CSX64

Processor Specification

Contents

[Purpose 9](#_Toc518332451)

[Processor Details 10](#_Toc518332452)

[General-Purpose Registers (CPU) 10](#_Toc518332453)

[The Stack 11](#_Toc518332454)

[Vector Registers (VPU) 12](#_Toc518332455)

[Floating-Point Registers (FPU) 13](#_Toc518332456)

[Flags Register 14](#_Toc518332457)

[Error Codes 15](#_Toc518332458)

[Main Memory 15](#_Toc518332459)

[Endianness 16](#_Toc518332460)

[Numeric Types 17](#_Toc518332461)

[Integer 17](#_Toc518332462)

[Unsigned Integers 17](#_Toc518332463)

[Signed Integers 17](#_Toc518332464)

[Floating-Point 18](#_Toc518332465)

[64-bit – Double Precision 19](#_Toc518332466)

[32-bit – Single Precision 21](#_Toc518332467)

[Machine Code 22](#_Toc518332468)

[CPU Register Format 22](#_Toc518332469)

[VPU Register Format 22](#_Toc518332470)

[FPU Register Format 22](#_Toc518332471)

[Reading CSX64 Machine Code Specifications 23](#_Toc518332472)

[Memory Address Format 23](#_Toc518332473)

[Conditions 23](#_Toc518332474)

[Translating Machine Code 24](#_Toc518332475)

[Assembly Language 26](#_Toc518332476)

[CSX64 Assembly Language 26](#_Toc518332477)

[Segments / Sections 26](#_Toc518332478)

[Text Segment (.text) 27](#_Toc518332479)

[Data Segment (.data) 27](#_Toc518332480)

[BSS Segment (.bss) 27](#_Toc518332481)

[imm – Immediate 27](#_Toc518332482)

[Character Literals 29](#_Toc518332483)

[Character Escapes 29](#_Toc518332484)

[Instant Imm 30](#_Toc518332485)

[r – Register 30](#_Toc518332486)

[m – Memory 30](#_Toc518332487)

[Operand Size Directive 31](#_Toc518332488)

[Address Size Directive 31](#_Toc518332489)

[Symbols 32](#_Toc518332490)

[Symbols in CSX64 32](#_Toc518332491)

[Predefined Symbols 33](#_Toc518332492)

[Labels 33](#_Toc518332493)

[Local Label 34](#_Toc518332494)

[Address Difference 34](#_Toc518332495)

[Label Macros 34](#_Toc518332496)

[Instruction Syntax 34](#_Toc518332497)

[Directives 35](#_Toc518332498)

[Global 35](#_Toc518332499)

[Extern 35](#_Toc518332500)

[Align (ALIGN, ALIGNB, ALIGNW, ALIGND, ALIGNQ, ALIGNX, ALIGNY, ALIGNZ) 35](#_Toc518332501)

[Declare (DB, DW, DD, DQ, DX, DY, DZ) 36](#_Toc518332502)

[Reserve (RESB, RESW, RESD, RESQ, RESX, RESY, RESZ) 36](#_Toc518332503)

[EQU 37](#_Toc518332504)

[Segment / Section 37](#_Toc518332505)

[Calling Conventions 37](#_Toc518332506)

[Calling a Function 37](#_Toc518332507)

[The Stack Approach 38](#_Toc518332508)

[The Register Approach 39](#_Toc518332509)

[Return Value 39](#_Toc518332510)

[Register Safety 39](#_Toc518332511)

[Standard Calling Conventions 40](#_Toc518332512)

[Command Line Arguments 40](#_Toc518332513)

[Assembly Process 40](#_Toc518332514)

[Development 41](#_Toc518332515)

[Assembling 42](#_Toc518332516)

[Linking 43](#_Toc518332517)

[Executing 44](#_Toc518332518)

[Virtual Operating System 45](#_Toc518332519)

[File Descriptors 45](#_Toc518332520)

[Managed File 45](#_Toc518332521)

[Unmanaged File 45](#_Toc518332522)

[Interactive File 45](#_Toc518332523)

[File Mode 45](#_Toc518332524)

[File Access 46](#_Toc518332525)

[File Seek Origin 46](#_Toc518332526)

[System Calls 46](#_Toc518332527)

[NOP/FNOP – No Operation 50](#_Toc518332528)

[HLT – Halt Execution 51](#_Toc518332529)

[SYSCALL – System Call 52](#_Toc518332530)

[PUSHF/PUSHFD/PUSHFQ – Push Flags 53](#_Toc518332531)

[POPF/POPFD/POPFQ – Pop Flags 54](#_Toc518332532)

[CLx/STx – Clear/Set flag 55](#_Toc518332533)

[SETcc – Conditional Set 56](#_Toc518332534)

[MOV – Move 57](#_Toc518332535)

[MOVcc – Conditional Move 58](#_Toc518332536)

[XCHG – Exchange 59](#_Toc518332537)

[JMP – Jump 60](#_Toc518332538)

[Jcc – Conditional Jump 61](#_Toc518332539)

[LOOP/LOOPcc – Loop 62](#_Toc518332540)

[CALL – Call 63](#_Toc518332541)

[RET – Return 64](#_Toc518332542)

[PUSH – Push 65](#_Toc518332543)

[POP – Pop 66](#_Toc518332544)

[LEA – Load Effective Address 67](#_Toc518332545)

[ADD – Add 68](#_Toc518332546)

[SUB – Subtract 69](#_Toc518332547)

[MUL – Unsigned Multiply 70](#_Toc518332548)

[IMUL – Signed Multiply 71](#_Toc518332549)

[DIV – Unsigned Divide 72](#_Toc518332550)

[IDIV – Signed Divide 73](#_Toc518332551)

[SHL – Shift Left 74](#_Toc518332552)

[SHR – Shift Right 75](#_Toc518332553)

[SAL – Shift Arithmetic Left 76](#_Toc518332554)

[SAR – Shift Arithmetic Right 77](#_Toc518332555)

[ROL – Rotate Left 78](#_Toc518332556)

[ROR – Rotate Right 79](#_Toc518332557)

[RCL – Rotate Carry Left 80](#_Toc518332558)

[RCR – Rotate Carry Right 81](#_Toc518332559)

[AND – Bitwise And 82](#_Toc518332560)

[OR – Bitwise Or 83](#_Toc518332561)

[XOR – Bitwise Exclusive Or 84](#_Toc518332562)

[INC – Increment 85](#_Toc518332563)

[DEC – Decrement 86](#_Toc518332564)

[NEG – Negate 87](#_Toc518332565)

[NOT – Bitwise Not 88](#_Toc518332566)

[CMP – Compare 89](#_Toc518332567)

[TEST – Logical Test 90](#_Toc518332568)

[BSWAP – Byte Swap 91](#_Toc518332569)

[BEXTR – Bitfield Extract 92](#_Toc518332570)

[BLSI – Binary Lowest-Set Isolate 93](#_Toc518332571)

[BLSMSK – Binary Lowest-Set Mask 94](#_Toc518332572)

[BLSR – Binary Lowest-Set Reset 95](#_Toc518332573)

[ANDN – Bitwise And Not 96](#_Toc518332574)

[BT / BTS / BTR / BTC – Bit Test 97](#_Toc518332575)

[CWD/CDQ/CQO/CBW/CWDE/CDQE – Convert Word Size 98](#_Toc518332576)

[MOVZX/MOVSX – Move Zero/Sign Extended 99](#_Toc518332577)

[FLD\_const – Floating-Point Load Constant 100](#_Toc518332578)

[FLD/FILD – Floating-Point Load 101](#_Toc518332579)

[FST/FSTP/FIST/FISTP – Floating-Point Store 102](#_Toc518332580)

[FXCH – Floating-Point Exchange 103](#_Toc518332581)

[FMOVcc – Floating-Point Conditional Move 104](#_Toc518332582)

[FADD/FADDP/FIADD – Floating-Point Add 105](#_Toc518332583)

[FSUB/FSUBP/FISUB – Floating-Point Subtract 106](#_Toc518332584)

[FSUBR/FSUBRP/FISUBR – Floating-Point Subtract 107](#_Toc518332585)

[FMUL/FMULP/FIMUL – Floating-Point Multiply 108](#_Toc518332586)

[FDIV/FDIVP/FIDIV – Floating-Point Divide 109](#_Toc518332587)

[FDIVR/FDIVRP/FIDIVR – Floating-Point Subtract 110](#_Toc518332588)

[F2XM1 – Floating-Point 2x-1 111](#_Toc518332589)

[FABS – Floating-Point Absolute Value 112](#_Toc518332590)

[FCHS – Floating-Point Change Sign 113](#_Toc518332591)

[FPREM – Floating-Point Partial Remainder 114](#_Toc518332592)

[FPREM1 – Floating-Point Partial Remainder 115](#_Toc518332593)

[FRNDINT – Floating-Point Round to Integer 116](#_Toc518332594)

[FSQRT – Floating-Point Square Root 117](#_Toc518332595)

[FYL2X – Floating-Point Y\*Log2(X) 118](#_Toc518332596)

[FYL2XP1 – Floating-Point Y\*Log2(X+1) 119](#_Toc518332597)

[FXTRACT – Floating-Point Extract 120](#_Toc518332598)

[FSCALE – Floating-Point Scale 121](#_Toc518332599)

[FXAM – Floating-Point Examine 122](#_Toc518332600)

[FTST – Floating-Point Test 123](#_Toc518332601)

[FCOM/FCOMP/FCOMPP/FICOM/FICOMP – Floating-Point Compare 124](#_Toc518332602)

[FUCOM/FUCOMP/FUCOMPP – Floating-Point Compare 125](#_Toc518332603)

[FCOMI/FCOMIP/FUCOMI/FUCOMIP – Floating-Point EFLAGS Compare 126](#_Toc518332604)

[FSIN – Floating-Point Sine 127](#_Toc518332605)

[FCOS – Floating-Point Cosine 128](#_Toc518332606)

[FSINCOS – Floating-Point Sine and Cosine 129](#_Toc518332607)

[FPTAN – Floating-Point Partial Tangent 130](#_Toc518332608)

[FPATAN – Floating-Point Partial Arctangent 131](#_Toc518332609)

[FINCSTP/FDECSTP – Floating-Point Increment/Decrement Stack 132](#_Toc518332610)

[FFREE – Floating-Point Free Register 133](#_Toc518332611)

[MOVQ/MOVSD – Move Quadword / Move Scalar Double 134](#_Toc518332612)

[MOVD/MOVSS – Move Doubleword / Move Scalar Single 135](#_Toc518332613)

[MOVDQA/MOVDQU – Move (Aligned/Unaligned) Packed Integers 136](#_Toc518332614)

[MOVDQA64/32/16/8 – Move Aligned Packed Integers 137](#_Toc518332615)

[MOVDQU64/32/16/8 – Move Unaligned Packed Integers 138](#_Toc518332616)

[MOVAPD/MOVUPD – Move (Aligned/Unaligned) Packed Doubles 139](#_Toc518332617)

[MOVAPS/MOVUPS – Move (Aligned/Unaligned) Packed Singles 140](#_Toc518332618)

[ADDSD – Add Scalar Double 141](#_Toc518332619)

[SUBSD – Subtract Scalar Double 142](#_Toc518332620)

[MULSD – Multiply Scalar Double 143](#_Toc518332621)

[DIVSD – Divide Scalar Double 144](#_Toc518332622)

[ADDSS – Add Scalar Single 145](#_Toc518332623)

[SUBSS – Subtract Scalar Single 146](#_Toc518332624)

[MULSS – Multiply Scalar Single 147](#_Toc518332625)

[DIVSS – Add Scalar Single 148](#_Toc518332626)

[ADDPD – Add Packed Doubles 149](#_Toc518332627)

[SUBPD – Subtract Packed Doubles 150](#_Toc518332628)

[MULPD – Multiply Packed Doubles 151](#_Toc518332629)

[DIVPD – Divide Packed Doubles 152](#_Toc518332630)

[ADDPS – Add Packed Singles 153](#_Toc518332631)

[SUBPS – Subtract Packed Singles 154](#_Toc518332632)

[MULPS – Multiply Packed Singles 155](#_Toc518332633)

[DIVPS – Divide Packed Singles 156](#_Toc518332634)

[PAND – Packed Bitwise AND 157](#_Toc518332635)

[POR – Packed Bitwise OR 158](#_Toc518332636)

[PXOR – Packed Bitwise XOR 159](#_Toc518332637)

[PANDN – Packed Bitwise ANDN 160](#_Toc518332638)

[PANDQ/ANDPD – Packed Bitwise AND of Quadwords or Doubles 161](#_Toc518332639)

[PORQ/ORPD – Packed Bitwise OR of Quadwords or Doubles 162](#_Toc518332640)

[PXORQ/XORPD – Packed Bitwise XOR of Quadwords or Doubles 163](#_Toc518332641)

[PANDNQ/ANDNPD – Packed Bitwise ANDN of Quadwords or Doubles 164](#_Toc518332642)

[PANDD/ANDPS – Packed Bitwise AND of Doublewords or Singles 165](#_Toc518332643)

[PORD/ORPS – Packed Bitwise OR of Doublewords or Singles 166](#_Toc518332644)

[PXORD/XORPS – Packed Bitwise XOR of Doublewords or Singles 167](#_Toc518332645)

[PANDND/ANDNPS – Packed Bitwise ANDN of Doublewords or Singles 168](#_Toc518332646)

[PADDQ/PADDD/PADDW/PADDB – Packed Addition of Integers 169](#_Toc518332647)

[PADDSW/PADDSB – Packed Saturated Addition of Integers 170](#_Toc518332648)

[PADDUSW/PADDUSB – Packed Unsigned Saturated Addition of Integers 171](#_Toc518332649)

[PSUBQ/PSUBD/PSUBW/PSUBB – Packed Subtraction of Integers 172](#_Toc518332650)

[PSUBSW/PSUBSB – Packed Saturated Subtraction of Integers 173](#_Toc518332651)

[PSUBUSW/PSUBUSB – Packed Unsigned Saturated Subtraction of Integers 174](#_Toc518332652)

[DEBUG – Debug 175](#_Toc518332653)

[Appendix 176](#_Toc518332654)

[Converting Binary Integers 176](#_Toc518332655)

[Proof of 2’s Complement 177](#_Toc518332656)

[The Heap 178](#_Toc518332657)

[Encoding IEEE 754 64-bit Floating-Point Values 179](#_Toc518332658)

# Purpose

The CSX64 virtual processor was designed primarily to be an educational tool for learning low-level programming. The processor acts as a 64-bit machine code interpreter with its own instruction set that includes hardware support for integral and floating-point computations. The instruction set was designed around the Intel x86\_64 processor architecture to be as realistic and conforming as possible, while cutting out all of the hardware-dependent aspects of real machine code that can be cumbersome when trying to learn how it works.

