the source is divided by the destination and the result is stored in the destination.

Usage:

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

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

Flags Affected:

None.

Format:

```
[FDIVR]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← imm / dest
     mode = 1: [address]
          dest ← M[address] / dest
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← src / dest
          mode2 = 1: [address]
               M[address] ← src / M[address]
     mode = 3: [size: imm]   [address]
          M[address] ← imm / M[address]
```

FPOW – Floating-Point Exponentiate

OP Code:

33

Description:

Computes the destination (floating-point) raised to the power of the source (also floating-point).

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

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

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

Flags Affected:

None.

Format:

```
[FPOW]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← pow(dest, imm)
     mode = 1: [address]
          dest ← pow(dest, M[address])
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← pow(dest, src)
          mode2 = 1: [address]
               M[address] ← pow(M[address], src)
     mode = 3: [size: imm]    [address]
          M[address] ← pow(M[address], imm)
```

FPOWR – Floating-Point Reverse Exponentiate

OP Code:

34

Description:

As FPOW except that the source is raised to the power of the destination and the result is stored in the destination.

Usage:

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

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

Flags Affected:

None.

Format:

```
[FPOWR]    [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← pow(imm, dest)
     mode = 1: [address]
          dest ← pow(M[address], dest)
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← pow(src, dest)
          mode2 = 1: [address]
               M[address] ← pow(src, M[address])
     mode = 3: [size: imm]   [address]
          M[address] ← pow(imm, M[address])
```

FLOG – Floating-Point Logarithm

OP Code:

35

Description:

Computes the base-source logarithm of the destination and stores the result in destination.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

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

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

Flags Affected:

None.

Format:

```
[FLOG]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← log(dest, imm)
     mode = 1: [address]
          dest ← log(dest, M[address])
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← log(dest, src)
          mode2 = 1: [address]
               M[address] ← log(M[address], src)
     mode = 3: [size: imm]    [address]
          M[address] ← log(M[address], imm)
```

FLOGR – Floating-Point Reverse Logarithm

OP Code:

36

Description:

As FLOG except that it finds the base-destination logarithm of the source and stores the result in the destination.

Usage:

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

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

Flags Affected:

None.

Format:

```
[FLOGR]    [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← log(imm, dest)
     mode = 1: [address]
          dest ← log(M[address], dest)
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← log(src, dest)
          mode2 = 1: [address]
               M[address] ← log(src, M[address])
     mode = 3: [size: imm]   [address]
          M[address] ← log(imm, M[address])
```

FSQRT – Floating-Point Square Root

OP Code:

37

Description:

Computes the square root of a floating-point value.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FSQRT(:size) r/m
```

Flags Affected:

None.

Format:

```
[FSQRT]   [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← sqrt(dest)
     mem = 1: [address]
          M[address] ← sqrt(M[address])
```

FNEG – Floating-Point Negate

OP Code:

38

Description:

Computes the negative of a floating-point value.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FNEG(:size) r/m
```

Flags Affected:

None.

Format:

```
[FNEG]   [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← -dest
     mem = 1: [address]
          M[address] ← -M[address]
```

FABS – Floating-Point Absolute Value

OP Code:

39

Description:

Computes the absolute value of a floating-point value.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FABS(:size) r/m
```

Flags Affected:

None.

Format:

```
[FABS]   [4: dest][2: size][1:][1: mem]
      mem = 0:
            dest ← abs(dest)
      mem = 1: [address]
            M[address] ← abs(M[address])
```

FFLOOR – Floating-Point Floor

OP Code:

3a

Description:

Computes the floor of a floating-point value. Result is still floating point.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FFLOOR(:size) r/m
```

Flags Affected:

None.

Format:

```
[FFLOOR]   [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← floor(dest)
     mem = 1: [address]
          M[address] ← floor(M[address])
```

FCEIL – Floating-Point Ceiling

OP Code:

3b

Description:

Computes the ceiling of a floating-point value. Result is still floating point.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FCEIL(:size) r/m
```

Flags Affected:

None.

Format:

```
[FCEIL]    [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← ceil(dest)
     mem = 1: [address]
          M[address] ← ceil(M[address])
```

FROUND – Floating-Point Round

OP Code:

3c

Description:

Computes the result of rounding a floating-point value to the nearest integer. Result is still floating point.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FROUND(:size) r/m
```

Flags Affected:

None.