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

# Processor Details

## General-Purpose Registers (CPU)

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

General-purpose registers are used for processing integral instructions (the most common type) and are under the complete control of the programmer (unlike many other registers in the processor that are at best indirectly modifiable). In x86\_64 architectures (like CSX64), there are 16 64-bit general purpose registers.

Although you can access the full 64-bit contents of any general-purpose register at any time, it is often unnecessary or even undesirable to do so. For this reason, the low 32, 16, and 8 bits (and semi-low 8-bits of RAX, RBX, RCX, and RDX) of each 64-bit register can be used independently of the high bit portions, as listed in the table below. If you plan to write even a simple program in assembly, you’ll need to become very familiar with this table.

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

It should be noted that these smaller subdivisions of the registers are still in fact just subdivisions of the same physical register. If you modify one subdivision of a register, the other subdivisions of that register will see the change. Writing to a partition only changes the bits that make up that partition, except in the case of writing to the 32-bit partitions, which – for compatibility reasons between 32 and 64-bit code – zeroes the upper 32-bits of the register (this even occurs if the operation is a 32-bit *conditional* write that takes the *false* branch e.g. [MOVcc](#_MOVcc_–_Conditional)).

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

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

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

### The Stack

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

Up till now, it’s unlikely you’ve had to deal with the stack directly. In most languages (even low-level ones) the stack is completely hidden from the programmer. In these languages we can only indirectly impact it. We already know that declaring a variable sets aside space on the stack, and that calling a function uses the stack as well. The difference now is that in assembly language, 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 accidentally 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:

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

To add an item to the stack you can use the [PUSH](#_PUSH_–_Push) instruction. To remove an item, you can use the [POP](#_POP_–_Pop) instruction. To call a function you can use the [CALL](#_CALL_–_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](#_RET_–_Return) operation, which pops a 64-bit address off the stack and jumps to it. All of these instructions assume that RSP points to the top of the stack. If this is not the case (e.g. you have destroyed the stack by “clobbering” – and not restoring – RSP), then you should not use these instructions at all.

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

## Vector Registers (VPU)

It’s fairly common to have situations where you have an array of data that you’re performing the same instruction on iteratively. For example, you might have a pair of coordinates (say, X and Y) that you use to define a position. If you wanted to add 2 of such coordinates structures, you’d have to compute Ax+Bx and Ay+By. This doesn’t seem so bad until you find out you need to do this 10 million times.

Modern computers have many specialized tricks up their sleeves, and one of these lies in what’s known as the Vector Processing Unit (VPU). The VPU contains vector registers, which are very large registers that you can put several data elements in and then process them all in parallel. The standard VPU registers and their partitions are listed in the table below. Note that just like with general-purpose registers, the partitions are still just partitions of the same physical register.

|  |  |  |
| --- | --- | --- |
| **VPU Registers** | | |
| **Full 512-bit** | **(Low) 256-bit** | **(Low) 128-bit** |
| ZMM0 | YMM0 | XMM0 |
| ⋮ | ⋮ | ⋮ |
| ZMM15 | YMM15 | XMM15 |
| ⋮ |  |  |
| ZMM31 |

Additionally, most VPU instructions accept a white mask that can be used to conditionally perform their operation on specific elements (e.g. you could have it perform addition on only elements 0, 2, 4, etc.).

The benefits are quite obvious: the VPU is blazingly fast. And, if you properly multithread your program in addition to using VPU instructions to perform super-parallel processing, you can get even higher speeds (though this is very sensitive to architecture). In fact, if you take said program and properly optimize it, it can actually produce so much heat that the CPU will lower the clock speed to keep it from overheating from sheer mathematical might of your unholy creation.

You’ve probably heard of processors being rated in FLOPS. A FLOP is one (32-bit) Floating-Point Operation per second, and the numbers they put in the hardware specs are taken from the optimal case of properly-optimized use of the VPU on 32-bit floating-point data using a single core (i.e. not multithreaded).

Here’s the catch: the VPU is very picky about the alignment of your data. While you can *load/store* an unaligned vector data block (at a steep performance penalty), most operations on that data (e.g. addition) don’t have unaligned equivalents (though you could do an unaligned move into a VPU register and then use that to perform the operation). Because of this, if you want to really get the most out of the VPU you’ll have to make sure that your data is properly aligned (e.g. 128-bit vectors must be aligned on 16-byte boundaries, 256-bit on 32-byte boundaries, and 512-bit on 64-byte boundaries).

As a consequence of being so large and powerful, the VPU uses a ridiculous amount of power to run (comparatively speaking). Because of this, the CPU will typically only keep the XMM pathways open, and only power up the YMM and ZMM pathways when requested to (which is relatively slow – several hundred clocks) and turns it off when it hasn’t been used in a while (usually after about a millisecond). Because of this, unless you’re processing massive amounts of information with no large pauses between data fetches, it’s likely that using XMM registers will outperform YMM and ZMM registers.

Although the most powerful use of vector registers is in performing vector operations on entire swaths of data at a time, the most common use is to process only a single element. When this is done in the VPU it is referred to as a scalar operation. These differ from simply using a white mask in that the data doesn’t need to be aligned to a vector register width and is potentially more efficient (depending on hardware). The reason for this is that the VPU has (limited) floating-point support and is much easier to use than the FPU due to having more registers and having a more orthodox structure and mnemonic syntax.

## Floating-Point Registers (FPU)

The Floating-Point Unit (FPU) is a coprocessor that is built into the motherboard to facilitate floating-point arithmetic. Being a coprocessor, the FPU is a bit tricky to deal with at first, as it has its own set of flags, registers, and an entirely different structure from the usual CPU/VPU instructions.

The FPU has 8 80-bit registers named either ST0-ST7 or ST(0)-ST(7) depending on the assembler (CSX64 allows both). The name ST is for “stack top”, and thus – in most assemblers – ST with no index will reference ST0. The FPU registers are 80-bit to reduce rounding errors from floating-point computations involving the usual 32-bit (“single precision”) and 64-bit (“double precision”) floating-point formats but can also be used to enable 80-bit (“extended precision”) computations (though CSX64 does not support this). Although the VPU does have (limited) floating-point support, it doesn’t use extended precision, and thus will have a larger mean error than its FPU counterpart.

Conveniently, extended precision has 64 significand bits, meaning an FPU register can losslessly store any 64-bit integer (so you could technically use the FPU for integral operations as well if you really wanted to).

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

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

Although the FPU registers are accessed in this weird stack system, you may still refer to any FPU register at any time. However, the FPU also keeps track of the state of its registers. For instance, it knows which registers have meaningful data in them.

Values are loaded from and stored to memory via pushing and popping this “register stack”. When you push a value onto the FPU stack the FPU first decrements the top of stack index by 1 (wrapping from 0 to 7), puts the value there, and marks that register as non-empty. When you pop a value off the FPU stack the FPU first gets the value at the top of the stack, marks that register as empty, and finally increments the top of stack index by 1 (wrapping from 7 to 0). Because of this, it is also possible to overflow or underflow the FPU stack, which are given their own error codes in CSX64.

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

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

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

## Flags Register

The processor itself needs to keep track of key pieces of status information. Some of this information is held in a special “flags” register, which holds many bitfields representing different processor states. The CSX64 flags register conforms to the Intel x86\_64 standard but uses several reserved flags in the high 32 bits for its own special purposes. The Flags are enumerated in order in the table below:

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

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

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

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

## Error Codes

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

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

| **Error Code** | **Value** | **Description** |
| --- | --- | --- |
| None | 0 | No error has occurred. |
| OutOfBounds | 1 | A memory access operation has exceeded the amount of memory allocated to the process. |
| UnhandledSyscall | 2 | A [system call](#_Default_Virtual_Operating) 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](#_Conditions) 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](#_File_Descriptors). |
| FDNotInUse | 10 | Attempted to perform an IO operation on a file descriptor that was not in use. |
| NotImplemented | 11 | An instruction was used that has not been implemented. |
| StackOverflow | 12 | A stack operation went outside the stack segment. |
| FPUStackOverflow | 13 | Attempt to push a value onto a full FPU register stack. |
| FPUStackUnderflow | 14 | Attempt to pop a value from an empty FPU register stack. |
| FPUError | 15 | An error occurred during and FPU computation. |
| FPUAccessViolation | 16 | Attempt to use an FPU register that doesn’t have a value. |
| AlignmentViolation | 17 | Attempt to read/write memory that is not properly aligned. |

## Main Memory

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

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

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

### Endianness

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

Let’s conceptualize this with base 10 numbers:

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

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

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

# Numeric Types

## Integer

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

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

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

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

For information on how to convert integers to and from binary, see the [examples page on binary integers](#_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](#_Proof_of_2’s)), 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](#_Heap_1)), and thus would likely not fit in a register for processing. Because of this, operations involving it would be highly entangled with [main memory](#_Main_Memory), which would slow things down tremendously. And even then, what about numbers that repeat infinitely? Where should we cut them off?

Conveniently, scientists have given us the answer with their inspirationally-titled *scientific notation*. If this entry in the CSX64 manual hasn’t phased you yet, you’re well accustomed to scientific notation already. The idea is to use a fixed-width binary number with negative powers of 2 (as discussed above), but also multiply it by a power of 2, which solves the issue of it being limited in range compared to an integer of equal width. Another key aspect of scientific notation is that it is of the form a.etc\*10b, 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 2n-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](#_Encoding_IEEE_754) in the Appendix.

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

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

### 32-bit – Single Precision

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

Here’s a graphical representation of the data structure:

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

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

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

# Machine Code

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

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

Machine code is ***very*** technical (in fact, ***maximum*** technical), and nearly all 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.

## CPU Register Format

As there are 16 general-purpose registers, they are designated by a 4-bit register id, where the registers are enumerated 0-15 in the order they’re listed in the [General-Purpose Registers](#_General-Purpose_Registers) section.

The partition of the register to use is denoted by a 2-bit size code.

|  |  |  |
| --- | --- | --- |
| **Size Code** | **Size** | **Size Name** |
| 0 | 8 bits | BYTE |
| 1 | 16 bits | WORD |
| 2 | 32 bits | DWORD |
| 3 | 64 bits | QWORD |

In the case of AH, BH, CH, and DH, an 8-bit size is specified and an additional flag elsewhere in the binary code is set to 1 (typically named dh or sh). This flag will be well-documented. It is possible an instruction does not allow the use of these registers, which would also be well-documented.

## VPU Register Format

As there are 32 VPU registers (ZMM0-ZMM31), vector registers are specified by a 5-bit code 0-31.

The width of the vector register to use (XMM, YMM, ZMM) is determined by a 2-bit vector size code. In the case of XMM and YMM registers (which are only 0-15), specifying a value outside of this range results in an [UndefinedBehavior](#_Flags) error.

|  |  |  |
| --- | --- | --- |
| **VPU Size Code** | **Size** | **Size Name** |
| 0 | 128 bits | XMMWORD |
| 1 | 256 bits | YMMWORD |
| 2 | 512 bits | ZMMWORD |

## FPU Register Format

As there are 8 FPU registers (ST0-ST7), they are specified by a 3-bit code 0-7.

## Reading CSX64 Machine Code Specifications

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

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

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

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

## Memory Address Format

In CSX64, [memory](#_Main_Memory) addresses as operands to an instruction are standardized and take the following form:

[1: use\_imm][1:][2: mult\_1][2: size][1: use\_r1][1: use\_r2]

([4: r1][4: r2]) ([size: imm])

If the “use\_imm” flag is 1, the “imm” field is assumed to be provided, otherwise it defaults to zero.

The “size” field holds the size code of the address components (i.e. r1, r2, and imm). One caveat is that 8-bit mode is not allowed (but a 256-byte address space is so small you wouldn’t want to do that anyway).

If the “use\_r1” or “use\_r2” flags are set, the “r1:r2” is assumed to be provided. For each of these flags, if it is set, that register’s contents are added to the resulting address. In the case of r1, the value is first pre-multiplied according to the “mult\_1” size code.

## Conditions

At the hardware level, conditionals are processed based on the value of the [flags register](#_Flags_Register). These flags are modified in different ways by many instructions and are used to perform conditional instructions (e.g. conditional jumps – aka if statements). These conditions are listed below:

| **Label** | **Alias** | **Name** | **Signage** | **Condition** |
| --- | --- | --- | --- | --- |
| Z | E | Zero / Equal | N/A | ZF = 1 |
| NZ | NE | Not Zero / Not Equal | ZF = 0 |
| P | PE | Parity / Parity Even | PF = 1 |
| NP | PO | Not Parity / Parity Odd | PF = 0 |
| O |  | Overflow | OF = 1 |
| NO |  | Not Overflow | OF = 0 |
| C |  | Carry | CF = 1 |
| NC |  | Not Carry | CF = 0 |
| S |  | Sign | SF = 1 |
| NS |  | Not Sign | SF = 0 |
| B | NAE | Below | Unsigned | CF = 1 |
| BE | NA | Below or Equal | CF = 1 or ZF = 1 |
| A | NBE | Above | CF = 0 and ZF = 0 |
| AE | NB | Above or Equal | CF = 0 |
| L | NGE | Less | Signed | SF != OF |
| LE | NG | Less or Equal | ZF = 1 or SF != OF |
| G | NLE | Greater | ZF = 0 and SF = OF |
| GE | NL | Greater or Equal | SF = OF |

The “Condition” column for the arithmetic comparisons is only there for academic purposes to show how, under the hood, they’re evaluated based on the other flags. Don’t bother memorizing those. In all likelihood, the only time you’ll ever need to know their expressions is when you’re putting them in a table for academic purposes to show how, under the hood, they’re evaluated based on the other flags.

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

## Translating Machine Code

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

The first thing we need to think about is how a computer knows what it’s supposed to do and when. This is accomplished by OP codes, which are numbers that a computer is hardwired (not really, nowadays, but you get the idea) to see and perform a specific action. In a standard Intel x86\_64 processor there is a register named RIP (64-bit instruction pointer) that holds a pointer to the current instruction being executed. When the processor goes to execute an instruction, it looks at the byte(s) at that location and acts accordingly.

However, this isn’t enough information. For all but the simplest instructions the processor will need to know more information to perform the action, such as locations in memory to load or store values. An example of this was shown in the section on CSX64 [memory addresses](#_Memory_Address_Format_1). In these cases, we need to follow the specific operation’s binary format to the letter.

For example, to add 17 to the 32-bit register partition ECX, we would first look up the [ADD](#_ADD_–_Add_1) 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](#_Jcc_–_Conditional_1)). In these cases, the order of the individual bytes of the OP code are how they should be arranged (i.e. do not try to account for endianness). For our example, this gives us the following binary form (in hexadecimal):

12 24 10 11 00 00 00

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

Now, say we wanted to compare the result to an element of an array (of 32-bit integers) in memory that starts at address 200. Let’s also say that we want the index of the item in the array to be the current value of RDI. We thus need to use [CMP](#_CMP_–_Compare_1)’s binary specification, which in turn uses the [address specification](#_Memory_Address_Format):

Which gives us the binary form:

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

Now let’s say we want to jump to address 1000 if the values were equal (i.e. result was zero – equal condition is an [alias](#_Conditions) for zero condition). We thus use [Jz](#_Jcc_–_Conditional_1)’s binary specification, which gives us the binary form:

0b 00 0a e8 03 00 00

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

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

12 24 10 11 00 00 00

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

0b 00 0a e8 03 00 00

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

R2 += 17;

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

# Assembly Language

As you can imagine, [writing programs in machine code](#_Translating_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](#_Main_Memory)). For all of those reasons and many, many more, programs are never written in machine code except for academic purposes. Assembly language serves as the bridge between the programmer and pure machine code and is the closest programming “language” (i.e. human-readable instructions) to the underlying hardware.

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

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

add ecx, 17

cmp ecx, [200 + 4\*rdi]

jz 1000

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

# CSX64 Assembly Language

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

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

## Segments / Sections

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

### Text Segment (.text)

The text segment holds all of your executable code and will typically dwarf the other segments in terms of size. The text segment is read-only and is the only segment that is executable (attempting to write to the text segment at runtime or execute anything outside it results in an [AccessViolation](#_Heap) 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](#_Declare_(db,_dw,) directive.

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

#### Read-Only Data Segment (.rodata)

The rodata segment is like the data segment except that it is read-only (attempting to write to the rodata segment at runtime results in an [AccessViolation](#_Flags) error).

### BSS Segment (.bss)

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

To add to the BSS segment, you must use the [Reserve](#_Reserve_(resb,_resw,) 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](#_Translating_Machine_Code).

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

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

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

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

Furthermore, there are several ways to denote numeric literals, which are described below. For all of these forms except character literals, underscores can be used (after the prefix) as a no-op separator character. This is occasionally useful in breaking up large binary literals.

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Example** |
| Hexadecimal Integer | Prefaced by “0x” | 0xdeadbeef |
| Binary Integer | Prefaced by “0b” | 0b1001\_1011 |
| Octal Integer | Prefaced by “0” | 0712 |
| Decimal Integer | Otherwise | 167 |
| Character (Integer) | Up to 8 characters enclosed in single, double, or back quotes.  Back quotes enable C-style character escapes. | ‘A’  “abcdefgh”  `hello\n` |
| Decimal Floating-Point | Begins with a digit and contains a decimal point and/or e or E with an exponent (which may optionally have a sign) for exponential notation.  “∞” for [infinity](#_64-bit).  “NaN” for [NaN](#_64-bit_–_Double) (case sensitive). | 0.1  3.14  2.5e4  1.67e-11  -∞  NaN |

### Character Literals

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

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

### Character Escapes

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

| **Escape Sequence** | **Resulting Value** |
| --- | --- |
| \’ | ‘ |
| \” | “ |
| \` | ` |
| \\ | \ |
| \? | ? |
| \a | Bell (alert) (ASCII 7) |
| \b | Backspace (ASCII 8) |
| \t | Tab (ASCII 9) |
| \n | New Line (ASCII 10) |
| \v | Vertical Tab (ASCII 11) |
| \f | Form Feed (ASCII 12) |
| \r | Carriage Return (ASCII 13) |
| \e | Escape (ASCII 27) |
| \377 | Up to 3 octal digits – literal byte |
| \xFF | Up to 2 hex digits – literal byte |
| \u1234 | Unicode character (not yet supported) |
| \U12345678 |

### Instant Imm

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

## r – Register

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

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

## m – Memory

A memory value is an expression that evaluates to the address of some value in memory. In CSX64 assembly language, addresses are enclosed in brackets “[…]” with the address expression inside. An address expression follows the same rules as any imm, with the added ability to use up to two 64-, 32-, or 16-bit registers – though they must both be of the same size – in computing the address, as discussed in the section on [memory addresses](#_Memory_Address_Format).

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

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

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

Finally, the assembler will combine multipliers of the same register (e.g. [6\*rdi + 8\*rbx - 5\*rdi] is converted to [rdi + 8\*rbx] and is thus legal). In fact, more than two registers may be specified, so long as only two of them have nonzero multipliers (e.g. [4\*rax + 8\*rbx - 4\*rax + rcx] is converted to [8\*rbx + rcx] and is thus legal).

### Operand Size Directive

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

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

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

|  |  |
| --- | --- |
| **Operand Size Directive** | **Operand Size** |
| BYTE PTR […] | 8 bits |
| WORD PTR […] | 16 bits |
| DWORD PTR […] | 32 bits |
| QWORD PTR […] | 64 bits |

### Address Size Directive

You should already be quite aware that addresses can have different sizes. However, in every language except assembly – though the address width may vary – they are always the same size in any one compilation. For instance, compiling a C++ program in 32-bit mode makes all pointers 32-bit, and compiling in 64-bit mode makes all pointers 64-bit. This is because these languages are not concerned with the specific machine-code-level realization of how something works, only that it does work. Because of this, you specify “pointer” or “double”, whereas in assembly you have to be more aware about how the hardware works.

One of the hallmarks of modern computing (and at the same time an unfortunate crutch) is its remarkable level of backwards-compatibility. Although we live in a mostly-64-bit processor era, most programs are 32-bit, and there are even some 16-bit programs still lurking about. Because we need these old programs to still work, our hardware has to be able to accommodate this, and because our hardware accommodates it, assembly has a way to toy with it explicitly. With this you could merge 64-, 32-, and 16-bit assembly files together if you really wanted to.

The only difference between the various address widths is their address range. Thus, if you know your program will definitely not need more than 4GB, you can use 32-bit addresses and save some space.

In CSX64 this is accomplished by specifying a size inside the brackets of a memory operand. **This should not be confused with an operand size directive**, which is on the outside. Also note that these directives don’t use PTR, which further differentiates an address size from an operand size and may help you to remember the difference (i.e. WORD PTR [DWORD …] is a pointer to a word and is accomplished via a 32-bit address. These directives are likewise not case sensitive.

|  |  |
| --- | --- |
| **Address Size Directive** | **Address Size** |
| … [BYTE …] | 8 bits |
| … [WORD …] | 16 bits |
| … [DWORD …] | 32 bits |
| … [QWORD …] | 64 bits |

## Symbols

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

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

### Symbols in CSX64

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

Symbols can be used in any imm expression, even before they’ve been defined (though that would make them not [instant imms](#_Instant_Imm)). 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](#_Local_Symbol).

Symbols that are not labels are created by the [EQU](#_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](#_Default_Virtual_Operating_1).

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](#_Instant_Imm) is required.

|  |  |
| --- | --- |
| **Symbol Name** | **Value** |
| \_\_heap\_\_ | The sum of the lengths of all the user-defined [segments](#_Segments). This is thus the low side of the heap/stack segment that is allocated by the virtual operating system upon program initialization. |

## Labels

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

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

### Local Label

A local label is a label that is associated with a non-local label. Because assembly language is prolific with labels, you can easily run into the problem of having clashing label names. Rather than naming your labels “loop\_top\_1”, “loop\_top\_2”, etc., you can give them local names. A local label is considered “tied” to the most-recent non-local label.

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

### Address Difference

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

string\_start: db “hello world!”

string\_end:

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

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

### Label Macros

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

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

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

## Instruction Syntax

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

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

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

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

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

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

## Directives

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

### Global

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

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

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

### Extern

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

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

### Align (ALIGN, ALIGNB, ALIGNW, ALIGND, ALIGNQ, ALIGNX, ALIGNY, ALIGNZ)

The Align directive takes a single operand that is an instant integral imm. When used inside a given segment, the next value written to that segment will be aligned to the specified number of bytes. This is commonly done for performance (as accessing aligned memory is usually around twice as fast), but is especially important for using VPU instructions, which require their memory blocks to be aligned on non-standard boundaries. The operand to Align must be a power of 2.

The suffixed flavors of Align take no operands and align specifically to their representative size. They are not standard directives in assembly languages but are added in CSX64 to parallel the Declare and Reserve directive flavors.

### Declare (DB, DW, DD, DQ, DX, DY, DZ)

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

DX, DY, and DZ are for XMM, YMM, and ZMM registers respectively. These don’t allow numeric operands (because it doesn’t know what size you want for each element in the vector), but they do allow string operands (because strings are always composed of 8-bit words).

An important thing to know about Declare is that it doesn’t align its data. You would have to do that explicitly with an Align directive immediately before the declare.

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

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

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

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

### Reserve (RESB, RESW, RESD, RESQ, RESX, RESY, RESZ)

Reserve is like Declare except that it can only be used in the [BSS segment](#_BSS_Segment_(.bss)) and cannot be used to initialize the data to a specific value (by virtue of being in the BSS segment). Additionally, Reserve takes only a single argument, which is the number of words of the given size to allocate (and zero) upon program initialization (e.g. resw 32 reserves 32 contiguous 16-bit locations). All of the normal suffixes function the same as their Declare counterparts.

Like the Declare directive, Reserve does not align its data. If you require this, you should use an Align directive immediately before the reserve.

### EQU

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

### Segment / Section

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

## Calling Conventions

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

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

### Calling a Function

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

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

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

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

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

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

### The Stack Approach

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

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

#### Stack Framing

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

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

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

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

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

### The Register Approach

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

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

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

### Return Value

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

### Register Safety

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

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

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

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

### Standard Calling Conventions

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

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

1. Use CALL/RET (do not perform bastardized jumps).
2. Put the first 6 integral arguments left-to-right into RDI, RSI, RDX, RCX, R8, R9 (or partitions thereof).
   1. Put the first 8 floating-point arguments right-to-left in XMM0-XMM7 (scalar moves).
   2. Put any extra arguments on the stack from right-to-left.
   3. If the data type will not fit in a register, consider taking a pointer, otherwise put it on the stack.
3. Place the return value in RAX if integral (or a partition thereof) or XMM0 if floating-point.
4. Registers RBP, RSP, and R13-R15 are call-safe.

### Command Line Arguments

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

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

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

## Assembly Process

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

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

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

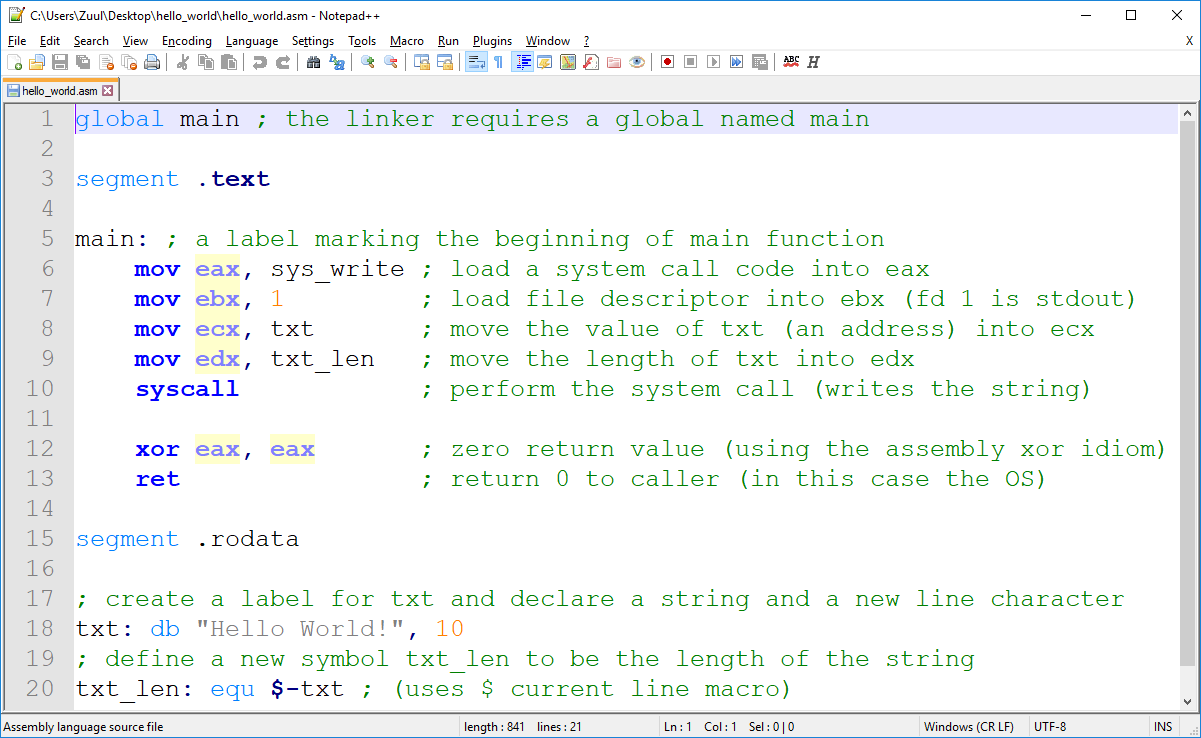
Well, you’ve made it this far, so how about we finally talk about how to do all this?