Format:

```
[FROUND]    [4: dest][2: size][1:][1: mem]
      mem = 0:
            dest ← round(dest)
      mem = 1: [address]
            M[address] ← round(M[address])
```

FTRUNC – Floating-Point Truncate

OP Code:

3d

Description:

Rounds a floating-point value towards zero by removing the fractional portion. Result is still floating point.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FTRUNC(:size) r/m
```

Flags Affected:

None.

Format:

```
[FTRUNC]    [4: dest][2: size][1:][1: mem]
      mem = 0:
            dest ← trunc(dest)
      mem = 1: [address]
            M[address] ← trunc(M[address])
```

FSIN – Floating-Point Sine

OP Code:

3e

Description:

Computes the sine of a floating-point value. Angles are in radians.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FSIN(:size) r/m
```

Flags Affected:

None.

Format:

```
[FSIN]   [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← sin(dest)
     mem = 1: [address]
          M[address] ← sin(M[address])
```

FCOS – Floating-Point Cosine

OP Code:

3f

Description:

Computes the cosine of a floating-point value. Angles are in radians.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FCOS(:size) r/m
```

Flags Affected:

None.

Format:

```
[FCOS]    [4: dest][2: size][1:][1: mem]
      mem = 0:
            dest ← cos(dest)
      mem = 1: [address]
            M[address] ← cos(M[address])
```

FTAN – Floating-Point Tangent

OP Code:

40

Description:

Computes the tangent of a floating-point value. Angles are in radians.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FTAN(:size) r/m
```

Flags Affected:

None.

Format:

```
[FTAN]   [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← tan(dest)
     mem = 1: [address]
          M[address] ← tan(M[address])
```

FSINH – Floating-Point Hyperbolic Sine

OP Code:

41

Description:

Computes the hyperbolic sine of a floating-point value. Angles are in radians.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FSINH(:size) r/m
```

Flags Affected:

None.

Format:

```
[FSINH]    [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← sinh(dest)
     mem = 1: [address]
          M[address] ← sinh(M[address])
```

FCOSH – Floating-Point Hyperbolic Cosine

OP Code:

42

Description:

Computes the hyperbolic cosine of a floating-point value. Angles are in radians.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FCOSH(:size) r/m
```

Flags Affected:

None.

Format:

```
[FCOSH]   [4: dest][2: size][1:][1: mem]
      mem = 0:
            dest ← cosh(dest)
      mem = 1: [address]
            M[address] ← cosh(M[address])
```

FTANH – Floating-Point Hyperbolic Tangent

OP Code:

43

Description:

Computes the hyperbolic tangent of a floating-point value. Angles are in radians.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FTANH(:size) r/m
```

Flags Affected:

None.

Format:

```
[FTANH]   [4: dest][2: size][1:][1: mem]
    mem = 0:
        dest ← tanh(dest)
    mem = 1: [address]
        M[address] ← tanh(M[address])
```

FASIN – Floating-Point Arcsine

OP Code:

44

Description:

Computes the inverse sine of a floating-point value. Angles are in radians.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FASIN(:size) r/m
```

Flags Affected:

None.

Format:

```
[FASIN]   [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← asin(dest)
     mem = 1: [address]
          M[address] ← asin(M[address])
```

FACOS – Floating-Point Arccosine

OP Code:

45

Description:

Computes the inverse cosine of a floating-point value. Angles are in radians.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FACOS(:size) r/m
```

Flags Affected:

None.

Format:

```
[FACOS]    [4: dest][2: size][1:][1: mem]
      mem = 0:
            dest ← acos(dest)
      mem = 1: [address]
            M[address] ← acos(M[address])
```

FATAN – Floating-Point Arctangent

OP Code:

46

Description:

Computes the inverse tangent of a floating-point value. Angles are in radians.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FATAN(:size) r/m
```

Flags Affected:

None.

Format:

```
[FATAN]   [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← atan(dest)
     mem = 1: [address]
          M[address] ← atan(M[address])
```

FATAN2 – Floating-Point Binary Arctangent

OP Code:

47

Description:

Computes the arctangent of two floating point values, where y = dest and x = val. Angles are in radians.