### Development

The development process is the only part that should concern you (as assembly is a cruel and vindictive mistress). Fortunately, you should also already be very familiar with it. This step just involves opening your editor of choice and coding away to your heart’s content.

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. Here’s our assembly source file:



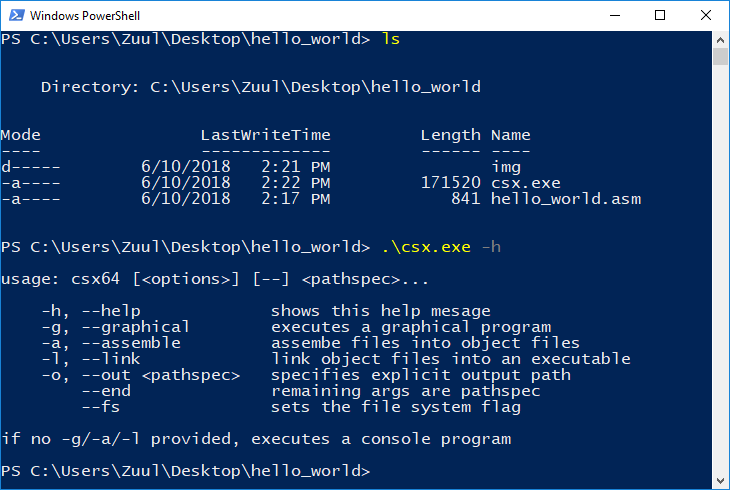
Here you see the basic structure of an assembly program and just how tedious even printing a single line of text can be (but rest assured there are ways around many of these problems).

A few things to note:

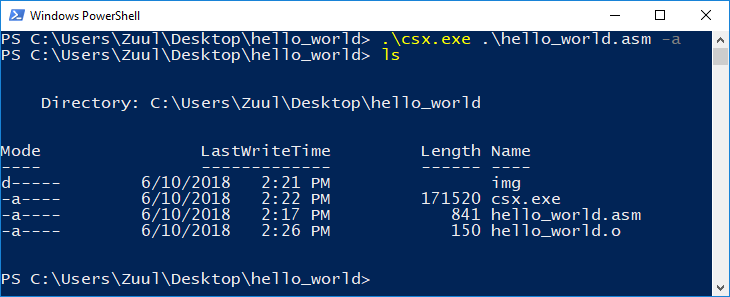
1. We defined a “function” called main and marked it as global for the linker.
2. We used several statements to get the registers into a state we wanted before actually performing the system call (this also applies to any function call).
3. We set eax (return value) to zero using the xor idiom. It is slightly more efficient in terms of executable size to xor a register with itself than to set it to zero, as setting it to zero would need to store the 4-byte value (of zero), that will be loaded into the register. In other languages if you set something to zero the compiler will often 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 in the data or read-only data segments. We create a label for the variable (in this case a string) and write out its value using the Declare directive DB.
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.

### 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. This will be the first occurrence of actually using the CSX64 application (csx.exe). Before we begin, I’ll show the working directory and provide the -h option to csx.exe to list all of the available options:



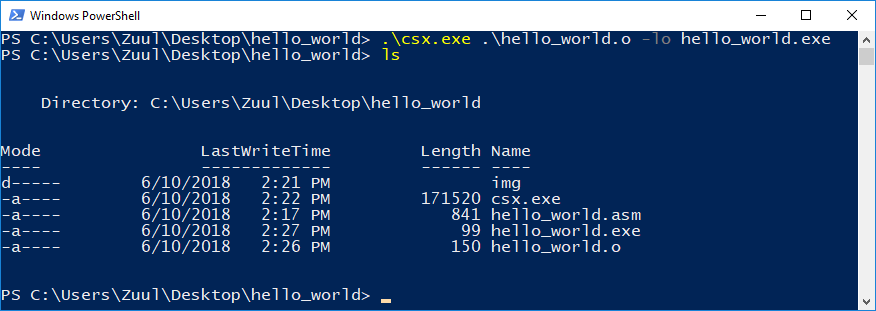
We want to assemble hello\_world.asm into an object file, so we need the -a option:



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

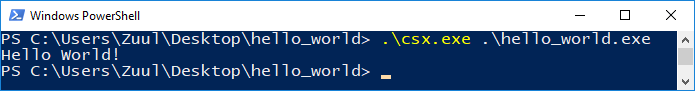
We now need to link our object files (in this case just the one) together to create an executable. To do this we use the -l (lowercase L) option:



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 non-descriptive.

### 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 -g/-a/-l options) defaults it to execution mode:



As a side note, all the options after csx.exe will be passed to the program, with the number of arguments in RDI and a pointer to an array of pointers to the arguments in RSI, as discussed in [Command Line Arguments](#_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](#_SYSCALL_–_System) 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 integers that can be used in some system calls to refer to an OS-specific structures used for managing a file.

There is usually a (relatively small) maximum 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](#_Heap) 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

In CSX64, 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](#_Default_Virtual_Operating_1), an unmanaged file will not be closed, but its connection to the file descriptor will be severed (thus freeing it for reuse). Note that this means some system-specific utilities (e.g. [DEBUG](#_DEBUG_–_Debug)) still have access to the file, but you don’t anymore.

Some examples of unmanaged files are the standard input, output, and error files.

### Interactive File

If a file is interactive, reading past the end will put the processor in the Suspended Read 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 (the vanilla console input is not interactive because the virtual operating system didn’t create it – the real operating system did, and it performs this feature by default).

### 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](#_Default_Virtual_Operating_1). You can also set the position via [sys\_seek](#_Default_Virtual_Operating_1). 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](#_SYSCALL_–_System) 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, modify, or even remove features.

To perform a system call in CSX64, a special “system call code” is loaded into RAX prior to executing a SYSCALL instruction. Based on the system call code, 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.

Arguments to system calls in CSX64 are loaded left-to-right into RAX, RBX, RCX, and RDX, where RAX is always the system call code. The system call calling conventions vary based on whatever operating system you’re on, which is why their behavior is almost always wrapped into a function that’s used instead. Fortunately, CSX64 is an emulated experience, which means you’re free to be prolific with raw system calls without the risk of running into this problem. The important thing to remember is that the system call calling conventions are set in stone for whatever system you’re on, as opposed to function calling conventions, which you could change if you wanted to by rewriting the function.

The default system call table is listed below:

| **Name** | **Code** | **Description** |
| --- | --- | --- |
| sys\_read | 0 | Reads binary data from a file.  RBX should contain the file descriptor to access.  RCX should contain the address of the location to store the data.  RDX should contain the maximum number of bytes to read.  The number of bytes read will be placed in RAX.  Attempting to read at EOF is not an error. In this case interactive files will trigger the Suspended Read state pending more data. Otherwise, zero is returned.  Fails with [OutOfBounds](#_Heap) if the fd is invalid or a memory access fails.  Fails with [FDNotInUse](#_Heap) if the file descriptor specified is not in use.  Fails with [AccessViolation](#_Flags) if attempting to write to read-only memory.  Fails with [IOFailure](#_Heap) if an IO error occurs. |
| sys\_write | 1 | Writes binary data to a file.  RBX should contain the file descriptor to access.  RCX should contain the address of the data array to write.  RDX should contain the number of bytes to write.  Fails with [OutOfBounds](#_Heap) if the fd is invalid or a memory access fails.  Fails with [FDNotInUse](#_Heap) if the file descriptor specified is not in use.  Fails with [IOFailure](#_Heap) if an IO error occurs. |
| sys\_open | 2 | Opens or creates a file and ties it to a file descriptor. Files opened with this system call are [managed](#_Managed_File).  RBX should contain the address of the C string to use as the path.  RCX should contain the file mode to open with.  RDX should contain the file access to open with.  The file descriptor opened will be placed in RAX.  Fails with [FSDisabled](#_Heap) if [FSF](#_Conditions) is not set.  Fails with [InsufficientFDs](#_Heap) if there are no unused file descriptors.  Fails with [OutOfBounds](#_Heap) if the string went out of memory bounds.  Fails with [IOFailure](#_Heap) if the file could not be opened. |
| sys\_close | 3 | Flushes and closes a file. Closing an unused file is no-op.  RBX should contain the file descriptor to close.  Fails with [OutOfBounds](#_Heap) if file descriptor is invalid.  Fails with [IOFailure](#_Heap) if the file failed to close properly. |
| sys\_flush | 4 | Flushes a file’s data buffer.  RBX should contain the file descriptor to flush.  Fails with [OutOfBounds](#_Heap) if file descriptor is invalid.  Fails with [FDNotInUse](#_Heap) if the file descriptor specified is not in use.  Fails with [IOFailure](#_Heap) if an IO error occurs. |
| sys\_seek | 5 | Moves the current read/write position in the file.  RBX should contain the file descriptor to seek in.  RCX should contain the address in the file to seek to.  RDX should contain the seek mode to use.  Fails with [OutOfBounds](#_Heap) if file descriptor is out of range.  Fails with [FDNotInUse](#_Heap) if the file descriptor specified is not in use.  Fails with [IOFailure](#_Heap) if an IO error occurs. |
| sys\_tell | 6 | Gets the current read/write position in the file.  RBX should contain the file descriptor to get the position of.  The current position will be placed in RAX.  Fails with [OutOfBounds](#_Heap) if file descriptor is out of range.  Fails with [FDNotInUse](#_Heap) if the file descriptor specified is not in use.  Fails with [IOFailure](#_Heap) if an IO error occurs. |
| sys\_move | 7 | Moves a file in the file system.  RBX should contain the address of the C string source path.  RCX should contain the address of the C string destination path.  Fails with [FSDisabled](#_Heap) if [FSF](#_Conditions) is not set.  Fails with [OutOfBounds](#_Heap) if either string went out of memory bounds.  Fails with [IOFailure](#_Heap) if the file could not be moved. |
| sys\_remove | 8 | Removes a file in the file system.  RBX should contain the address of the C string path to remove.  Fails with [FSDisabled](#_Heap) if [FSF](#_Conditions) is not set.  Fails with [OutOfBounds](#_Heap) if the string went out of memory bounds.  Fails with [IOFailure](#_Heap) if the file could not be removed. |
| sys\_mkdir | 9 | Creates a new directory in the file system.  RBX should contain the address of the C string path to create.  Fails with [FSDisabled](#_Heap) if [FSF](#_Conditions) is not set.  Fails with [OutOfBounds](#_Heap) if the string went out of memory bounds.  Fails with [IOFailure](#_Heap) if the directory could not be created. |
| sys\_rmdir | 10 | Removes a directory from the file system.  RBX should contain the address of the C string path to remove.  Fails with [FSDisabled](#_Heap) if [FSF](#_Conditions) is not set.  Fails with [OutOfBounds](#_Heap) if the string went out of memory bounds.  Fails with [IOFailure](#_Heap) 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 RAX.  RBX should contain the return value to report. |

## NOP/FNOP – No Operation

OP Code:

00

Description:

Specifies no operation. Usually only used for timing purposes.

FNOP is used by the FPU, but in CSX64 is identical to NOP.

Usage:

NOP

FNOP

Flags Affected:

None.

Format:

[NOP]

## HLT – Halt Execution

OP Code:

01

Description:

Terminates the program with the [Abort](#_Heap) error code and yields to the virtual operating system.

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

Usage:

HLT

Flags Affected:

None.

Format:

[HLT]

## SYSCALL – System Call

OP Code:

02

Description:

Causes program execution to temporarily yield to the [virtual operating system](#_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](#_Heap) error.

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

Usage:

SYSCALL

Flags Affected:

Virtual operating system-dependent.

Format:

[SYSCALL]

## PUSHF/PUSHFD/PUSHFQ – Push Flags

OP Code:

03

Description:

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

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

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

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

Usage:

PUSHF

Flags Affected:

None.

Format:

[PUSHF] [8: ext]

ext = 0: push FLAGS

ext = 1: push EFLAGS

ext = 2: push RFLAGS

## POPF/POPFD/POPFQ – Pop Flags

OP Code:

04

Description:

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

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

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

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

Usage:

POPF

Flags Affected:

None.

Format:

[POPF] [8: ext]

ext = 0: pop FLAGS

ext = 1: pop EFLAGS

ext = 2: pop RFLAGS

## CLx/STx – Clear/Set flag

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

Description:

Clears (CLx) or sets (STx) a bit in the flags register. In compliance with standard x86\_64, not all flags are represented by this family of instructions.

Usage:

CLx/STx

Flags Affected:

None.

Format:

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

## SETcc – Conditional Set

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

Description:

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

Usage:

SETcc r8/m8

Flags Affected:

None.

Format:

[SETcc] [4: dest][2: size][1: h][1: mem]

mem = 0: dest <- f(dest)

mem = 1: [address] M[address] <- f(M[address])

(h marks AH, BH, CH, or DH for dest)

## MOV – Move

OP Code:

07

Description:

Copies a value from some source to a destination.

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

Usage:

MOV r, imm/r/m

MOV m, imm/r

Flags Affected:

None.

Format:

[MOV] [4: dest][2: size][1:dh][1: sh] [4: mode][4: src]

Mode = 0:

dest <- f(dest, src)

Mode = 1: [size: imm]

dest <- f(dest, imm)

Mode = 2: [address]

dest <- f(dest, M[address])

Mode = 3: [address]

M[address] <- f(M[address], src)

Mode = 4: [address] [size: imm]

M[address] <- f(M[address], imm)

Else UND

(dh and sh mark AH, BH, CH, or DH for dest or src)

## MOVcc – Conditional Move

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

Description:

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

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

Usage:

MOVcc r, imm/r/m

MOVcc m, imm/r

Flags Affected:

None.

Format:

[MOVcc] [4: dest][2: size][1:dh][1: sh] [4: mode][4: src]

Mode = 0:

dest <- f(dest, src)

Mode = 1: [size: imm]

dest <- f(dest, imm)

Mode = 2: [address]

dest <- f(dest, M[address])

Mode = 3: [address]

M[address] <- f(M[address], src)

Mode = 4: [address] [size: imm]

M[address] <- f(M[address], imm)

Else UND

(dh and sh mark AH, BH, CH, or DH for dest or src)

## XCHG – Exchange

OP Code:

09

Description:

Exchanges two values.

Does not support memory-to-memory transfers.

Usage:

XCHG r, r/m

XCHG m, r

Flags Affected:

None.

Format:

[XCHG] [4: r1][2: size][1: r1h][1: mem]

mem = 0: [1: r2h][3:][4: r2]

r1 🡨 r2

r2 🡨 r1

mem = 1: [address]

r1 🡨 M[address]

M[address] 🡨 r1

(r1h and r2h mark AH, BH, CH, or DH for r1 or r2)

## JMP – Jump

OP Code:

0a

Description:

Jumps execution to the provided address.

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

Usage:

JMP imm/r/m

Flags Affected:

None.

Format:

[JMP] [4: reg][2: size][2: mode]

mode = 0: reg

mode = 1: h reg (AH, BH, CH, or DH)

mode = 2: [size: imm] imm

mode = 3: [address] M[address]

## Jcc – Conditional Jump

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

Description:

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

These instructions form the basis for all conditional branching.

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

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

Usage:

Jcc imm/r/m

Flags Affected:

None.

Format:

As [JMP](#_JMP_–_Jump).

## LOOP/LOOPcc – Loop

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

Description:

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

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

Care should be taken that the count register is not initially zero, as the pre-decrement would cause an underflow and ultimately perform 264 loop iterations (which is very likely not what you intended). For this reason, it is common practice to use a [Jrcxz](#_Jcc_–_Conditional_1) variant to bypass the loop structure entirely in the case that the count starts at zero.

Usage:

LOOPcc imm/r/m

Flags Affected:

None.

Format:

As [JMP](#_Jcc_–_Conditional).

## CALL – Call

OP Code:

0d

Description:

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

This instruction is the backbone of all reasonable [calling conventions](#_Assembly_Process).

Usage:

CALL imm/r/m

Flags Affected:

None.

Format:

As [JMP](#_JMP_–_Jump).

## RET – Return

OP Code:

0e

Description:

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

Usage:

RET

Flags Affected:

None.

Format:

[RET]

## PUSH – Push

OP Code:

0f

Description:

Pushes a value onto [the stack](#_The_Stack). The size parameter determines the size of the value that is pushed.

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

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

Usage:

PUSH imm/r/m

Flags Affected:

None.

Format:

[PUSH] [4: reg][2: size][2: mode]

mode = 0: reg

mode = 1: h reg (AH, BH, CH, or DH)

mode = 2: [size: imm] imm

mode = 3: [address] M[address]

## POP – Pop

OP Code:

10

Description:

Removes a size byte value from [the stack](#_Stack).

Usage:

POP(:size) r

Flags Affected:

None.

Format:

[POP] [4: dest][2: size][1:][1: mem]

mem = 0:

pop value

dest 🡨 value

mem = 1: [address]

pop value

M[address] 🡨 value

## LEA – Load Effective Address

OP Code:

11

Description:

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

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

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

Usage:

LEA r, m

Flags Affected:

None.

Format:

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

dest 🡨 address

## ADD – Add

OP Code:

12

Description:

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

Usage:

ADD r, imm/r/m

ADD m, imm/r

Flags Affected:

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

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

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

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

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

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

Format:

[ADD] [4: dest][2: size][1:dh][1: sh] [4: mode][4: src]

Mode = 0:

dest 🡨 dest + src

Mode = 1: [size: imm]

dest 🡨 dest + imm

Mode = 2: [address]

dest 🡨 dest + M[address]

Mode = 3: [address]

M[address] 🡨 M[address] + src

Mode = 4: [address] [size: imm]

M[address] 🡨 M[address] + imm

Else UND

(dh and sh mark AH, BH, CH, or DH for dest or src)

## SUB – Subtract

OP Code:

13

Description:

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

SUB can and should be used in place of [CMP](#_CMP) if the result is needed.

Usage:

SUB r, imm/r/m

SUB m, imm/r

Flags Affected:

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

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

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

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

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

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

Format:

As [Add](#_ADD_–_Add_1).

## MUL – Unsigned Multiply

OP Code:

14

Description:

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

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

Usage:

MUL imm/r/m

Operation:

byte: AX 🡨 AL \* val

word: DX:AX 🡨 AX \* val

dword: EDX:EAX 🡨 EAX \* val

qword: RDX:RAX 🡨 RAX \* val

Flags Affected:

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

SF, ZF, AF, and PF are undefined.

Format:

[MUL] [4: reg][2: size][2: mode]

mode = 0: reg

mode = 1: h reg (AH, BH, CH, or DH)

mode = 2: [size: imm] imm

mode = 3: [address] M[address]

## IMUL – Signed Multiply

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

## DIV – Unsigned Divide

OP Code:

16

Description:

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

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

Dividing by zero or producing a quotient that cannot be stored in the operand register produces an [arithmetic error](#_Heap).

Usage:

DIV imm/r/m

Operation:

byte:

AX / val

AL 🡨 quotient, AH 🡨 remainder

word:

DX:AX / val

AX 🡨 quotient, DX 🡨 remainder

dword:

EDX:EAX / val

EAX 🡨 quotient, EDX 🡨 remainder

qword:

RDX:RAX / val

RAX 🡨 quotient, RDX 🡨 remainder

Flags Affected:

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

Format:

As [MUL](#_UMUL).

## IDIV – Signed Divide

OP Code:

17

Description:

As [DIV](#_UDIV) except that it performs signed division.

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

Usage:

IDIV imm/r/m

Operation:

byte:

AX / val

AL 🡨 quotient, AH 🡨 remainder

word:

DX:AX / val

AX 🡨 quotient, DX 🡨 remainder

dword:

EDX:EAX / val

AX 🡨 quotient, DX 🡨 remainder

qword:

RDX:RAX / val

AX 🡨 quotient, DX 🡨 remainder

Flags Affected:

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

Format:

As [MUL](#_UMUL).

## SHL – Shift Left

OP Code:

18

Description:

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

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

Vacated bits are filled with 0. This operation is identical to [SAL](#_SAL_–_Shift) but is a distinct operation to specify intent.

Usage:

SHL r, imm/r/m 8

SHL m, imm/r 8

Flags Affected:

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

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

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

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

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

AF is undefined.

Format:

As [ADD](#_ADD_–_Add_1) except the source is 8-bit regardless of size field.

## SHR – Shift Right

OP Code:

19

Description:

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

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

Vacated bits are filled with 0. For non-negative numbers, this instruction is equivalent to [SAR](#_SAR_–_Shift).

Usage:

SHR r, imm/r/m 8

SHR m, imm/r 8

Flags Affected:

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

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

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

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

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

AF is undefined.

Format:

As [ADD](#_ADD_–_Add_1) except the source is 8-bit regardless of size field.

## SAL – Shift Arithmetic Left

OP Code:

1a

Description:

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

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

Vacated bits are filled with 0. This operation is identical to [SHL](#_SHL_–_Shift) but is a distinct operation to specify intent.

Usage:

SAL r, imm/r/m 8

SAL m, imm/r 8

Flags Affected:

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

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

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

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

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

AF is undefined.

Format:

As [ADD](#_ADD_–_Add_1) except the source is 8-bit regardless of size field.

## SAR – Shift Arithmetic Right

OP Code:

1b

Description:

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

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

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

Usage:

SAR r, imm/r/m 8

SAR m, imm/r 8

Flags Affected:

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

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

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

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

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

AF is undefined.

Format:

As [ADD](#_ADD_–_Add_1) except the source is 8-bit regardless of size field.

## ROL – Rotate Left

OP Code:

1c

Description:

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

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

Usage:

ROL r, imm/r/m 8

ROL m, imm/r 8

Flags Affected:

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

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

Format:

As [ADD](#_ADD_–_Add_1) except the source is 8-bit regardless of size field.

## ROR – Rotate Right

OP Code:

1d

Description:

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

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

Usage:

ROR r, imm/r/m 8

ROR m, imm/r 8

Flags Affected:

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

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

Format:

As [ADD](#_ADD_–_Add_1) except the source is 8-bit regardless of size field.

## RCL – Rotate Carry Left

OP Code:

1e

Description:

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

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

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

Usage:

RCL r, imm/r/m 8

RCL m, imm/r 8

Flags Affected:

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

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

Format:

As [ADD](#_ADD_–_Add_1) except the source is 8-bit regardless of size field.

## RCR – Rotate Carry Right

OP Code:

1f

Description:

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

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

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

Usage:

RCR r, imm/r/m 8

RCR m, imm/r 8

Flags Affected:

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

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

Format:

As [ADD](#_ADD_–_Add_1) except the source is 8-bit regardless of size field.

## AND – Bitwise And

OP Code:

20

Description:

ANDs the destination with the source.

AND can and should be used in place of [TEST](#_TEST) if the result is needed.

Usage:

AND r, imm/r/m

AND m, imm/r

Flags Affected:

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

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

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

OF and CF are cleared.

AF is undefined.

Format:

As [ADD](#_ADD_–_Add_1).

## OR – Bitwise Or

OP Code:

21

Description:

ORs the destination with the source.

Usage:

OR r, imm/r/m

OR m, imm/r

Flags Affected:

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

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

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

OF and CF are cleared.

AF is undefined.

Format:

As [ADD](#_ADD_–_Add_1).

## XOR – Bitwise Exclusive Or

OP Code:

22

Description:

XORs the destination with the source.

Usage:

XOR r, imm/r/m

XOR m, imm/r

Flags Affected:

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

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

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

OF and CF are cleared.

AF is undefined.

Format:

As [ADD](#_ADD_–_Add_1).

## INC – Increment

OP Code:

23

Description:

Equivalent to using [ADD](#_ADD) with a value of 1 except that CF is preserved.

Usage:

INC r/m

Flags Affected:

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

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

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

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

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

Format:

[INC] [4: dest][2: size][1: dh][1: mem]

mem = 0:

dest 🡨 dest + 1

mem = 1: [address]

M[address] 🡨 M[address] + 1

(dh marks AH, BH, CH, or DH for dest)

## DEC – Decrement

OP Code:

24

Description:

Equivalent to using [SUB](#_SUB) with a value of 1 except that CF is preserved.

Usage:

DEC r/m

Flags Affected:

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

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

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

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

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

Format:

As [INC](#_CMP).

## NEG – Negate

OP Code:

25

Description:

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

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

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

Usage:

NEG r/m

Flags Affected:

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

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

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

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

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

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

Format:

As [INC](#_CMP).

## NOT – Bitwise Not

OP Code:

26

Description:

Performs a bitwise NOT on the destination.

Usage:

NOT r/m

Flags Affected:

None.

Format:

As [INC](#_CMP).

## CMP – Compare

OP Code:

27 (CMP), 28 (CMPZ)

Description:

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

This operation is identical to [SUB](#_SUB), but the result is discarded.

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

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

Usage:

CMP r, imm/r/m

CMP m, imm/r

Flags Affected:

As SUB.

Format:

As SUB.

## TEST – Logical Test

OP Code:

29

Description:

As [AND](#_AND), but the result is discarded.

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

Usage:

TEST r, imm/r/m

TEST m, imm/r

Flags Affected:

As AND.

Format:

As AND.

## BSWAP – Byte Swap

OP Code:

2a

Description:

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

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

Usage:

BSWAP r/m

Flags Affected:

None.

Format:

As [INC](#_CMP).

## BEXTR – Bitfield Extract

OP Code:

2b

Description:

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

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

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

Usage:

BEXTR r, imm/r/m

BEXTR m, imm/r

Flags Affected:

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

AF, SF, and PF are undefined.

All other ([public](#_Conditions)) flags are cleared.

Format:

As [ADD](#_ADD_–_Add_1) except the source is 16-bit regardless of size field.

## BLSI – Binary Lowest-Set Isolate

OP Code:

2c

Description:

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

Usage:

BLSI r/m

Flags Affected:

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

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

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

OF is cleared.

AF and PF are undefined.

Format:

As [INC](#_CMP).

## BLSMSK – Binary Lowest-Set Mask

OP Code:

2d

Description:

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

Usage:

BLSMSK r/m

Flags Affected:

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

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

ZF and OF are cleared.

AF and PF are undefined.

Format:

As [INC](#_CMP).

## BLSR – Binary Lowest-Set Reset

OP Code:

2e

Description:

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

Usage:

BLSR r/m

Flags Affected:

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

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

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

OF is cleared.

AF and PF are undefined.

Format:

As [INC](#_CMP).

## ANDN – Bitwise And Not

OP Code:

2f

Description:

Computes the bitwise AND of the inverted second operand and the third operand. The result is stored in the first operand.

This instruction is only available for 32 and 64-bit values.

Usage:

ANDN r32, r32, r/m32

ANDN r64, r64, r/m64

Flags Affected:

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

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

OF and CF are cleared.

AF and PF are undefined.

Format:

As [ADD](#_ADD_–_Add_1).

## BT / BTS / BTR / BTC – Bit Test

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

Description:

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

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

Usage:

BT r/m, imm/r 8

Flags Affected:

CF is loaded with the extracted bit.

OF, SF, AF, and PF are undefined.