May produce special values based on inputs. Refer to 64-bit Floating-Point Numbers for more information.

Usage:

```
FATAN2(:size) r, imm/r/m

FATAN2(:size) m, imm/r
```

Flags Affected:

None.

Format:

```
[FATAN2]    [4: dest][2: size][2: mode]
      mode = 0: [size: imm]
            dest ← atan2 (dest, imm)
      mode = 1: [address]
            dest ← atan2 (dest, M[address])
      mode = 2: [3:][1: mode2][4: src]
            mode2 = 0:
                  dest ← atan2 (dest, src)
            mode2 = 1: [address]
                  M[address] ← atan2 (M[address], src)
      mode = 3: [size: imm]    [address]
            M[address] ← atan2(M[address], imm)
```

FTOI – Floating-Point to Integer

OP Code:

48

Description:

Converts a floating-point value into a signed integer via truncation. Result is integral.

If the floating-point value cannot be represented as a signed integer of the specified width after truncation (which includes large-magnitude numbers, both infinities, and all NaN values), the result is 10…0 in binary (i.e. the minimum value for a signed integer of that width). Thus, the result of FTOI can be compared to this value to test for a conversion failure (except in the single case where the conversion would correctly yield this value, though this chance is insignificant for most applications.

Usage:

```
FTOI(:size) r/m
```

Flags Affected:

None.

Format:

```
[FTOI]   [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← int(dest)
     mem = 1: [address]
          M[address] ← int(M[address])
```

ITOF – Integer to Floating-Point

OP Code:

49

Description:

Converts a signed integer value into a floating-point value. Result is floating-point.

Usage:

```
ITOF(:size) r/m
```

Flags Affected:

None.

Format:

```
[ITOF]   [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← float(dest)
     mem = 1: [address]
          M[address] ← float(M[address])
```

BSWAP – Byte Swap

OP Code:

4a

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(:size) r/m
```

Flags Affected:

None.

Format:

```
[BSWAP]    [4: dest][2: size][1:][1: mem]
        mem = 0:
                dest ← reverse(dest)
        mem = 1: [address]
                M[address] ← reverse(M[address])
```

BEXTR – Bitfield Extract

OP Code:

4b

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

```
BEXTR(:size) 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:

```
[BEXTR]   [4: dest][2: size][2: mode]
      mode = 0: [16: imm]
            dest ← extract(dest, imm)
      mode = 1: [address]
            dest ← extract(dest, (16)M[address])
      mode = 2: [3:][1: mode2][4: src]
            mode2 = 0:
                  dest ← extract(dest, (16)src)
            mode2 = 1: [address]
                  M[address] ← extract(M[address], (16)src)
      mode = 3: [16: imm]   [address]
            M[address] ← extract(M[address], imm)
```

BLSI – Binary Lowest-Set Isolate

OP Code:

4c

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(:size) 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:

```
[BLSI]    [4: dest][2: size][1:][1: mem]
     mem = 0:
          dest ← dest & -dest
     mem = 1: [address]
          M[address] ← M[address] & -M[address]
```

BLSMSK – Binary Lowest-Set Mask

OP Code:

4d

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(:size) 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:

```
[BLSMSK]    [4: dest][2: size][1:][1: mem]
      mem = 0:
            dest ← dest ^ (dest - 1)
      mem = 1: [address]
            M[address] ← M[address] ^ (M[address] - 1)
```

BLSR – Binary Lowest-Set Reset

OP Code:

4e

Description:

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

Usage:

```
BLSR(:size) 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:

```
[BLSR]    [4: dest][2: size][1:][1: mem]
       mem = 0:
              dest ← dest & (dest - 1)
       mem = 1: [address]
              M[address] ← M[address] & (M[address] - 1)
```

ANDN – Bitwise And Not

OP Code:

4f

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:

```
[ANDN]   [4: dest][2: size][2: mode]
     mode = 0: [size: imm]
          dest ← dest & ~imm
     mode = 1: [address]
          dest ← dest & ~M[address]
     mode = 2: [3:][1: mode2][4: src]
          mode2 = 0:
               dest ← dest & ~src
          mode2 = 1: [address]
               M[address] ← M[address] & ~src
     mode = 3: [size: imm]   [address]
          M[address] ← M[address] & ~imm
```

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