Format:

As [ADD](#_ADD_–_Add_1) except the source is 8-bit regardless of size field.

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

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

Description:

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

Usage:

CWD/CDQ/DQO

Flags Affected:

None.

Format:

[op]

## MOVZX/MOVSX – Move Zero/Sign Extended

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

Description:

Loads a zero or sign extended value from memory. Complying with standard x86\_64, not all combinations of sizes are allowed (allowed size conversions are described below).

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

Usage:

MOVZX/MOVSX r, m

Flags Affected:

None.

Format:

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

mode = 0: 16 <- 8 Zero

mode = 1: 16 <- 8 Sign

mode = 2: 32 <- 8 Zero

mode = 3: 32 <- 16 Zero

mode = 4: 32 <- 8 Sign

mode = 5: 32 <- 16 Sign

mode = 6: 64 <- 8 Zero

mode = 7: 64 <- 16 Zero

mode = 8: 64 <- 8 Sign

mode = 9: 64 <- 16 Sign

else UND

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

## FLD\_const – Floating-Point Load Constant

|  |  |  |
| --- | --- | --- |
| **Mnemonic** | **OP Code** | **Constant Loaded** |
| FLD1 | 33 00 | 1.0 |
| FLDL2T | 33 01 | Log2(10) |
| FLDL2E | 33 02 | Log2(e) |
| FLDPI | 33 03 | **π** |
| FLDLG2 | 33 04 | Log10(2) |
| FLDLN2 | 33 05 | Loge(2) |
| FLDZ | 33 06 | 0.0 |

Description:

Pushes a constant value onto the FPU register stack.

Usage:

op

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

[op]

## FLD/FILD – Floating-Point Load

|  |  |  |
| --- | --- | --- |
| **Mnemonic** | **OP Code** | **Description** |
| FLD | 34 | Pushes a floating-point value onto the FPU register stack. |
| FILD | 34 | Pushes an integral value (converted to floating-point) onto the FPU register stack. |

|  |  |
| --- | --- |
| **Usage** | |
| **FLD** | **FILD** |
| FLD ST(i) | FILD m16int |
| FLD m32fp | FILD m32int |
| FLD m64fp | FILD m64int |

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

[1:][3: i][1:][3: mode]

mode = 0: push ST(i)

mode = 1: push m32fp

mode = 2: push m64fp

mode = 3: push m16int

mode = 4: push m32int

mode = 5: push m64int

else UND

## FST/FSTP/FIST/FISTP – Floating-Point Store

|  |  |  |
| --- | --- | --- |
| **Mnemonic** | **OP Code** | **Description** |
| FST | 35 | Stores ST0 to memory or an FPU register. Storing to an [“empty” FPU register](#_FPU_Registers) marks it as not empty and is not an error. |
| FSTP | As FST but also pops the FPU register stack afterwards. |
| FIST | 35 | Stores a truncated integral representation of ST0 to memory. |
| FISTP | As FIST but also pops the FPU register stack afterwards. |

|  |  |  |  |
| --- | --- | --- | --- |
| Usage | | | |
| FST | FSTP | FIST | FISTP |
| FST ST(i) | FSTP ST(i) | FIST m16int | FISTP m16int |
| FST m32fp | FSTP m32fp | FIST m32int | FISTP m32int |
| FST m64fp | FSTP m64fp |  | FISTP m64int |

Discussion:

Because storing to another FPU register allows the destination to be empty, this can be used to mark an FPU register is legal for use without having to use an FLD variant.

FSTP ST(0) is a common idiom to pop the FPU stack.

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

[1:][3: i][4: mode]

mode = 0: st(i) <- st(0)

mode = 1: || + pop

mode = 2: fp32M <- st(0)

mode = 3: || + pop

mode = 4: fp64M <- st(0)

mode = 5: || + pop

mode = 6: int16M <- st(0)

mode = 7: || + pop

mode = 8: int32M <- st(0)

mode = 9: || + pop

mode = 10: int64M <- st(0) + pop

else UND

## FXCH – Floating-Point Exchange

OP Code:

36

Description:

Swaps the contents of ST0 and ST(i). If ST(i) is not specified, it defaults to ST1.

Usage:

FXCH ST(i)

FXCH

Flags Affected:

C0, C2, and C3 are undefined.

C1 is cleared.

Format:

[FXCH] [5:][3: i]

## FMOVcc – Floating-Point Conditional Move

|  |  |  |  |
| --- | --- | --- | --- |
| **Mnemonic** | **Alias** | **OP Code** | **Condition** |
| FMOVe |  | 37 | Move if equal. |
| FMOVne |  | Move if not equal. |
| FMOVb | FMOVnae | Move if below. |
| FMOVbe | FMOVna | Move if below or equal. |
| FMOVa | FMOVnbe | Move if above. |
| FMOVae | FMOVnb | Move if above or equal. |
| FMOVu |  | Move if unordered (PF = 1). |
| FMOVnu |  | Move if not unordered (PF = 0). |

Description:

Conditionally moves ST(i) to ST0 based on the EFLAGS register.

Aside from the new U and NU conditions, the conditions correspond to the usual [flag combinations](#_Conditions) for integral CMP comparisons (floating-point comparison instructions set the flags in such a way that you use **unsigned conditionals**, not signed).

The U and NU conditions correspond to the “unordered” comparison result, which is produced by floating-point comparisons when either value is NaN.

Usage:

FMOVcc ST0, ST(i)

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

[1:][3: i][1:][3: cc]

cc = 0: E (=Z)

cc = 1: NE (=NZ)

cc = 2: B

cc = 3: BE

cc = 4: A

cc = 5: AE

cc = 6: U (=P)

cc = 7: NU (=NP)

## FADD/FADDP/FIADD – Floating-Point Add

|  |  |  |
| --- | --- | --- |
| **OP Code** | **Mnemonic** | **Description** |
| 38 | FADD *m32fp* | Add *m32fp* to ST(0) and store result in ST(0). |
| FADD *m64fp* | Add *m64fp* to ST(0) and store result in ST(0). |
| FADD ST(0), ST(i) | Add ST(0) to ST(i) and store result in ST(0). |
| FADD ST(i), ST(0) | Add ST(i) to ST(0) and store result in ST(i). |
| FADDP ST(i), ST(0) | Add ST(0) to ST(i), store result in ST(i), and pop the register stack. |
| FADDP | Add ST(0) to ST(1), store result in ST(1), and pop the register stack. |
| FIADD *m32int* | Add *m32int* to ST(0) and store result in ST(0). |
| FIADD *m16int* | Add *m16int* to ST(0) and store result in ST(0). |

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

[1: pop][3: i][1:][3: mode]

mode = 0: st(0) <- st(0) + st(i)

mode = 1: st(i) <- st(i) + st(0)

mode = 2: || + pop

mode = 3: st(0) <- st(0) + m32fp

mode = 4: st(0) <- st(0) + m64fp

mode = 5: st(0) <- st(0) + m16int

mode = 6: st(0) <- st(0) + m32int

else UND

## FSUB/FSUBP/FISUB – Floating-Point Subtract

|  |  |  |
| --- | --- | --- |
| **OP Code** | **Mnemonic** | **Description** |
| 39 | FSUB *m32fp* | Subtract *m32fp* from ST(0) and store result in ST(0). |
| FSUB *m64fp* | Subtract *m64fp* from ST(0) and store result in ST(0). |
| FSUB ST(0), ST(i) | Subtract ST(i) from ST(0) and store result in ST(0). |
| FSUB ST(i), ST(0) | Subtract ST(0) from ST(i) and store result in ST(i). |
| FSUBP ST(i), ST(0) | Subtract ST(0) from ST(i), store result in ST(i), and pop register stack. |
| FSUBP | Subtract ST(0) from ST(1), store result in ST(1), and pop register stack. |
| FISUB *m32int* | Subtract *m32int* from ST(0) and store result in ST(0). |
| FISUB *m16int* | Subtract *m16int* from ST(0) and store result in ST(0). |

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

As [FADD](#_FADD/FADDP/FIADD_–_Floating-Point).

## FSUBR/FSUBRP/FISUBR – Floating-Point Subtract

|  |  |  |
| --- | --- | --- |
| **OP Code** | **Mnemonic** | **Description** |
| 3a | FSUBR *m32fp* | Subtract ST(0) from *m32fp* and store result in ST(0). |
| FSUBR *m64fp* | Subtract ST(0) from *m64fp* and store result in ST(0). |
| FSUBR ST(0), ST(i) | Subtract ST(0) from ST(i) and store result in ST(0). |
| FSUBR ST(i), ST(0) | Subtract ST(i) from ST(0) and store result in ST(i). |
| FSUBRP ST(i), ST(0) | Subtract ST(i) from ST(0), store result in ST(i), and pop register stack. |
| FSUBRP | Subtract ST(1) from ST(0), store result in ST(1), and pop register stack. |
| FISUBR *m32int* | Subtract ST(0) from *m32int* and store result in ST(0). |
| FISUBR *m16int* | Subtract ST(0) from *m16int* and store result in ST(0). |

Description:

Performs the same operation as its FSUB equivalent except that, where an FSUB variant would compute A-B, FSUBR would compute B-A. This is often done to avoid unnecessary register shuffling.

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

As [FADD](#_FADD/FADDP/FIADD_–_Floating-Point).

## FMUL/FMULP/FIMUL – Floating-Point Multiply

|  |  |  |
| --- | --- | --- |
| **OP Code** | **Mnemonic** | **Description** |
| 3b | FMUL *m32fp* | Multiply ST(0) by *m32fp* and store result in ST(0). |
| FMUL *m64fp* | Multiply ST(0) by *m64fp* and store result in ST(0). |
| FMUL ST(0), ST(i) | Multiply ST(0) by ST(i) and store result in ST(0). |
| FMUL ST(i), ST(0) | Multiply ST(i) by ST(0) and store result in ST(i). |
| FMULP ST(i), ST(0) | Multiply ST(i) by ST(0), store result in ST(i), and pop the register stack. |
| FMULP | Multiply ST(1) by ST(0), store result in ST(1), and pop the register stack. |
| FIMUL *m32int* | Multiply ST(0) by *m32int* and store result in ST(0). |
| FIMUL *m16int* | Multiply ST(0) by *m16int* and store result in ST(0). |

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

As [FADD](#_FADD/FADDP/FIADD_–_Floating-Point).

## FDIV/FDIVP/FIDIV – Floating-Point Divide

|  |  |  |
| --- | --- | --- |
| **OP Code** | **Mnemonic** | **Description** |
| 3c | FDIV *m32fp* | Divide ST(0) by *m32fp* and store result in ST(0). |
| FDIV *m64fp* | Divide ST(0) by *m64fp* and store result in ST(0). |
| FDIV ST(0), ST(i) | Divide ST(0) by ST(i) and store result in ST(0). |
| FDIV ST(i), ST(0) | Divide ST(i) by ST(0) and store result in ST(i). |
| FDIVP ST(i), ST(0) | Divide ST(i) by ST(0), store result in ST(i), and pop the register stack. |
| FDIVP | Divide ST(1) by ST(0), store result in ST(1), and pop the register stack. |
| FIDIV *m32int* | Divide ST(0) by *m32int* and store result in ST(0). |
| FIDIV *m16int* | Divide ST(0) by *m16int* and store result in ST(0). |

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

As [FADD](#_FADD/FADDP/FIADD_–_Floating-Point).

## FDIVR/FDIVRP/FIDIVR – Floating-Point Subtract

|  |  |  |
| --- | --- | --- |
| **OP Code** | **Mnemonic** | **Description** |
| 3d | FDIVR *m32fp* | Divide *m32fp* by ST(0) and store result in ST(0). |
| FDIVR *m64fp* | Divide *m64fp* by ST(0) and store result in ST(0). |
| FDIVR ST(0), ST(i) | Divide ST(i) by ST(0) and store result in ST(0). |
| FDIVR ST(i), ST(0) | Divide ST(0) by ST(i) and store result in ST(i). |
| FDIVRP ST(i), ST(0) | Divide ST(0) by ST(i), store result in ST(i), and pop the register stack. |
| FDIVRP | Divide ST(0) by ST(1), store result in ST(1), and pop the register stack. |
| FIDIVR *m32int* | Divide *m32int* by ST(0) and store result in ST(0). |
| FIDIVR *m16int* | Divide *m16int* by ST(0) and store result in ST(0). |

Description:

Performs the same operation as its FDIV equivalent except that, where an FDIV variant would compute A/B, FDIVR would compute B/A. This is often done to avoid unnecessary register shuffling.

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

As [FADD](#_FADD/FADDP/FIADD_–_Floating-Point).

## F2XM1 – Floating-Point 2x-1

OP Code:

3e

Description:

Computes 2ST(0)-1 and stores the result in ST(0). ST(0) must be in the range [-1,1]. If the source value is outside of this range CSX64 generates an [FPUError](#_Flags) error.

This instruction is typically used as a means of performing exponentiation, where the value to exponentiate is split into its integral and fractional components and this instruction is then used to handle the fractional component.

Usage:

F2XM1

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

[F2XM1]

## FABS – Floating-Point Absolute Value

OP Code:

3f

Description:

Clears the sign bit of ST(0), effectively taking its absolute value.

Usage:

FABS

Flags Affected:

C0, C2, and C3 are undefined.

C1 is cleared.

Format:

[FABS]

## FCHS – Floating-Point Change Sign

OP Code:

40

Description:

Flips the sign bit of ST(0), effectively taking its negative.

Usage:

FCHS

Flags Affected:

C0, C2, and C3 are undefined.

C1 is cleared.

Format:

[FCHS]

## FPREM – Floating-Point Partial Remainder

OP Code:

41

Description:

Computes the remainder of dividing ST(0) by ST(1) and stores the result in ST(0).

The remainder is calculated as ST(0) – Q \* ST(1) where Q is ST(0) / ST(1) after being truncated.

Usage:

FPREM

Flags Affected:

C0 is set to bit 2 of the quotient.

C1 is set to bit 0 of the quotient.

C2 is cleared.

C3 is set to bit 1 of the quotient.

Format:

[FPREM]

## FPREM1 – Floating-Point Partial Remainder

OP Code:

42

Description:

Computes the IEEE remainder of dividing ST(0) by ST(1) and stores the result in ST(0).

The remainder is calculated as ST(0) – Q \* ST(1) where Q is ST(0) / ST(1) after being rounded to the nearest integer.

Usage:

FPREM1

Flags Affected:

C0 is set to bit 2 of the quotient.

C1 is set to bit 0 of the quotient.

C2 is cleared.

C3 is set to bit 1 of the quotient.

Format:

[FPREM1]

## FRNDINT – Floating-Point Round to Integer

OP Code:

43

Description:

Rounds ST(0) to the nearest integer and stores the result in ST(0).

Usage:

FRNDINT

Flags Affected:

C0, C2, and C3 are undefined.

C1 is set if the value was rounded up and cleared otherwise.

Format:

[FRNDINT]

## FSQRT – Floating-Point Square Root

OP Code:

44

Description:

Computes the square root of ST(0) and stores the result in ST(0).

Usage:

FSQRT

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

[FSQRT]

## FYL2X – Floating-Point Y\*Log2(X)

OP Code:

45

Description:

Computes ST(1) \* Log2(ST(0)), stores the result in ST(1), and pops the FPU stack.

The purpose of multiplying by ST(1) is so logarithms of other bases can taken with one instruction.

Usage:

FYL2X

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

[FYL2X]

## FYL2XP1 – Floating-Point Y\*Log2(X+1)

OP Code:

46

Description:

Computes ST(1) \* Log2(ST(0)+1), stores the result in ST(1), and pops the FPU stack.

The purpose of multiplying by ST(1) is so logarithms of other bases can taken with one instruction.

Usage:

FYL2XP1

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

[FYL2XP1]

## FXTRACT – Floating-Point Extract

OP Code:

47

Description:

Extracts the exponent and significand of ST(0). The exponent is stored in the current ST(0) and the significand is pushed onto the FPU stack.

Usage:

FXTRACT

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

[FXTRACT]

## FSCALE – Floating-Point Scale

OP Code:

48

Description:

Adds ST(1) – after truncation – to the exponent of ST(0), enabling rapid multiplication by powers of 2.

FSCALE can be thought of as the inverse of [FXTRACT](#_FXTRACT_–_Floating-Point), though FSCALE does not pop the stack.

Usage:

FSCALE

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

[FSCALE]

## FXAM – Floating-Point Examine

OP Code:

49

Description:

Examines the contents of ST(0) and sets the FPU flags based on the contents.

Usage:

FXAM

Flags Affected:

|  |  |  |  |
| --- | --- | --- | --- |
| **Value** | **C3** | **C2** | **C0** |
| Unsupported | 0 | 0 | 0 |
| NaN | 0 | 0 | 1 |
| Normal finite number | 0 | 1 | 0 |
| Infinity | 0 | 1 | 1 |
| Zero | 1 | 0 | 0 |
| Empty | 1 | 0 | 1 |
| Denormal number | 1 | 1 | 0 |

C1 is set to the sign bit of the value, regardless of if ST(0) was empty or full.

Format:

[FXAM]

## FTST – Floating-Point Test

OP Code:

4a

Description:

Compares the contents of ST(0) with 0.0 and sets the FPU flags accordingly.

If ST(0) currently contains NaN, the result is “unordered” and an [ArithmeticError](#_Flags) is generated.

Usage:

FTST

Flags Affected:

|  |  |  |  |
| --- | --- | --- | --- |
| **Condition** | **C3** | **C2** | **C0** |
| ST(0) > 0.0 | 0 | 0 | 0 |
| ST(0) < 0.0 | 0 | 0 | 1 |
| ST(0) = 0.0 | 1 | 0 | 0 |
| Unordered | 1 | 1 | 1 |

C1 is cleared.

Format:

[FTST]

## FCOM/FCOMP/FCOMPP/FICOM/FICOMP – Floating-Point Compare

|  |  |  |
| --- | --- | --- |
| **OP Code** | **Mnemonic** | **Description** |
| 4b | FCOM *m32fp* | Compare ST(0) with *m32fp*. |
| FCOM *m64fp* | Compare ST(0) with *m64fp*. |
| FCOM ST(i) | Compare ST(0) with ST(i). |
| FCOM | Compare ST(0) with ST(1). |
| FCOMP *m32fp* | Compare ST(0) with *m32fp* and pop register stack. |
| FCOMP *m64fp* | Compare ST(0) with m64fp and pop register stack. |
| FCOMP ST(i) | Compare ST(0) with ST(i) and pop register stack. |
| FCOMP | Compare ST(0) with ST(1) and pop register stack. |
| FCOMPP | Compare ST(0) with ST(1) and pop register stack twice. |
| FICOM *m16int* | Compare ST(0) with *m16int.* |
| FICOM *m32int* | Compare ST(0) with *m32int.* |
| FICOMP *m16int* | Compare ST(0) with *m16int* and pop stack register. |
| FICOMP *m32int* | Compare ST(0) with *m32int* and pop stack register. |

Description:

Sets the FPU flags according to the values of ST(0) and some other source.

If either value is NaN, the result is unordered.

Flags Affected:

|  |  |  |  |
| --- | --- | --- | --- |
| **Condition** | **C3** | **C2** | **C0** |
| ST(0) > src | 0 | 0 | 0 |
| ST(0) < src | 0 | 0 | 1 |
| ST(0) = src | 1 | 0 | 0 |
| Unordered | 1 | 1 | 1 |

C1 is cleared.

Format:

[1:][3: i][4: mode]

mode = 0: cmp st(0), st(i)

mode = 1: || + pop

mode = 2: || + 2x pop

mode = 3: cmp st(0), fp32

mode = 4: || + pop

mode = 5: cmp st(0), fp64

mode = 6: || + pop

mode = 7: cmp st(0), int16

mode = 8: || + pop

mode = 9: cmp st(0), int32

mode = 10: || + pop

else UND

## FUCOM/FUCOMP/FUCOMPP – Floating-Point Compare

4c

!! NOT IMPLEMENTED !!

Description:

Performs the same action as FCOM except that an [ArithmeticError](#_Flags) is only generated if either value is [SNaN](#_64-bit), whereas FCOM raises an error on any NaN.

Flags Affected:

|  |  |  |  |
| --- | --- | --- | --- |
| **Condition** | **C3** | **C2** | **C0** |
| ST(0) > src | 0 | 0 | 0 |
| ST(0) < src | 0 | 0 | 1 |
| ST(0) = src | 1 | 0 | 0 |
| Unordered | 1 | 1 | 1 |

C1 is cleared.

Format:

## FCOMI/FCOMIP/FUCOMI/FUCOMIP – Floating-Point EFLAGS Compare

|  |  |  |
| --- | --- | --- |
| **OP Code** | **Mnemonic** | **Description** |
| 4d | FCOMI ST, ST(i) | Compare ST(0) with ST(i) and set status flags accordingly. |
| FCOMIP ST, ST(i) | Compare ST(0) with ST(i), set status flags accordingly, and pop register stack. |
| FUCOMI ST, ST(i) | Compare ST(0) with ST(i), check for ordered values, and set status flags accordingly. |
| FUCOMIP ST, ST(i) | Compare ST(0) with ST(i), check for ordered values, set status flags accordingly, and pop register stack. |

Description:

Compares the contents of ST(0) to ST(i) and sets EFLAGS accordingly. The manner in which this instruction modifies EFLAGS makes it so ***unsigned*** arithmetic [conditionals](#_Conditions) can be used afterwards.

Because these instructions modify EFLAGS rather than the FPU flags, these instructions are typically used much more commonly than any of the other floating-point comparison instructions.

FCOMI and FCOMIP trigger an [ArithmeticError](#_Flags) if either value is NaN.

FUCOMI and FUCOMIP only trigger an ArithmeticError if either value is [SNaN](#_64-bit).

As PF is only set on the unordered condition, PF can be used to catch if either value was NaN. Thus, FUCOMI ST(0), ST(0) can be used to test if ST(0) is NaN.

Flags Affected:

|  |  |  |  |
| --- | --- | --- | --- |
| **Condition** | **ZF** | **PF** | **CF** |
| ST(0) > ST(i) | 0 | 0 | 0 |
| ST(0) < ST(i) | 0 | 0 | 1 |
| ST(0) = ST(i) | 1 | 0 | 0 |
| Unordered | 1 | 1 | 1 |

C1 is cleared.

Format:

[1: pop][1: unordered][2:][1:][3: i]

cmp st(0), st(i)

## FSIN – Floating-Point Sine

OP Code:

4e

Description:

Computes the sine of ST(0) and stores the result in ST(0).

Angles are in radians.

Usage:

FSIN

Flags Affected:

C0, C1, and C3 are undefined.

C2 is cleared.

Format:

[FSIN]

## FCOS – Floating-Point Cosine

OP Code:

4f

Description:

Computes the cosine of ST(0) and stores the result in ST(0).

Angles are in radians.

Usage:

FCOS

Flags Affected:

C0, C1, and C3 are undefined.

C2 is cleared.

Format:

[FCOS]

## FSINCOS – Floating-Point Sine and Cosine

OP Code:

50

Description:

Computes the sine and cosine of ST(0). The sine is stored in the previous ST(0) and the cosine is pushed onto the FPU stack.

Angles are in radians.

Usage:

FSINCOS

Flags Affected:

C0, C1, and C3 are undefined.

C2 is cleared.

Format:

[FSINCOS]

## FPTAN – Floating-Point Partial Tangent

OP Code:

51

Description:

Computes the tangent of ST(0) and stores the result in ST(0). Also pushes 1.0 onto the FPU stack to be backwards compatible with older floating-point coprocessors.

Angles are in radians.

Usage:

FPTAN

Flags Affected:

C0, C1, and C3 are undefined.

C2 is cleared.

Format:

[FPTAN]

## FPATAN – Floating-Point Partial Arctangent

OP Code:

52

Description:

Computes the arctangent of ST(1)/ST(0) – preserving the correct quadrant – and stores the result in ST(1) before popping the stack.

Angles are in radians. The calculated angle is in the range [-π, π], depending on the quadrant.

FPATAN can be thought of as the inverse of FPTAN due to FPTAN pushing 1.0 onto the stack and FPATAN popping the stack after the computation.

Usage:

FPATAN

Flags Affected:

C0, C1, and C3 are undefined.

C2 is cleared.

Format:

[FPATAN]

## FINCSTP/FDECSTP – Floating-Point Increment/Decrement Stack

OP Code:

53

Description:

Increments or decrements the [FPU stack pointer](#_Floating-Point_Unit_(FPU)) but does not change the empty/full status of any registers.

Usage:

FINCSTP

FDECSTP

Flags Affected:

C0, C2, and C3 are undefined.

C1 is cleared.

Format:

[op] [7:][1: mode]

mode = 0: inc

mode = 1: dec

## FFREE – Floating-Point Free Register

OP Code:

54

Description:

Marks the specified FPU register as empty.

This is a convenient instruction in the case that you used [FST](#_FST/FSTP/FIST/FISTP_–_Floating-Poin) to mark a register as having a value rather than pushing it onto the FPU stack with [FLD](#_FLD/FILD_–_Floating-Point). This is also useful for cleaning up the FPU stack after a function that didn’t – or may not – properly clean up of its own accord (e.g. after an exception).

Usage:

FFREE ST(i)

Flags Affected:

C0, C1, C2, and C3 are undefined.

Format:

[op] [5:][3: i]

free st(i)

## MOVQ/MOVSD – Move Quadword / Move Scalar Double

OP Code:

55

Description:

Performs a scalar vector move between vector registers (of equal size) or between a vector register and a memory location. The value to be moved is 64-bit. Although these instructions are often synonymous, depending on architecture it may improve throughput to use both forms where appropriate.

Usage:

MOVQ/MOVSD r128/256/512, r128/256/512

MOVQ/MOVSD r128/256/512, m64

MOVQ/MOVSD m64, r128/256/512

Flags Affected:

None.

Format:

[5: reg][1: aligned][2: reg\_size]

[1: has\_mask][1: zmask][1: scalar][1:][2: elem\_size][2: mode]

([count: mask])

mode = 0: [3:][5: src] reg <- src

mode = 1: [address] reg <- M[address]

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

else UND

## MOVD/MOVSS – Move Doubleword / Move Scalar Single

OP Code:

55

Description:

Performs a scalar vector move between vector registers (of equal size) or between a vector register and a memory location. The value to be moved is 32-bit. Although these instructions are often synonymous, depending on architecture it may improve throughput to use both forms where appropriate.

Usage:

MOVD/MOVSS r128/256/512, r128/256/512

MOVD/MOVSS r128/256/512, m32

MOVD/MOVSS m32, r128/256/512

Flags Affected:

None.

Format:

As [MOVQ](#_MOVQ/MOVSD_–_Move).

## MOVDQA/MOVDQU – Move (Aligned/Unaligned) Packed Integers

OP Code:

55

Description:

Performs a vector move between vector registers (of equal size) or between a vector register and a memory location. The total width of the vector to be moved is the same size as the vector register used. The aligned option is significantly faster than the unaligned equivalent but requires the memory operand be aligned to the width of the vector register.

The (possibly confusing) name is historical. At one point, when these instructions were being standardized, only xmm registers existed in (typical) vector processors (no ymm or zmm). Because xmm registers can hold 2 quadwords, their full size was named double quadword, hence the name. However, MOVDQA/MOVDQU became synonymous with “move a full vector of stuff”, so when larger vector registers began to become commonplace, the name stuck even for these higher sizes, despite the disconnect between name and meaning.

Usage:

MOVDQA/MOVDQU r128/256/512, r128/256/512

MOVDQA/MOVDQU r128/256/512, m128/256/512

MOVDQA/MOVDQU m128/256/512, r128/256/512

Flags Affected:

None.

Format:

As [MOVQ](#_MOVQ/MOVSD_–_Move).

## MOVDQA64/32/16/8 – Move Aligned Packed Integers

OP Code:

55

Description:

Performs a (potentially masked) vector move between vector registers (of equal size) or between a vector register and a memory location. The total width of the vector to be moved is the same size as the vector register used, and each element has the size (in bits) as specified by the instruction used.

Usage:

MOVDQA64/32/16/8 r128/256/512 {k}z, r128/256/512

MOVDQA64/32/16/8 r128/256/512 {k}z, m128/256/512

MOVDQA64/32/16/8 m128/256/512 {k}z, r128/256/512

Flags Affected:

None.

Format:

As [MOVQ](#_MOVQ/MOVSD_–_Move).

## MOVDQU64/32/16/8 – Move Unaligned Packed Integers

OP Code:

55

Description:

Performs a (potentially masked) vector move between vector registers (of equal size) or between a vector register and a memory location. The total width of the vector to be moved is the same size as the vector register used, and each element has the size (in bits) as specified by the instruction used. The aligned option is significantly faster than the unaligned equivalent but requires the memory operand be aligned to the width of the vector register.

Usage:

MOVDQU64/32/16/8 r128/256/512 {k}z, r128/256/512

MOVDQU64/32/16/8 r128/256/512 {k}z, m128/256/512

MOVDQU64/32/16/8 m128/256/512 {k}z, r128/256/512

Flags Affected:

None.

Format:

As [MOVQ](#_MOVQ/MOVSD_–_Move).

## MOVAPD/MOVUPD – Move (Aligned/Unaligned) Packed Doubles

OP Code:

55

Description:

Performs a (potentially masked) vector move between vector registers (of equal size) or between a vector register and a memory location. The total width of the vector to be moved is the same size as the vector register used, and each element is 64-bit. The aligned option is significantly faster than the unaligned equivalent but requires the memory operand be aligned to the width of the vector register.

Usage:

MOVDQAPD/MOVUPD r128/256/512 {k}z, r128/256/512

MOVDQAPD/MOVUPD r128/256/512 {k}z, m128/256/512

MOVDQAPD/MOVUPD m128/256/512 {k}z, r128/256/512

Flags Affected:

None.

Format:

As [MOVQ](#_MOVQ/MOVSD_–_Move).

## MOVAPS/MOVUPS – Move (Aligned/Unaligned) Packed Singles

OP Code:

55

Description:

Performs a (potentially masked) vector move between vector registers (of equal size) or between a vector register and a memory location. The total width of the vector to be moved is the same size as the vector register used, and each element is 32-bit. The aligned option is significantly faster than the unaligned equivalent but requires the memory operand be aligned to the width of the vector register.

Usage:

MOVDQAPS/MOVUPS r128/256/512 {k}z, r128/256/512

MOVDQAPS/MOVUPS r128/256/512 {k}z, m128/256/512

MOVDQAPS/MOVUPS m128/256/512 {k}z, r128/256/512

Flags Affected:

None.

Format:

As [MOVQ](#_MOVQ/MOVSD_–_Move).

## ADDSD – Add Scalar Double

OP Code:

56

Description:

Performs the scalar addition of 2 double-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

ADDSD r128/256/512, r128/256/512

ADDSD r128/256/512, m64

ADDSD r128/256/512, r128/256/512, r128/256/512

ADDSD r128/256/512, r128/256/512, m64

Flags Affected:

None.

Format:

[5: dest][1: aligned][2: dest\_size]

[1: has\_mask][1: zmask][1: scalar][1:][2: elem\_size][1:][1: mem]

([count: mask]) [3:][5: src1]

mem = 0: [3:][5: src2] dest <- f(src1, src2)

mem = 1: [address] dest <- f(src1, M[address])

## SUBSD – Subtract Scalar Double

OP Code:

57

Description:

Performs the scalar subtraction of 2 double-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

SUBSD r128/256/512, r128/256/512

SUBSD r128/256/512, m64

SUBSD r128/256/512, r128/256/512, r128/256/512

SUBSD r128/256/512, r128/256/512, m64

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## MULSD – Multiply Scalar Double

OP Code:

58

Description:

Performs the scalar multiplication of 2 double-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

MULSD r128/256/512, r128/256/512

MULSD r128/256/512, m64

MULSD r128/256/512, r128/256/512, r128/256/512

MULSD r128/256/512, r128/256/512, m64

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## DIVSD – Divide Scalar Double

OP Code:

59

Description:

Performs the scalar division of 2 double-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

DIVSD r128/256/512, r128/256/512

DIVSD r128/256/512, m64

DIVSD r128/256/512, r128/256/512, r128/256/512

DIVSD r128/256/512, r128/256/512, m64

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## ADDSS – Add Scalar Single

OP Code:

56

Description:

Performs the scalar addition of 2 single-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

ADDSS r128/256/512, r128/256/512

ADDSS r128/256/512, m32

ADDSS r128/256/512, r128/256/512, r128/256/512

ADDSS r128/256/512, r128/256/512, m32

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## SUBSS – Subtract Scalar Single

OP Code:

57

Description:

Performs the scalar subtraction of 2 single-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

SUBSS r128/256/512, r128/256/512

SUBSS r128/256/512, m32

SUBSS r128/256/512, r128/256/512, r128/256/512

SUBSS r128/256/512, r128/256/512, m32

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## MULSS – Multiply Scalar Single

OP Code:

58

Description:

Performs the scalar multiplication of 2 single-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

MULSS r128/256/512, r128/256/512

MULSS r128/256/512, m32

MULSS r128/256/512, r128/256/512, r128/256/512

MULSS r128/256/512, r128/256/512, m32

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## DIVSS – Add Scalar Single

OP Code:

59

Description:

Performs the scalar division of 2 single-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

DIVSS r128/256/512, r128/256/512

DIVSS r128/256/512, m32

DIVSS r128/256/512, r128/256/512, r128/256/512

DIVSS r128/256/512, r128/256/512, m32

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## ADDPD – Add Packed Doubles

OP Code:

56

Description:

Performs the addition of double-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

ADDPD r128/256/512, r/m128/256/512

ADDPD r128/256/512, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## SUBPD – Subtract Packed Doubles

OP Code:

57

Description:

Performs the subtraction of double-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

SUBPD r128/256/512, r/m128/256/512

SUBPD r128/256/512, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## MULPD – Multiply Packed Doubles

OP Code:

58

Description:

Performs the multiplication of double-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

MULPD r128/256/512, r/m128/256/512

MULPD r128/256/512, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## DIVPD – Divide Packed Doubles

OP Code:

59

Description:

Performs the division of double-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

DIVPD r128/256/512, r/m128/256/512

DIVPD r128/256/512, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## ADDPS – Add Packed Singles

OP Code:

56

Description:

Performs the addition of single-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

ADDPS r128/256/512, r/m128/256/512

ADDPS r128/256/512, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## SUBPS – Subtract Packed Singles

OP Code:

57

Description:

Performs the subtraction of single-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

SUBPS r128/256/512, r/m128/256/512

SUBPS r128/256/512, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## MULPS – Multiply Packed Singles

OP Code:

58

Description:

Performs the multiplication of single-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

MULPS r128/256/512, r/m128/256/512

MULPS r128/256/512, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## DIVPS – Divide Packed Singles

OP Code:

56

Description:

Performs the division of single-precision floating-point values.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

DIVPS r128/256/512, r/m128/256/512

DIVPS r128/256/512, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PAND – Packed Bitwise AND

OP Code:

5a

Description:

Performs a packed bitwise AND.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PAND r128/256/512, r/m128/256/512

PAND r128/256/512, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## POR – Packed Bitwise OR

OP Code:

5b

Description:

Performs a packed bitwise OR.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

POR r128/256/512, r/m128/256/512

POR r128/256/512, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PXOR – Packed Bitwise XOR

OP Code:

5c

Description:

Performs a packed bitwise XOR.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PXOR r128/256/512, r/m128/256/512

PXOR r128/256/512, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PANDN – Packed Bitwise ANDN

OP Code:

5d

Description:

Performs a packed bitwise AND of the inverted left operand and the right operand.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PANDN r128/256/512, r/m128/256/512

PANDN r128/256/512, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PANDQ/ANDPD – Packed Bitwise AND of Quadwords or Doubles

OP Code:

5a

Description:

Performs a packed bitwise AND. This version allows masking.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PANDQ/ANDPD r128/256/512 {k}z, r/m128/256/512

PANDQ/ANDPD r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PORQ/ORPD – Packed Bitwise OR of Quadwords or Doubles

OP Code:

5b

Description:

Performs a packed bitwise OR. This version allows masking.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PORQ/ORPD r128/256/512 {k}z, r/m128/256/512

PORQ/ORPD r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PXORQ/XORPD – Packed Bitwise XOR of Quadwords or Doubles

OP Code:

5c

Description:

Performs a packed bitwise XOR. This version allows masking.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PXORQ/XORPD r128/256/512 {k}z, r/m128/256/512

PXORQ/XORPD r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PANDNQ/ANDNPD – Packed Bitwise ANDN of Quadwords or Doubles

OP Code:

5d

Description:

Performs a packed bitwise AND of the inverted left operand and the right operand. This version allows masking.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PANDNQ/ANDNPD r128/256/512 {k}z, r/m128/256/512

PANDNQ/ANDNPD r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PANDD/ANDPS – Packed Bitwise AND of Doublewords or Singles

OP Code:

5a

Description:

Performs a packed bitwise AND. This version allows masking.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PANDD/ANDPS r128/256/512 {k}z, r/m128/256/512

PANDD/ANDPS r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PORD/ORPS – Packed Bitwise OR of Doublewords or Singles

OP Code:

5b

Description:

Performs a packed bitwise OR. This version allows masking.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PORD/ORPS r128/256/512 {k}z, r/m128/256/512

PORD/ORPS r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PXORD/XORPS – Packed Bitwise XOR of Doublewords or Singles

OP Code:

5c

Description:

Performs a packed bitwise XOR. This version allows masking.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PXORD/XORPS r128/256/512 {k}z, r/m128/256/512

PXORD/XORPS r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PANDND/ANDNPS – Packed Bitwise ANDN of Doublewords or Singles

OP Code:

5d

Description:

Performs a packed bitwise AND of the inverted left operand and the right operand. This version allows masking.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PANDND/ANDNPS r128/256/512 {k}z, r/m128/256/512

PANDND/ANDNPS r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PADDQ/PADDD/PADDW/PADDB – Packed Addition of Integers

OP Code:

5e

Description:

Performs the addition of integers. Allows masking.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PADDQ r128/256/512 {k}z, r/m128/256/512

PADDQ r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PADDSW/PADDSB – Packed Saturated Addition of Integers

OP Code:

5f

Description:

Performs the saturated addition of integers. Allows masking.

Saturated addition is defined as capping on signed integral limits rather than over/underflowing (i.e. numbers that overflow will instead be “saturated” to the highest positive value and numbers that underflow will instead be “saturated” to the lowest negative value).

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PADDSW r128/256/512 {k}z, r/m128/256/512

PADDSW r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PADDUSW/PADDUSB – Packed Unsigned Saturated Addition of Integers

OP Code:

60

Description:

Performs the unsigned saturated addition of integers. Allows masking.

Unsigned saturated addition is defined as capping on unsigned integral limits rather than overflowing (i.e. numbers that overflow will instead be “saturated” to the largest value).

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PADDUSW r128/256/512 {k}z, r/m128/256/512

PADDUSW r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PSUBQ/PSUBD/PSUBW/PSUBB – Packed Subtraction of Integers

OP Code:

61

Description:

Performs the subtraction of integers. Allows masking.

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PSUBQ r128/256/512 {k}z, r/m128/256/512

PSUBQ r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PSUBSW/PSUBSB – Packed Saturated Subtraction of Integers

OP Code:

62

Description:

Performs the saturated subtraction of integers. Allows masking.

Saturated subtraction is defined as capping on signed integral limits rather than over/underflowing (i.e. numbers that overflow will instead be “saturated” to the highest positive value and numbers that underflow will instead be “saturated” to the lowest negative value).

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PSUBSW r128/256/512 {k}z, r/m128/256/512

PSUBSW r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## PSUBUSW/PSUBUSB – Packed Unsigned Saturated Subtraction of Integers

OP Code:

63

Description:

Performs the unsigned saturated subtraction of integers. Allows masking.

Unsigned saturated subtraction is defined as capping on unsigned integral limits rather than overflowing (i.e. numbers that underflow will instead be “saturated” to zero).

When given 2 operands, the result is stored in the first operand as per usual. When given 3 operands, the values are taken from operands 2 and 3 and the result is stored in the first operand.

Usage:

PSUBUSW r128/256/512 {k}z, r/m128/256/512

PSUBUSW r128/256/512 {k}z, r128/256/512, r/m128/256/512

Flags Affected:

None.

Format:

As [ADDSD](#_ADDSD_–_Add).

## DEBUG – Debug

|  |  |  |
| --- | --- | --- |
| **OP Code** | **Mnemonic** | **Description** |
| ff 00 | DEBUG\_CPU | Displays debugging info on all the non-vpu registers |
| ff 01 | DEBUG\_VPU | Displays debugging info on vpu registers |
| ff 02 | DEBUG\_FULL | Equivalent to using CPU and VPU debugging |

Description:

Writes a debugging string to stdout, containing state information on all registers.

This instruction family is included as a means of retrieving the contents of registers as they currently exist without the overhead of having to tediously push/pop them in order to use an alternative method (e.g. C’s printf function).

Note that the behavior of this instruction is not changed by closing the stdout file, as explained in the section on [Unmanaged Files](#_Unmanaged_File).

Usage:

DEBUG

Flags Affected:

None.

Format:

[DEBUG]

# 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** | | | |
| 103 | 102 | 101 | 100 | 23 | 22 | 21 | 20 |
| 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.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **410710** | | | **10112** | | |
| **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:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **75310** | | | **17210** | | |
| **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 00012** |  | **Result:** | **1010 11002** |

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